TECHNICAL SPECIFICATION

ISO/IEC TS 21425

First edition
2017-11

Programming languages — C++ Extensions for ranges

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

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

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

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`~~`R` is an array of unknown ~~size~~bound or an array of incomplete type, the program is ill-formed;

(2.3.2) — if ~~_RangeT~~the *for-range-initializer* is ~~a~~an expression of class type C, the *unqualified-ids* `begin` and `end` are looked up in the scope of ~~class _RangeT~~C as if by class member access lookup (3.4.5), and if either (or both) finds at least one declaration, *begin-expr* and *end-expr* are `__range.begin()` and `__range.end()`, respectively;

(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

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

6.3.3 Using the library [using]

6.3.3.1 Overview [using.overview]

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

6.3.3.2 Headers [using.headers]

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

6.3.3.3 Linkage [using.linkage]

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

6.3.4 Constraints on programs [constraints]

6.3.4.1 Overview [constraints.overview]

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

6.3.4.2 Derived classes [derived.classes]

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

6.3.4.3 Other functions [res.on.functions]

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

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

6.3.4.4 Function arguments [res.on.arguments]

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

6.3.4.5 Library object access [res.on.objects]

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

6.3.4.6 Requires paragraph [res.on.required]

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

6.3.4.7 Semantic requirements [res.on.requirements]

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

6.3.5 Conforming implementations [conforming]

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

6.3.5.1 Customization Point Objects [customization.point.object]

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

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

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

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

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

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

7 [*Note:* Many of the customization point objects in the library evaluate function call expressions with an unqualified name which results in a call to a user-defined function found by argument dependent name

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

7 Concepts library [concepts.lib]

7.1 General [concepts.lib.general]

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

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

Table 3 — Fundamental concepts library summary

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

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

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

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

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

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

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

6 Where a *requires-expression* declares an expression that is non-modifying for some constant lvalue operand, additional variations of that expression that accept a non-constant lvalue or (possibly constant) rvalue for the given operand are also required except where such an expression variation is explicitly required with differing semantics. These *implicit expression variations* must meet the semantic requirements of the declared expression. The extent to which an implementation validates the syntax of the variations is unspecified.

[*Example:*

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

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

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

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

had been declared as well.

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

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

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


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

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

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

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

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

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

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

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