



BSI Standards Publication

Programming languages — C++ Extensions for ranges

National foreword

This Published Document is the UK implementation of ISO/IEC TS 21425:2017.

The UK participation in its preparation was entrusted to Technical Committee IST/5, Programming languages, their environments and system software interfaces.

A list of organizations represented on this committee can be obtained on request to its secretary.

This publication does not purport to include all the necessary provisions of a contract. Users are responsible for its correct application.

© The British Standards Institution 2018
Published by BSI Standards Limited 2018

ISBN 978 0 580 52141 6

ICS 35.060

Compliance with a British Standard cannot confer immunity from legal obligations.

This Published Document was published under the authority of the Standards Policy and Strategy Committee on 30 April 2018.

Amendments/corrigenda issued since publication

Date	Text affected
------	---------------

Programming languages — C++ Extensions for ranges

Langages de programmation — Extensions C++ pour les «ranges»



Reference number
ISO/IEC TS 21425:2017(E)

AVA Documents
Document Sharing Hub
© ISO/IEC 2017



COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2017, Published in Switzerland

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized otherwise in any form or by any means, electronic or mechanical, including photocopying, or posting on the internet or an intranet, without prior written permission. Permission can be requested from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
Ch. de Blandonnet 8 • CP 401
CH-1214 Vernier, Geneva, Switzerland
Tel. +41 22 749 01 11
Fax +41 22 749 09 47
copyright@iso.org
www.iso.org

Contents

Foreword	v
1 Scope	1
2 Normative references	1
3 Terms and definitions	1
4 General principles	2
4.1 Implementation compliance	2
4.2 Namespaces, headers, and modifications to standard classes	2
5 Statements	3
5.1 Iteration statements	3
6 Library introduction	4
6.1 General	4
6.2 Method of description (Informative)	4
6.3 Library-wide requirements	6
7 Concepts library	8
7.1 General	8
7.2 Header <experimental/ranges/concepts> synopsis	9
7.3 Core language concepts	11
7.4 Comparison concepts	16
7.5 Object concepts	18
7.6 Callable concepts	19
8 General utilities library	21
8.1 General	21
8.2 Utility components	21
8.3 Function objects	22
8.4 Metaprogramming and type traits	26
8.5 Tagged tuple-like types	30
9 Iterators library	34
9.1 General	34
9.2 Header <experimental/ranges/iterator> synopsis	34
9.3 Iterator requirements	42
9.4 Indirect callable requirements	50
9.5 Common algorithm requirements	52
9.6 Iterator primitives	54
9.7 Iterator adaptors	58
9.8 Stream iterators	86
10 Ranges library	94
10.1 General	94
10.2 decay_copy	94
10.3 Header <experimental/ranges/range> synopsis	94
10.4 Range access	95
10.5 Range primitives	97
10.6 Range requirements	98

11 Algorithms library	101
11.1 General	101
11.2 Tag specifiers	117
11.3 Non-modifying sequence operations	118
11.4 Mutating sequence operations	123
11.5 Sorting and related operations	133
12 Numerics library	146
12.1 Uniform random number generator requirements	146
A Compatibility features	147
A.1 General	147
A.2 Rvalue range access	147
A.3 Range-and-a-half algorithms	147
B Acknowledgements	149
C Compatibility	150
C.1 C++ and Ranges	150
C.2 Ranges and the Palo Alto TR (N3351)	151
Bibliography	153
Index	154
Index of library names	155

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation on the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see the following URL: www.iso.org/iso/foreword.html.

This document was prepared by Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

Programming languages — C++ Extensions for ranges

1 Scope

[intro.scope]

- ¹ This document describes extensions to the C++ Programming Language (2) that permit operations on ranges of data. These extensions include changes and additions to the existing library facilities as well as the extension of one core language facility. In particular, changes and extensions to the Standard Library include:
- (1.1) — The formulation of the foundational and iterator concept requirements using the syntax of the Concepts TS (2).
 - (1.2) — Analogues of the Standard Library algorithms specified in terms of the new concepts.
 - (1.3) — The loosening of the algorithm constraints to permit the use of *sentinels* to denote the end of a range and corresponding changes to algorithm return types where necessary.
 - (1.4) — The addition of new concepts describing *range* and *view* abstractions; that is, objects with a begin iterator and an end sentinel.
 - (1.5) — New algorithm overloads that take range objects.
 - (1.6) — Support of *callable objects* (as opposed to *function objects*) passed as arguments to the algorithms.
 - (1.7) — The addition of optional *projection* arguments to the algorithms to permit on-the-fly data transformations.
 - (1.8) — Analogues of the iterator primitives and new primitives in support of the addition of sentinels to the library.
 - (1.9) — Constrained analogues of the standard iterator adaptors and stream iterators that satisfy the new iterator concepts.
 - (1.10) — New iterator adaptors (`counted_iterator` and `common_iterator`) and sentinels (`unreachable`).
- ² Changes to the core language include:
- (2.1) — the extension of the range-based `for` statement to support the new iterator range requirements (10.4).
- ³ This document does not specify constrained analogues of other parts of the Standard Library (e.g., the numeric algorithms), nor does it add range support to all the places that could benefit from it (e.g., the containers).
- ⁴ This document does not specify any new range views, actions, or facade or adaptor utilities; all are left as future work.

2 Normative references

[intro.refs]

- ¹ The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.
- (1.1) — ISO/IEC 14882:2014, *Programming Languages - C++*
 - (1.2) — ISO/IEC TS 19217:2015, *Programming Languages - C++ Extensions for Concepts*

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

3 Terms and definitions

[intro.defs]

For the purposes of this document, the terms and definitions given in ISO/IEC 14882:2014, ISO/IEC TS 19217:2015, and the following apply.

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

- ISO Online browsing platform: available at <http://www.iso.org/obp>

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

3.1

[defns.expr.equiv]

expression-equivalent

relationship that exists between two expressions E1 and E2 such that

- E1 and E2 have the same effects,
- `noexcept(E1) == noexcept(E2)`, and
- E1 is a constant subexpression — an expression whose evaluation as subexpression of a conditional-expression CE (ISO/IEC 14882:2014 §5.16) would not prevent CE from being a core constant expression (ISO/IEC 14882:2014 §5.19) — if and only if E2 is a constant subexpression

3.2

[defns.projection]

projection

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

[Example:

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

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

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

— end example]

4 General principles

[intro]

4.1 Implementation compliance

[intro.compliance]

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

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

- ¹ Since the extensions described in this document are experimental and not part of the C++ standard library, they should not be declared directly within namespace `std`. Unless otherwise specified, all components described in this document either:
 - (1.1) — modify an existing interface in the C++ Standard Library in-place,
 - (1.2) — are declared in namespace `std::experimental::ranges::v1`.
- ² The International Standard, ISO/IEC 14882, together with ISO/IEC TS 19217:2015 (the Concepts TS), provide important context and specification for this document. In places, this document suggests changes to be made to components in namespace `std` in-place. In other places, entire chapters and sections are copied from ISO/IEC 14882 and modified so as to define similar but different components in namespace `std::experimental::ranges::v1`.
- ³ Instructions to modify or add paragraphs are written as explicit instructions. Modifications made to existing text from the International Standard use underlining to represent added text and ~~strikethrough~~ to represent deleted text.
- ⁴ This document assumes that the contents of the `std::experimental::ranges::v1` namespace will become a new constrained version of the C++ Standard Library that will be delivered alongside the existing unconstrained version.
- ⁵ Unless otherwise specified, references to other entities described in this document are assumed to be qualified with `std::experimental::ranges::`, and references to entities described in the International Standard are assumed to be qualified with `std::`.
- ⁶ New header names are prefixed with `experimental/ranges/`. Where the final element of a new header name is the same as an existing standard header name (e.g., `<experimental/ranges/algorithm>`), the new header shall include the standard header as if by

#include <algorithm>

5 Statements

[stmt]

5.1 Iteration statements

[stmt.iter]

5.1.1 The range-based for statement

[stmt.ranged]

¹ [Note: This clause is presented as a set of differences to apply to ISO/IEC 14882:2014 §6.5.4 to allow differently typed begin and end iterators, as in C++17. — end note]

² For a range-based for statement of the form

for (*for-range-declaration* : *expression*) *statement*
let *range-init* be equivalent to the *expression* surrounded by parentheses
(*expression*)
and for a range-based for statement of the form

for (*for-range-declaration* : *braced-init-list*) *statement*
let *range-init* be equivalent to the *braced-init-list*. In each case, a range-based for statement is equivalent to

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

The range-based for statement

for (*for-range-declaration* : *for-range-initializer*) *statement*
is equivalent to

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

where

- (2.1) — if the *for-range-initializer* is an *expression*, it is regarded as if it were surrounded by parentheses (so that a comma operator cannot be reinterpreted as delimiting two *init-declarators*);
- (2.2) — *__range*, *__begin*, and *__end* are variables defined for exposition only; and ~~*__RangeT* is the type of the expression, and *begin-expr* and *end-expr* are determined as follows:~~
- (2.3) — *begin-expr* and *end-expr* are determined as follows:
 - (2.3.1) — if ~~*__RangeT*~~the *for-range-initializer* is an *expression* of array type *R*, *begin-expr* and *end-expr* are *__range* and *__range* + *__bound*, respectively, where *__bound* is the array bound. If ~~*__RangeT*~~ is an array of unknown *size* *bound* or an array of incomplete type, the program is ill-formed;

- (2.3.2) — if `__range` is an expression of class type `C`, the *unqualified-ids* `begin` and `end` are looked up in the scope of `class __rangeC` 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;
- (2.3.3) — otherwise, *begin-expr* and *end-expr* are `begin(__range)` and `end(__range)`, respectively, where `begin` and `end` are looked up in the associated namespaces (3.4.2). [*Note*: Ordinary unqualified lookup (3.4.1) is not performed. — *end note*]

[*Example*:

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

— *end example*]

- ³ In the *decl-specifier-seq* of a *for-range-declaration*, each *decl-specifier* shall be either a *type-specifier* or *constexpr*. The *decl-specifier-seq* shall not define a class or enumeration.

6 Library introduction

[library]

6.1 General

[library.general]

- ¹ This Clause describes the contents of the *Ranges library*, how a well-formed C++ program makes use of the library, and how a conforming implementation may provide the entities in the library.
- ² Clause 6.3, Clauses 7 through 12, and Annex Annex A specify the contents of the library, as well as library requirements and constraints on both well-formed C++ programs and conforming implementations.
- ³ Detailed specifications for each of the components in the library are in Clauses 7–12, as shown in Table 1.

Table 1 — Library categories

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

- ⁴ The concepts library (Clause 7) describes library components that C++ programs may use to perform compile-time validation of template parameters and perform function dispatch based on properties of types.
- ⁵ The general utilities library (Clause 8) includes components used by other library elements and components used as infrastructure in C++ programs, such as function objects.
- ⁶ The iterators library (Clause 9) describes components that C++ programs may use to perform iterations over containers (Clause ISO/IEC 14882:2014 §23), streams (ISO/IEC 14882:2014 §27.7), stream buffers (ISO/IEC 14882:2014 §27.6), and ranges (10).
- ⁷ The ranges library (Clause 10) describes components for dealing with ranges of elements.
- ⁸ The algorithms library (Clause 11) describes components that C++ programs may use to perform algorithmic operations on containers (Clause ISO/IEC 14882:2014 §23) and other sequences.
- ⁹ The numerics library (Clause 12) provides concepts that are useful to constrain numeric algorithms.

6.2 Method of description (Informative)

[description]

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

6.2.1 Structure of each clause

[structure]

6.2.1.1 Elements

[structure.elements]

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

- (1.1) — Summary
- (1.2) — Requirements
- (1.3) — Detailed specifications

6.2.1.2 Summary

[structure.summary]

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

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

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

- (3.1) — macros
- (3.2) — values
- (3.3) — types
- (3.4) — classes and class templates
- (3.5) — functions and function templates
- (3.6) — objects
- (3.7) — concepts

6.2.1.3 Requirements

[structure.requirements]

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

- (1.1) — Template arguments
- (1.2) — Derived classes
- (1.3) — Containers, iterators, and algorithms that meet an interface convention or satisfy a concept

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

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

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

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

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

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

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.

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

6.2.1.4 Detailed specifications**[structure.specifications]**

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

6.2.2 Other conventions**[conventions]**

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

6.2.2.1 Functions within classes**[functions.within.classes]**

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

6.2.2.2 Private members**[objects.within.classes]**

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

6.3 Library-wide requirements**[requirements]**

- ¹ This subclause specifies requirements that apply to the entire Ranges library. Clauses 7 through 12 and Annex Annex A specify the requirements of individual entities within the library.
- ² Requirements specified in terms of interactions between threads do not apply to programs having only a single thread of execution.
- ³ Within this subclause, 6.3.1 describes the library's contents and organization, 6.3.3 describes how well-formed C++ programs gain access to library entities, 6.3.4 describes constraints on well-formed C++ programs, and 6.3.5 describes constraints on conforming implementations.

6.3.1 Library contents and organization**[organization]**

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

6.3.1.1 Library contents**[contents]**

- ¹ The Ranges library provides definitions for the entities and macros specified in the Ranges library headers (6.3.2).
- ² All library entities are defined within an inline namespace `v1` within the namespace `std::experimental::ranges` or namespaces nested within namespace `std::experimental::ranges::v1`. It is unspecified whether names declared in a specific namespace are declared directly in that namespace or in an inline namespace inside that namespace.

6.3.2 Headers**[headers]**

- ¹ Each element of the Ranges library is declared or defined (as appropriate) in a header.
- ² The Ranges library provides the Ranges library headers, shown in Table 2.

Table 2 — Ranges TS library headers

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

6.3.3 Using the library**[using]****6.3.3.1 Overview****[using.overview]**

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

6.3.3.2 Headers**[using.headers]**

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

6.3.3.3 Linkage [using.linkage]

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

6.3.4 Constraints on programs [constraints]

6.3.4.1 Overview [constraints.overview]

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

6.3.4.2 Derived classes [derived.classes]

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

6.3.4.3 Other functions [res.on.functions]

- ¹ In certain cases (operations on types used to instantiate Ranges library template components), the Ranges library depends on components supplied by a C++ program. If these components do not meet their requirements, this document places no requirements on the implementation.
- ² In particular, the effects are undefined if an incomplete type (ISO/IEC 14882:2014 §3.9) is used as a template argument when instantiating a template component or evaluating a concept, unless specifically allowed for that component.

6.3.4.4 Function arguments [res.on.arguments]

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

6.3.4.5 Library object access [res.on.objects]

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

6.3.4.6 Requires paragraph [res.on.required]

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

6.3.4.7 Semantic requirements [res.on.requirements]

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

6.3.5 Conforming implementations [conforming]

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

6.3.5.1 Customization Point Objects [customization.point.object]

- ¹ A *customization point object* is a function object (8.3) with a literal class type that interacts with user-defined types while enforcing semantic requirements on that interaction.
- ² The type of a customization point object shall satisfy `Semiregular` (7.5.3).
- ³ All instances of a specific customization point object type shall be equal (7.1.1).
- ⁴ The type of a customization point object `T` shall satisfy `Invocable<const T, Args...>` (7.6.2) when the types of `Args...` meet the requirements specified in that customization point object's definition. Otherwise, `T` shall not have a function call operator that participates in overload resolution.
- ⁵ Each customization point object type constrains its return type to satisfy a particular concept.
- ⁶ The library defines several named customization point objects. In every translation unit where such a name is defined, it shall refer to the same instance of the customization point object.
- ⁷ [Note: Many of the customization point objects in the library evaluate function call expressions with an unqualified name which results in a call to a user-defined function found by argument dependent name

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

7 Concepts library [concepts.lib]

7.1 General [concepts.lib.general]

- ¹ This Clause describes library components that C++ programs may use to perform compile-time validation of template parameters and perform function dispatch based on properties of types. The purpose of these concepts is to establish a foundation for equational reasoning in programs.
- ² The following subclauses describe core language concepts, comparison concepts, object concepts, and function concepts as summarized in Table 3.

Table 3 — Fundamental concepts library summary

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

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

- ¹ An expression is *equality preserving* if, given equal inputs, the expression results in equal outputs. The inputs to an expression are the set of the expression's operands. The output of an expression is the expression's result and all operands modified by the expression.
- ² Not all input values must be valid for a given expression; e.g., for integers `a` and `b`, the expression `a / b` is not well-defined when `b` is 0. This does not preclude the expression `a / b` being equality preserving. The *domain* of an expression is the set of input values for which the expression is required to be well-defined.
- ³ Expressions required by this specification to be equality preserving are further required to be stable: two evaluations of such an expression with the same input objects must have equal outputs absent any explicit intervening modification of those input objects. [*Note*: This requirement allows generic code to reason about the current values of objects based on knowledge of the prior values as observed via equality preserving expressions. It effectively forbids spontaneous changes to an object, changes to an object from another thread of execution, changes to an object as side effects of non-modifying expressions, and changes to an object as side effects of modifying a distinct object if those changes could be observable to a library function via an equality preserving expression that is required to be valid for that object. — *end note*]
- ⁴ Expressions declared in a *requires-expression* in this document are required to be equality preserving, except for those annotated with the comment “not required to be equality preserving.” An expression so annotated may be equality preserving, but is not required to be so.
- ⁵ An expression that may alter the value of one or more of its inputs in a manner observable to equality preserving expressions is said to modify those inputs. This document uses a notational convention to specify which expressions declared in a *requires-expression* modify which inputs: except where otherwise specified, an expression operand that is a non-constant lvalue or rvalue may be modified. Operands that are constant lvalues or rvalues must not be modified.
- ⁶ Where a *requires-expression* declares an expression that is non-modifying for some constant lvalue operand, additional variations of that expression that accept a non-constant lvalue or (possibly constant) rvalue for the given operand are also required except where such an expression variation is explicitly required with differing semantics. These *implicit expression variations* must meet the semantic requirements of the declared expression. The extent to which an implementation validates the syntax of the variations is unspecified.

[*Example*:

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

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

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

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

had been declared as well.

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

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

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

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

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

// 7.5, object concepts:
// 7.5.1, Movable:
template <class T>

```

```

concept bool Movable = see below;

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

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

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

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

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

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

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

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

```

7.3 Core language concepts

[[concepts.lib.corelang](#)]

7.3.1 General

[[concepts.lib.corelang.general](#)]

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

7.3.2 Concept Same

[[concepts.lib.corelang.same](#)]

```

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

```

- 1 There need not be any subsumption relationship between `Same<T, U>` and `is_same<T, U>::value`.
- 2 *Remarks:* For the purposes of constraint checking, `Same<T, U>` implies `Same<U, T>`.

7.3.3 Concept DerivedFrom

[[concepts.lib.corelang.derived](#)]

```

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

```

- 1 There need not be any subsumption relationship between `DerivedFrom<T, U>` and either `is_base_of<U, T>::value` or `is_convertible<remove_cv_t<T>*, remove_cv_t<U>*>::value`.
- 2 [*Note:* `DerivedFrom<T, U>` is satisfied if and only if `T` is publicly and unambiguously derived from `U`, or `T` and `U` are the same class type ignoring *cv*-qualifiers. — *end note*]

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

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

¹ Let `test` be the invented function:

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

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

- (1.1) — `To` is not an object or reference-to-object type, or `static_cast<To>(f())` is equal to `test(f)`.
- (1.2) — `From` is not a reference-to-object type, or
- (1.2.1) — If `From` is an rvalue reference to a non const-qualified type, the resulting state of the object referenced by `f()` after either above expression is valid but unspecified (ISO/IEC 14882:2014 §17.6.5.15).
- (1.2.2) — Otherwise, the object referred to by `f()` is not modified by either above expression.

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

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

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

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

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

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

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

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

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

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

```

```
common_reference_t<
    add_lvalue_reference_t<const T>,
    add_lvalue_reference_t<const U>>>;
```

- 2 Let `C` be `common_type_t<T, U>`. Let `t` be a function whose return type is `T`, and let `u` be a function whose return type is `U`. `Common<T, U>` is satisfied only if:
- (2.1) — `C(t())` equals `C(t())` if and only if `t()` is an equality preserving expression (7.1.1).
 - (2.2) — `C(u())` equals `C(u())` if and only if `u()` is an equality preserving expression (7.1.1).
- 3 [Note: Users can customize the behavior of `Common` by specializing the `common_type` class template (8.4.2). — end note]

7.3.7 Concept Integral

[concepts.lib.corelang.integral]

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

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

7.3.8 Concept SignedIntegral

[concepts.lib.corelang.signedintegral]

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

- 1 There need not be any subsumption relationship between `SignedIntegral<T>` and `is_signed<T>::value`.
- 2 [Note: `SignedIntegral<T>` may be satisfied even for types that are not signed integral types (ISO/IEC 14882:2014 §3.9.1); for example, `char`. — end note]

7.3.9 Concept UnsignedIntegral

[concepts.lib.corelang.unsignedintegral]

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

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

7.3.10 Concept Assignable

[concepts.lib.corelang.assignable]

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

- 1 Let `t` be an lvalue that refers to an object `o` such that `decltype((t))` is `T`, and `u` an expression such that `decltype((u))` is `U`. Let `u2` be a distinct object that is equal to `u`. `Assignable<T, U>` is satisfied only if
- (1.1) — `addressof(t = u) == addressof(o)`.
 - (1.2) — After evaluating `t = u`:
 - (1.2.1) — `t` is equal to `u2`, unless `u` is a non-const xvalue that refers to `o`.
 - (1.2.2) — If `u` is a non-const xvalue, the resulting state of the object to which it refers is valid but unspecified (ISO/IEC 14882:2014 §17.6.5.15).
 - (1.2.3) — Otherwise, if `u` is a glvalue, the object to which it refers is not modified.

- 2 There need not be any subsumption relationship between `Assignable<T, U>` and `is_lvalue_reference<T>::value`.
- 3 [*Note:* Assignment need not be a total function (6.2.1.3); in particular, if assignment to an object `x` can result in a modification of some other object `y`, then `x = y` is likely not in the domain of `=`. — *end note*]

7.3.11 Concept Swappable

[`concepts.lib.corelang.swappable`]

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

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

- 1 This subclause provides definitions for swappable types and expressions. In these definitions, let `t` denote an expression of type `T`, and let `u` denote an expression of type `U`.
- 2 An object `t` is *swappable with* an object `u` if and only if `SwappableWith<T, U>` is satisfied. `SwappableWith<T, U>` is satisfied only if given distinct objects `t2` equal to `t` and `u2` equal to `u`, after evaluating either `ranges::swap(t, u)` or `ranges::swap(u, t)`, `t2` is equal to `u` and `u2` is equal to `t`.
- 3 An rvalue or lvalue `t` is *swappable* if and only if `t` is swappable with any rvalue or lvalue, respectively, of type `T`.
- [*Example:* User code can ensure that the evaluation of `swap` calls is performed in an appropriate context under the various conditions as follows:

```
#include <utility>

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

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

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

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

```

    void swap(Proxy p, A& x) { swap(x, p); }           // satisfy symmetry constraint
}

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

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

— end example]

```

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

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

```

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

```

- ² There need not be any subsumption relationship between **Destructible**<T> and **is_nothrow_destructible**<T>::value.
- ³ [Note: Unlike the **Destructible** library concept in the C++ Standard (ISO/IEC 14882:2014 §17.6.3.1), this concept forbids destructors that are **noexcept(false)**, even if non-throwing. — end note]

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

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

```

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

```

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

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

```

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

```

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

```

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

```

- ¹ If T is an object type, then let **rv** be an rvalue of type T and **u2** a distinct object of type T equal to **rv**. **MoveConstructible**<T> is satisfied only if
- (1.1) — After the definition **T u = rv**;, **u** is equal to **u2**.
- (1.2) — **T{rv}** is equal to **u2**.
- (1.3) — If T is not **const**, **rv**'s resulting state is valid but unspecified (ISO/IEC 14882:2014 §17.6.5.15); otherwise, it is unchanged.

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

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

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

- (1.1) — After the definition *T u = v*;, *u* is equal to *v*.
- (1.2) — *T{v}* is equal to *v*.

7.4 Comparison concepts**[concepts.lib.compare]****7.4.1 General****[concepts.lib.compare.general]**

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

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

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

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

² Given *const* lvalues *b1* and *b2* of type *remove_reference_t*, then *Boolean* is satisfied only if

- (2.1) — *bool(b1) == !bool(!b1)*.
- (2.2) — (*b1 && b2*), (*b1 && bool(b2)*), and (*bool(b1) && b2*) are all equal to (*bool(b1) && bool(b2)*), and have the same short-circuit evaluation.
- (2.3) — (*b1 || b2*), (*b1 || bool(b2)*), and (*bool(b1) || b2*) are all equal to (*bool(b1) || bool(b2)*), and have the same short-circuit evaluation.
- (2.4) — *bool(b1 == b2)*, *bool(b1 == bool(b2))*, and *bool(bool(b1) == b2)* are all equal to (*bool(b1) == bool(b2)*).
- (2.5) — *bool(b1 != b2)*, *bool(b1 != bool(b2))*, and *bool(bool(b1) != b2)* are all equal to (*bool(b1) != bool(b2)*).

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

7.4.3 Concept EqualityComparable

[concepts.lib.compare.equalitycomparable]

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

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

- (1.1) — $t == u$, $u == t$, $t != u$, and $u != t$ have the same domain.
- (1.2) — $\text{bool}(u == t) == \text{bool}(t == u)$.
- (1.3) — $\text{bool}(t != u) == !\text{bool}(t == u)$.
- (1.4) — $\text{bool}(u != t) == \text{bool}(t != u)$.

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

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

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

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

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

4 Let t be a const lvalue of type `remove_reference_t<T>`, u be a const lvalue of type `remove_reference_t<U>`, and C be:

`common_reference_t<const remove_reference_t<T>&, const remove_reference_t<U>&>`

`EqualityComparableWith<T, U>` is satisfied only if:

- (4.1) — $\text{bool}(t == u) == \text{bool}(C(t) == C(u))$.

7.4.4 Concept StrictTotallyOrdered

[concepts.lib.compare.stricttotallyordered]

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

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

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

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

2 Let `t` be a `const` lvalue of type `remove_reference_t<T>`, `u` be a `const` lvalue of type `remove_reference_t<U>`, and `C` be:

```
common_reference_t<const remove_reference_t<T>&, const remove_reference_t<U>&>
```

`StrictTotallyOrderedWith<T, U>` is satisfied only if

- (2.1) — `bool(t < u) == bool(C(t) < C(u))`.
- (2.2) — `bool(t > u) == bool(C(t) > C(u))`.
- (2.3) — `bool(t <= u) == bool(C(t) <= C(u))`.
- (2.4) — `bool(t >= u) == bool(C(t) >= C(u))`.
- (2.5) — `bool(u < t) == bool(C(u) < C(t))`.
- (2.6) — `bool(u > t) == bool(C(u) > C(t))`.
- (2.7) — `bool(u <= t) == bool(C(u) <= C(t))`.
- (2.8) — `bool(u >= t) == bool(C(u) >= C(t))`.

7.5 Object concepts

[`concepts.lib.object`]

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

7.5.1 Concept Movable

[concepts.lib.object.movable]

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

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

7.5.2 Concept Copyable

[concepts.lib.object.copyable]

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

7.5.3 Concept Semiregular

[concepts.lib.object.semiregular]

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

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

7.5.4 Concept Regular

[concepts.lib.object.regular]

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

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

7.6 Callable concepts

[concepts.lib.callable]

7.6.1 General

[concepts.lib.callable.general]

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

7.6.2 Concept Invocable

[concepts.lib.callable.invocable]

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

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

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

7.6.3 Concept RegularInvocable

[concepts.lib.callable.regularinvocable]

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

- 1 The `invoke` function call expression shall be equality-preserving and shall not modify the function object or the arguments (7.1.1). [*Note*: This requirement supersedes the annotation in the definition of `Invocable`. — *end note*]
- 2 [*Note*: A random number generator does not satisfy `RegularInvocable`. — *end note*]
- 3 [*Note*: The distinction between `Invocable` and `RegularInvocable` is purely semantic. — *end note*]

7.6.4 Concept Predicate

[`concepts.lib.callable.predicate`]

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

7.6.5 Concept Relation

[`concepts.lib.callable.relation`]

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

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

7.6.6 Concept StrictWeakOrder

[`concepts.lib.callable.strictweakorder`]

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

- 1 A `Relation` satisfies `StrictWeakOrder` only if it imposes a *strict weak ordering* on its arguments.
- 2 The term *strict* refers to the requirement of an irreflexive relation (`!comp(x, x)` for all `x`), and the term *weak* to requirements that are not as strong as those for a total ordering, but stronger than those for a partial ordering. If we define `equiv(a, b)` as `!comp(a, b) && !comp(b, a)`, then the requirements are that `comp` and `equiv` both be transitive relations:
- (2.1) — `comp(a, b) && comp(b, c)` implies `comp(a, c)`
- (2.2) — `equiv(a, b) && equiv(b, c)` implies `equiv(a, c)` [*Note*: Under these conditions, it can be shown that
- (2.2.1) — `equiv` is an equivalence relation
- (2.2.2) — `comp` induces a well-defined relation on the equivalence classes determined by `equiv`
- (2.2.3) — The induced relation is a strict total ordering. — *end note*]

8 General utilities library

[utilities]

8.1 General

[utilities.general]

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

Table 4 — General utilities library summary

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

8.2 Utility components

[utility]

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

Header <experimental/ranges/utility> synopsis

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

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {  
    // 8.2.1, swap:  
    namespace {  
        constexpr unspecified swap = unspecified ;  
    }  
  
    // 8.2.2, exchange:  
    template <MoveConstructible T, class U=T>  
        requires Assignable<T&, U>  
    constexpr T exchange(T& obj, U&& new_val) noexcept(see below);  
  
    // 8.5.2, struct with named accessors  
    template <class T>  
    concept bool TagSpecifier = see below;  
  
    template <class F>  
    concept bool TaggedType = see below;  
  
    template <class Base, TagSpecifier... Tags>  
        requires sizeof...(Tags) <= tuple_size<Base>::value  
    struct tagged;  
  
    // 8.5.4, tagged pairs  
    template <TaggedType T1, TaggedType T2> using tagged_pair = see below;  
  
    template <TagSpecifier Tag1, TagSpecifier Tag2, class T1, class T2>  
    constexpr see below make_tagged_pair(T1&& x, T2&& y);  
}}}  
  
namespace std {  
    // 8.5.3, tuple-like access to tagged  
    template <class Base, class... Tags>  
    struct tuple_size<experimental::ranges::tagged<Base, Tags...>>;
```

```

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

```

8.2.1 swap

[utility.swap]

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

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

```

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

```

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

- (1.2) — Otherwise, (void)**swap_ranges**(E1, E2) if E1 and E2 are lvalues of array types (ISO/IEC 14882:2014 §3.9.2) of equal extent and **ranges::swap**(*(E1), *(E2)) is a valid expression, except that **noexcept**(**ranges::swap**(E1, E2)) is equal to **noexcept**(**ranges::swap**(*(E1), *(E2))).
- (1.3) — Otherwise, if E1 and E2 are lvalues of the same type T which meets the syntactic requirements of **MoveConstructible**<T> and **Assignable**<T&, T>, exchanges the referenced values. **ranges::swap**(E1, E2) is a constant expression if the constructor selected by overload resolution for **T{std::move(E1)}** is a constexpr constructor and the expression **E1 = std::move(E2)** can appear in a constexpr function. **noexcept**(**ranges::swap**(E1, E2)) is equal to **is_nothrow_move_constructible**<T>::value && **is_nothrow_move_assignable**<T>::value. If either **MoveConstructible** or **Assignable** is not satisfied, the program is ill-formed with no diagnostic required.
- (1.4) — Otherwise, **ranges::swap**(E1, E2) is ill-formed.

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

8.2.2 exchange

[utility.exchange]

```

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

```

- ¹ *Effects:* Equivalent to:

```

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

```

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

```

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

```

8.3 Function objects

[function.objects]

- ¹ Header <experimental/ranges/functional> synopsis

```

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

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

```

```

struct equal_to;

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

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

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

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

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

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

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

```

8.3.1 Function template invoke

[func.invoke]

```

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

```

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

8.3.2 Comparisons

[comparisons]

- ¹ The library provides basic function object classes for all of the comparison operators in the language (ISO/IEC 14882:2014 §5.9, ISO/IEC 14882:2014 §5.10).
- ² In this section, *BUILTIN_PTR_CMP*(*T*, *op*, *U*) for types *T* and *U* and where *op* is an equality (ISO/IEC 14882:2014 §5.10) or relational operator (ISO/IEC 14882:2014 §5.9) is a boolean constant expression. *BUILTIN_PTR_CMP*(*T*, *op*, *U*) is true if and only if *op* in the expression *declval<T>()* *op* *declval<U>()* resolves to a built-in operator comparing pointers.
- ³ There is an implementation-defined strict total ordering over all pointer values of a given type. This total ordering is consistent with the partial order imposed by the builtin operators <, >, <=, and >=.

```

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

```

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

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

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

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

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

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

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

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

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

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

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

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

    typedef unspecified is_transparent;
};
```

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

11 *Effects:*

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

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

```

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

    typedef unspecified is_transparent;
};

```

12 `operator()` has effects equivalent to:

```

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

```

```

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

    typedef unspecified is_transparent;
};

```

13 `operator()` has effects equivalent to:

```

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

```

```

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

    typedef unspecified is_transparent;
};

```

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

15 *Effects:*

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

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

```

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

    typedef unspecified is_transparent;
};

```

16 `operator()` has effects equivalent to:

```

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

```

```

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

```

```
typedef unspecified is_transparent;
};
```

17 operator() has effects equivalent to:

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

8.3.3 Class identity

[func.identity]

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

    typedef unspecified is_transparent;
};
```

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

8.4 Metaprogramming and type traits

[meta]

8.4.1 Header <experimental/ranges/type_traits> synopsis

[meta.type.synop]

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

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

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

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

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

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

8.4.2 Type properties

[meta.unary.prop]

- 1 These templates provide access to some of the more important properties of types.
- 2 It is unspecified whether the library defines any full or partial specializations of any of these templates.
- 3 For all of the class templates *X* declared in this subclause, instantiating that template with a template argument that is a class template specialization may result in the implicit instantiation of the template argument if and only if the semantics of *X* require that the argument must be a complete type.
- 4 For the purpose of defining the templates in this subclause, a function call expression `declval<T>()` for any type *T* is considered to be a trivial (ISO/IEC 14882:2014 §3.9, ISO/IEC 14882:2014 §12) function call that is not an odr-use (ISO/IEC 14882:2014 §3.2) of `declval` in the context of the corresponding definition notwithstanding the restrictions of (ISO/IEC 14882:2014 §20.2.5).

Table 5 — Additional type property predicates

Template	Condition	Precondition
<pre>template <class T, class U> struct is_swappable_with;</pre>	The expressions <code>ranges::swap(declval<T>(), declval<U>())</code> and <code>ranges::swap(declval<U>(), declval<T>())</code> are each well-formed when treated as an unevaluated operand (Clause ISO/IEC 14882:2014 §5). Access checking is performed as if in a context unrelated to T and U. Only the validity of the immediate context of the <code>swap</code> expressions is considered. [<i>Note</i>: 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. — <i>end note</i>]	T and U shall be complete types, (possibly <i>cv</i>-qualified) <code>void</code>, or arrays of unknown bound.
<pre>template <class T> struct is_swappable;</pre>	For a referenceable type T, the same result as <code>is_swappable_with_v<T&, T&></code>, otherwise <code>false</code>.	T shall be a complete type, (possibly <i>cv</i>-qualified) <code>void</code>, or an array of unknown bound.
<pre>template <class T, class U> struct is_nothrow_swappable_with;</pre>	<code>is_swappable_with_v<T, U></code> is true and each <code>swap</code> expression of the definition of <code>is_swappable_with<T, U></code> is known not to throw any exceptions (ISO/IEC 14882:2014 §5.3.7).	T and U shall be complete types, (possibly <i>cv</i>-qualified) <code>void</code>, or arrays of unknown bound.
<pre>template <class T> struct is_nothrow_swappable;</pre>	For a referenceable type T, the same result as <code>is_nothrow_swappable_with_v<T&, T&></code>, otherwise <code>false</code>.	T shall be a complete type, (possibly <i>cv</i>-qualified) <code>void</code>, or an array of unknown bound.

8.4.3 Other transformations

[meta.trans.other]

Table 6 — Other transformations

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

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

- (1.1) — If A and B are both lvalue reference types, `COMMON_REF(A, B)` is `COND_RES(COPYCV(X, Y) &, COPYCV(Y, X) &)`.
- (1.2) — Otherwise, let C be `RREF_RES(COMMON_REF(X&, Y&))`. If A and B are both rvalue reference types, and C is well-formed, and `is_convertible<A, C>::value` and `is_convertible<B, C>::value` are true, then `COMMON_REF(A, B)` is C.
- (1.3) — Otherwise, let D be `COMMON_REF(const X&, Y&)`. If A is an rvalue reference and B is an lvalue reference and D is well-formed and `is_convertible<A, D>::value` is true, then `COMMON_REF(A, B)` is D.
- (1.4) — Otherwise, if A is an lvalue reference and B is an rvalue reference, then `COMMON_REF(A, B)` is `COMMON_REF(B, A)`.
- (1.5) — Otherwise, `COMMON_REF(A, B)` is `decay_t<COND_RES(CREF(A), CREF(B))>`.

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

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

- (2.1) — If `sizeof...(T)` is zero, there shall be no member **type**.
- (2.2) — Otherwise, if `sizeof...(T)` is one, let T1 denote the sole type in the pack T. The member typedef **type** shall denote the same type as `decay_t<T1>`.
- (2.3) — Otherwise, if `sizeof...(T)` is two, let T1 and T2 denote the two types in the pack T, and let D1 and D2 be `decay_t<T1>` and `decay_t<T2>` respectively. Then
 - (2.3.1) — If D1 and T1 denote the same type and D2 and T2 denote the same type, then
 - (2.3.1.1) — If `std::common_type_t<T1, T2>` is well-formed, then the member typedef **type** denotes `std::common_type_t<T1, T2>`.
 - (2.3.1.2) — If `COMMON_REF(T1, T2)` is well-formed, then the member typedef **type** denotes that type.

- (2.3.1.3) — Otherwise, there shall be no member `type`.
- (2.3.2) — Otherwise, if `common_type_t<D1, D2>` is well-formed, then the member typedef `type` denotes that type.
- (2.3.3) — Otherwise, there shall be no member `type`.
- (2.4) — Otherwise, if `sizeof...(T)` is greater than two, let `T1`, `T2`, and `Rest`, respectively, denote the first, second, and (pack of) remaining types comprising `T`. Let `C` be the type `common_type_t<T1, T2>`. Then:
 - (2.4.1) — If there is such a type `C`, the member typedef `type` shall denote the same type, if any, as `common_type_t<C, Rest...>`.
 - (2.4.2) — Otherwise, there shall be no member `type`.
- 3 Note B: Notwithstanding the provisions of ISO/IEC 14882:2014 §20.10.2, and pursuant to ISO/IEC 14882:2014 §17.6.4.2.1, a program may specialize `common_type_t<T1, T2>` for types `T1` and `T2` such that `is_same<T1, decay_t<T1>>::value` and `is_same<T2, decay_t<T2>>::value` are each `true`. [*Note:* Such specializations are needed when only explicit conversions are desired between the template arguments. — *end note*] Such a specialization need not have a member named `type`, but if it does, that member shall be a *typedef-name* for an accessible and unambiguous *cv*-unqualified non-reference type `C` to which each of the types `T1` and `T2` is explicitly convertible. Moreover, `common_type_t<T1, T2>` shall denote the same type, if any, as does `common_type_t<T2, T1>`. No diagnostic is required for a violation of this Note's rules.
- 4 For the `common_reference` trait applied to a parameter pack `T` of types, the member `type` shall be either defined or not present as follows:
 - (4.1) — If `sizeof...(T)` is zero, there shall be no member `type`.
 - (4.2) — Otherwise, if `sizeof...(T)` is one, let `T1` denote the sole type in the pack `T`. The member typedef `type` shall denote the same type as `T1`.
 - (4.3) — Otherwise, if `sizeof...(T)` is two, let `T1` and `T2` denote the two types in the pack `T`. Then
 - (4.3.1) — If `T1` and `T2` are reference types and `COMMON_REF(T1, T2)` is well-formed and denotes a reference type then the member typedef `type` denotes that type.
 - (4.3.2) — Otherwise, if `basic_common_reference<UNCVREF(T1), UNCVREF(T2), XREF(T1), XREF(T2)>::type` is well-formed, then the member typedef `type` denotes that type.
 - (4.3.3) — Otherwise, if `COND_RES(T1, T2)` is well-formed, then the member typedef `type` denotes that type.
 - (4.3.4) — Otherwise, if `common_type_t<T1, T2>` is well-formed, then the member typedef `type` denotes that type.
 - (4.3.5) — Otherwise, there shall be no member `type`.
 - (4.4) — Otherwise, if `sizeof...(T)` is greater than two, let `T1`, `T2`, and `Rest`, respectively, denote the first, second, and (pack of) remaining types comprising `T`. Let `C` be the type `common_reference_t<T1, T2>`. Then:
 - (4.4.1) — If there is such a type `C`, the member typedef `type` shall denote the same type, if any, as `common_reference_t<C, Rest...>`.
 - (4.4.2) — Otherwise, there shall be no member `type`.
- 5 Notwithstanding the provisions of ISO/IEC 14882:2014 §20.10.2, and pursuant to ISO/IEC 14882:2014 §17.6.4.2.1, a program may specialize `basic_common_reference<T, U, TQual, UQual>` for types `T` and `U` such that `is_same<T, decay_t<T>>::value` and `is_same<U, decay_t<U>>::value` are each `true`. [*Note:* Such specializations are needed when only explicit conversions are desired between the template arguments. — *end note*] Such a specialization need not have a member named `type`, but if it does, that member shall be a *typedef-name* for an accessible and unambiguous type `C` to which each of the types `TQual<T>` and `UQual<U>` is convertible. Moreover, `basic_common_reference<T, U, TQual, UQual>::type` shall denote the same type, if any, as does `basic_common_reference<U, T, UQual, TQual>::type`. A program may not specialize `basic_common_reference` on the third or fourth parameters, `TQual` or `UQual`. No diagnostic is required for a violation of these rules.

8.5 Tagged tuple-like types

[taggedtuple]

8.5.1 General

[taggedtuple.general]

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

8.5.2 Class template `tagged`

[taggedtuple.tagged]

- ¹ Class template `tagged` augments a tuple-like class type (e.g., `pair` (ISO/IEC 14882:2014 §20.3), `tuple` (ISO/IEC 14882:2014 §20.4)) by giving it named accessors. It is used to define the alias templates `tagged_pair` (8.5.4) and `tagged_tuple` (8.5.5).
- ² In the class synopsis below, let i be in the range $[0, \text{sizeof} \dots (\text{Tags}))$ and T_i be the i^{th} type in `Tags`, where indexing is zero-based.

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

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

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

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

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

getter's member functions is called its *ElementIndex*. The name of the getter's member function is called its *ElementName*

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

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

- 5 A *tag specifier* is a type that facilitates a mapping from a tuple-like type and an element index into a *tagged getter* that gives named access to the element at that index. `TagSpecifier<T>` is satisfied if and only if *T* is a tag specifier. The tag specifiers in the `Tags` parameter pack shall be unique. [Note: The mapping mechanism from tag specifier to tagged getter is unspecified. — end note]
- 6 Let *TAGGET*(*D*, *T*, *N*) name a tagged getter type that gives named access to the *N*-th element of the tuple-like type *D*.
- 7 It shall not be possible to delete an instance of class template `tagged` through a pointer to any base other than `Base`.
- 8 `TaggedType<F>` is satisfied if and only if *F* is a unary function type with return type *T* which satisfies `TagSpecifier<T>`. Let *TAGSPEC*(*F*) name the tag specifier of the `TaggedType` *F*, and let *TAGELEM*(*F*) name the argument type of the `TaggedType` *F*.

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

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

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

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

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

- 11 *Effects*: Initializes `Base` with `that`.

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

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

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

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

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

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

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

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

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

16 *Effects:* Assigns `static_cast<Other&&>(that)` to `static_cast<Base&>(*this)`.
 17 *Returns:* `*this`.
 18 *Remarks:* The expression in the `noexcept` is equivalent to:
 `is_nothrow_assignable<Base&, Other>::value`

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

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

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

21 *Effects:* Assigns `std::forward<U>(u)` to `static_cast<Base&>(*this)`.
 22 *Returns:* `*this`.
 23 *Remarks:* The expression in the `noexcept` is equivalent to:
 `is_nothrow_assignable<Base&, U>::value`

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

24 *Effects:* Calls `swap` on the result of applying `static_cast` to `*this` and `that`.
 25 *Throws:* Nothing unless the call to `swap` on the `Base` sub-objects throws.
 26 *Remarks:* The expression in the `noexcept` is equivalent to:
 `noexcept(swap(declval<Base&>(), declval<Base&>()))`

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

27 *Effects:* Equivalent to `lhs.swap(rhs)`.
 28 *Remarks:* The expression in the `noexcept` is equivalent to:
 `noexcept(lhs.swap(rhs))`

8.5.3 Tuple-like access to tagged

[tagged.astuple]

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

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

8.5.4 Alias template `tagged_pair`

[tagged.pairs]

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

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

1 [Example:

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

— end example]

8.5.4.1 Tagged pair creation functions

[tagged.pairs.creation]

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

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

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

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

3 [Example: In place of:

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

a C++ program may contain:

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

— end example]

8.5.5 Alias template `tagged_tuple`

[tagged.tuple]

1 Header <experimental/ranges/tuple> synopsis

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {  
    template <TaggedType... Types>  
    using tagged_tuple = tagged<tuple<TAGELEM(Types)...>,  
                                TAGSPEC(Types)...>;  
  
    template <TagSpecifier... Tags, class... Types>  
    requires sizeof...(Tags) == sizeof...(Types)  
    constexpr see below make_tagged_tuple(Types&&... t);  
}}}}
```

2

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

3 [Example:

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

— end example]

8.5.5.1 Tagged tuple creation functions

[tagged.tuple.creation]

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

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

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

³ [*Example:*

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

creates a tagged tuple of type

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

— *end example*]

9 Iterators library

[iterators]

9.1 General

[iterators.general]

- ¹ This Clause describes components that C++ programs may use to perform iterations over containers (Clause ISO/IEC 14882:2014 §23), streams (ISO/IEC 14882:2014 §27.7), and stream buffers (ISO/IEC 14882:2014 §27.6).
- ² The following subclauses describe iterator requirements, and components for iterator primitives, predefined iterators, and stream iterators, as summarized in Table 7.

Table 7 — Iterators library summary

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

9.2 Header <experimental/ranges/iterator> synopsis

[iterator.synopsis]

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

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

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

  // 9.3.3, associated types:
  // 9.3.3.1, difference_type:
  template <class> struct difference_type;
```

```

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

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

// 9.3.3.3, iterator_category:
template <class> struct iterator_category;
template <class T> using iterator_category_t
    = typename iterator_category<T>::type;

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

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

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

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

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

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

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

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

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

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

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

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

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

// 9.3.14, BidirectionalIterator:

```

```

template <class I>
concept bool BidirectionalIterator = see below;

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

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

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

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

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

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

template <class> struct indirect_result_of;

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

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

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

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

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

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

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

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

// 9.5.4 IndirectlySwappable:
template <class I1, class I2 = I1>
concept bool IndirectlySwappable = see below;

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

```

```

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

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

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

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

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

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

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

// 9.7, predefined iterators and sentinels:

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

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

```

```

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

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

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

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

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

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

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

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

```

```

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

template <Semiregular S> class move_sentinel;

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

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

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

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

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

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

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

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

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

// 9.7.5, default sentinels:
class default_sentinel;

```

// 9.7.6, counted iterators:

```
template <Iterator I> class counted_iterator;

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

// 9.7.8, unreachable sentinels:

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

// 9.7.7, dangling wrapper:

```
template <class T> class dangling;
```

```

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

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

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

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

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

```

9.3 Iterator requirements

[iterator.requirements]

9.3.1 General

[iterator.requirements.general]

- ¹ Iterators are a generalization of pointers that allow a C++ program to work with different data structures (for example, containers and ranges) in a uniform manner. To be able to construct template algorithms that work correctly and efficiently on different types of data structures, the library formalizes not just the interfaces but also the semantics and complexity assumptions of iterators. All input iterators *i* support the expression `*i`, resulting in a value of some object type *T*, called the *value type* of the iterator. All output iterators support the expression `*i = o` where *o* is a value of some type that is in the set of types that are *writable* to the particular iterator type of *i*. For every iterator type *X* there is a corresponding signed integer type called the *difference type* of the iterator.
- ² Since iterators are an abstraction of pointers, their semantics are a generalization of most of the semantics of pointers in C++. This ensures that every function template that takes iterators works as well with regular pointers. This document defines five categories of iterators, according to the operations defined on them: *input iterators*, *output iterators*, *forward iterators*, *bidirectional iterators* and *random access iterators*, as shown in Table 8.

Table 8 — Relations among iterator categories

Random Access	→	Bidirectional	→	Forward	→	Input
						→ Output

- ³ The five categories of iterators correspond to the iterator concepts `InputIterator`, `OutputIterator`, `ForwardIterator`, `BidirectionalIterator`, and `RandomAccessIterator`, respectively. The generic term *iterator* refers to any type that satisfies `Iterator`.
- ⁴ Forward iterators satisfy all the requirements of input iterators and can be used whenever an input iterator is specified; Bidirectional iterators also satisfy all the requirements of forward iterators and can be used whenever a forward iterator is specified; Random access iterators also satisfy all the requirements of bidirectional iterators and can be used whenever a bidirectional iterator is specified.
- ⁵ Iterators that further satisfy the requirements of output iterators are called *mutable iterators*. Nonmutable iterators are referred to as *constant iterators*.
- ⁶ Just as a regular pointer to an array guarantees that there is a pointer value pointing past the last element of the array, so for any iterator type there is an iterator value that points past the last element of a corresponding sequence. These values are called *past-the-end* values. Values of an iterator *i* for which the expression `*i` is defined are called *dereferenceable*. The library never assumes that past-the-end values are dereferenceable. Iterators can also have singular values that are not associated with any sequence. [Example: After the declaration of an uninitialized pointer *x* (as with `int* x;`), *x* must always be assumed to have a singular value of a pointer. — end example] Results of most expressions are undefined for singular values; the only exceptions are destroying an iterator that holds a singular value, the assignment of a non-singular value to an iterator that holds a singular value, and using a value-initialized iterator as the source of a copy or move operation. [Note: This guarantee is not offered for default initialization, although the distinction only matters for types with trivial default constructors such as pointers or aggregates holding pointers. — end note] In these cases the singular value is overwritten the same way as any other value. Dereferenceable values are always non-singular.
- ⁷ Most of the library's algorithmic templates that operate on data structures have interfaces that use ranges. A range is an iterator and a *sentinel* that designate the beginning and end of the computation, or an iterator and a count that designate the beginning and the number of elements to which the computation is to be applied.
- ⁸ An iterator and a sentinel denoting a range are comparable. The types of a sentinel and an iterator that denote a range must satisfy `Sentinel` (9.3.9). A range `[i,s)` is empty if `i == s`; otherwise, `[i,s)` refers to the elements in the data structure starting with the element pointed to by *i* and up to but not including the element pointed to by the first iterator *j* such that `j == s`.
- ⁹ A sentinel *s* is called *reachable* from an iterator *i* if and only if there is a finite sequence of applications of the expression `++i` that makes `i == s`. If *s* is reachable from *i*, `[i,s)` denotes a range.

- ¹⁰ A counted range `[i,n)` is empty if `n == 0`; otherwise, `[i,n)` refers to the `n` elements in the data structure starting with the element pointed to by `i` and up to but not including the element pointed to by the result of incrementing `i` `n` times.
- ¹¹ A range `[i,s)` is valid if and only if `s` is reachable from `i`. A counted range `[i,n)` is valid if and only if `n == 0`; or `n` is positive, `i` is dereferenceable, and `[++i,--n)` is valid. The result of the application of functions in the library to invalid ranges is undefined.
- ¹² All the categories of iterators require only those functions that are realizable for a given category in constant time (amortized).
- ¹³ Destruction of an iterator may invalidate pointers and references previously obtained from that iterator.
- ¹⁴ An *invalid* iterator is an iterator that may be singular.³

9.3.2 Customization points

[iterator.custpoints]

9.3.2.1 `iter_move`

[iterator.custpoints.iter_move]

- ¹ The name `iter_move` denotes a *customization point object* (6.3.5.1). The expression `ranges::iter_move(E)` for some subexpression `E` is expression-equivalent to the following:
- (1.1) — `static_cast<decltype(iter_move(E))>(iter_move(E))`, if that expression is well-formed when evaluated in a context that does not include `ranges::iter_move` but does include the lookup set produced by argument-dependent lookup (ISO/IEC 14882:2014 §3.4.2).
 - (1.2) — Otherwise, if the expression `*E` is well-formed:
 - (1.2.1) — if `*E` is an lvalue, `std::move(*E)`;
 - (1.2.2) — otherwise, `static_cast<decltype(*E)>(*E)`.
 - (1.3) — Otherwise, `ranges::iter_move(E)` is ill-formed.
- ² If `ranges::iter_move(E)` does not equal `*E`, the program is ill-formed with no diagnostic required.

9.3.2.2 `iter_swap`

[iterator.custpoints.iter_swap]

- ¹ The name `iter_swap` denotes a *customization point object* (6.3.5.1). The expression `ranges::iter_swap(E1, E2)` for some subexpressions `E1` and `E2` is expression-equivalent to the following:
- (1.1) — `(void)iter_swap(E1, E2)`, if that expression is well-formed when evaluated in a context that does not include `ranges::iter_swap` but does include the lookup set produced by argument-dependent lookup (ISO/IEC 14882:2014 §3.4.2) and the following declaration:


```
void iter_swap(auto, auto) = delete;
```
 - (1.2) — Otherwise, if the types of `E1` and `E2` both satisfy `Readable`, and if the reference type of `E1` is swappable with (7.3.11) the reference type of `E2`, then `ranges::swap(*E1, *E2)`
 - (1.3) — Otherwise, if the types `T1` and `T2` of `E1` and `E2` satisfy `IndirectlyMovableStorable<T1, T2> && IndirectlyMovableStorable<T2, T1>`, `(void)(*E1 = iter_exchange_move(E2, E1))`, except that `E1` is evaluated only once.
 - (1.4) — Otherwise, `ranges::iter_swap(E1, E2)` is ill-formed.
- ² If `ranges::iter_swap(E1, E2)` does not swap the values denoted by the expressions `E1` and `E2`, the program is ill-formed with no diagnostic required.
- ³ `iter_exchange_move` is an exposition-only function specified as:

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

- ⁴ *Effects:* Equivalent to:

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

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