Document Number: P0898R0
Date: 2018-02-12
Reply to: Casey Carter
casey@carter.net
Audience: Library Evolution Working Group,
Library Working Group

Standard Library Concepts

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

Contents

1 Introduction [intro]

Proposal P0802R0 "Applying Concepts to the Standard Library" and the LEWG discussion thereof captured in P0872R0 "Discussion Summary: Applying Concepts to the Standard Library" call for a proposal to insert the concepts library from the Ranges TS into the C++20 WD. This is that proposal.

The motivating discussion from P0802R0 suggests that the Ranges TS can provide a basis of concepts for use in other library work, so we can avoid a string of proposals that all define small variations on common ideas:

> How can the C++ Concepts core language feature be best applied to the standard library?
>
> It seems clear that the basis for introducing concepts to the standard library must be the Ranges TS. That paper encapsulates the committee's knowledge and experience with fundamental library concepts and how these library concepts can be applied to improve the existing standard library. The Ranges TS has been implemented and exposed to the C++ community for several years; any other approach would be pure invention and speculation.
>
> The Ranges TS has two separable components: a library of fundamental concepts (TS Clauses 6 and 7), and revisions of existing library components (TS Clauses 8-12, also known as STL2). The characteristics of these two components are quite different, so they should be considered and adopted separately.

This proposal includes the "library of fundamental concepts," the "revisions of existing library components" are in the sister proposal P0896. Again quoting P0802R0:

> **Recommendation: Fundamental Library Concepts**
>
> Ranges TS clause 7 (Concepts library) should adopted by the C++20 WP as soon as a proposal can be prepared and processed by LEWG/LWG. We recommend that Casey Carter and Eric Niebler lead this effort and that they be given sufficient authority to include other fundamental material from the Ranges TS.
>
> **Rationale:** The fundamental concepts are mature and well-known, as they are based on standard library requirements that have been developed and refined from C++98 onward. Because concepts are an entirely new core language feature, these fundamental concepts can be defined in the standard library without breaking any existing C++ code (modulo the usual namespace caveats). Furthermore, failure to standardize these fundamental concepts quickly is likely to result in proliferation of similar but subtly different user-supplied concepts, often with the same names. Confusion seems inevitable under such circumstances.

This document proposes the following parts of the Ranges TS for inclusion in C++20:

— The Concepts library (Clause 7) to be defined in namespace `std` inside a new `<concepts>` header

— Portions of the utilities library which do not break existing code: the `std2::swap` customization point which supports the `Swappable` concept, the `identity` function object, changes to `common_type` and the addition of the `common_reference` type trait

— The numerics library (which consists of only the `UniformRandomBitGenerator` concept)

Some of the library concepts introduced share the names of requirement tables defined in [utility.arg.requirements]; the names of those requirement tables are changed to "make way".

1.1 Renaming "requirements tables" [intro.stl1]

1 [Editor's note: Before applying the changes in the remainder of this specification, prepend the prefix "STL1" to uses of the names below in the Standard Library clauses:]

(1.1) — `EqualityComparable`

(1.2) — `DefaultConstructible`

(1.3) — `MoveConstructible`

(1.4) — `CopyConstructible`

(1.5) — `MoveAssignable`

(1.6) — `CopyAssignable`

(1.7) — `Destructible`

This document reuses these names for concept definitions.

[Editor's note: What about "swappable"/"swappable with"/"swappable requirements"?]

1.2 Style of presentation [intro.style]

1 The remainder of this document is a technical specification in the form of editorial instructions directing that changes be made to the text of the C++ working draft. The formatting of the text suggests the origin of each portion of the wording.

Existing wording from the C++ working draft - included to provide context - is presented without decoration.

Entire clauses / subclauses / paragraphs incorporated from the ISO/IEC 21425:2017 (the "Ranges TS") are presented in a distinct cyan color.

In-line additions of wording from the Ranges TS to the C++ working draft are presented in cyan with underline.

~~In-line bits of wording to be struck from the C++ working draft are presented in red with strike-through.~~

Wording to be added which is original to this document appears in gold with underline.

~~Wording from the Ranges TS which IS NOT to be added to the C++ working draft is presented in magenta with strikethrough.~~

Ideally, these formatting conventions make it clear which wording comes from which document in this three-way merge.

20 Library introduction [library]

20.1 General [library.general]

[Editor's note: Modify Table 15 as follows (note that the consequent renumbering of the clauses following the newly-inserted "Concepts library" is NOT depicted here or in the remainder of this document for ease of review):]

Table 15 — Library categories

Clause	Category
Clause 21	Language support library
Clause 22	Concepts library
Clause 22	Diagnostics library
Clause 23	General utilities library
Clause 24	Strings library
Clause 25	Localization library
Clause 26	Containers library
Clause 27	Iterators library
Clause 28	Algorithms library
Clause 29	Numerics library
Clause 30	Input/output library
Clause 31	Regular expressions library
Clause 32	Atomic operations library
Clause 33	Thread support library

[Editor's note: Add a new paragraph between paragraphs 4 and 5:]

5 The concepts library (Clause 22) 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.

20.3 Definitions [definitions]

[Editor's note: Add a new definition for "expression-equivalent":]

20.3.11 [defns.expression.equivalent]

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

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

— `noexcept(E1) == noexcept(E2)`, and

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

20.4 Method of description (Informative) [description]

20.4.1 Structure of each clause [structure]

20.4.1.2 Summary [structure.summary]

[Editor's note: Add a new bullet to the list in paragraph 2:]

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

(2.1) — macros

(2.2) — values

(2.3) — types

(2.4) — classes and class templates

(2.5) — functions and function templates

(2.6) — objects

(2.7) — concepts

20.4.1.3 Requirements [structure.requirements]

[Editor's note: Modify paragraph 1 as follows:]

1 Requirements describe constraints that shall be met by a C++ program that extends the standard 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

[Editor's note: Modify paragraph 4 as follows:]

4 Requirements are stated in terms of well-defined expressions that define valid terms of the types that satisfy the requirements. For every set of well-defined expression requirements there is either a named concept or a table that specifies an initial set of the valid expressions and their semantics. Any generic algorithm (28) that uses the well-defined expression requirements is described in terms of the valid expressions for its template type parameters.

[Editor's note: Add new paragraphs after the existing paragraphs:]

7 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 (22.4.4) 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.

8 A declaration may explicitly impose requirements through its associated constraints (17.4.2). When the associated constraints refer to a concept (17.6.8), additional semantic requirements are imposed on the use of the declaration.

20.4.2 Other conventions [conventions]

20.4.2.1 Type descriptions [type.descriptions]

[Editor's note: Add a new subclause after [character.seq]:]

20.4.2.1.6 Customization Point Object types [customization.point.object]

1 A *customization point object* is a function object (23.14) 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` (22.5.3).

3 All instances of a specific customization point object type shall be equal (22.1.1).

4 The type of a customization point object `T` shall satisfy `Invocable<const T&, Args...>` (22.6.2) when the types of `Args...` meet the requirements specified in that customization point object's definition. ~~Otherwise~~When the types of `Args...` do not meet the customization point object's requirements, `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 (6.4.2). 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 (17.5.6.2) or more constrained (17.4.4) to be used by a customization point object. — *end note*]

20.5 Library-wide requirements [requirements]

20.5.1.1 Library contents [contents]

[Editor's note: Modify paragraphs 2 and 3 as follows:]

2 All library entities except `operator new` and `operator delete` are defined within the namespace `std` or `std2` or namespaces nested within namespace `std` or `std2`.[1] It is unspecified whether names declared in a specific namespace are declared directly in that namespace or in an inline namespace inside that namespace.[2]

3 Whenever a name `x` defined in the standard library is mentioned, the name `x` is assumed to be fully qualified as `::std::x`, if `x` is defined in namespace `std`, or `::std2::x`, if `x` is defined in namespace `std2`, unless explicitly described otherwise. For example, if the *Effects:* element for library function `F` is described as calling library function `G` which is defined in namespace `std`, the function `::std::G` is meant.

20.5.1.2 Headers [headers]

[Editor's note: Add header `<concepts>` to Table 16]

Table 16 — C++ library headers

`<algorithm>`	`<fstream>`	`<new>`	`<string_view>`
`<any>`	`<functional>`	`<numeric>`	`<strstream>`
`<array>`	`<future>`	`<optional>`	`<syncstream>`
`<atomic>`	`<initializer_list>`	`<ostream>`	`<system_error>`
`<bitset>`	`<iomanip>`	`<queue>`	`<thread>`
`<charconv>`	`<ios>`	`<random>`	`<tuple>`
`<chrono>`	`<iosfwd>`	`<ratio>`	`<type_traits>`
`<codecvt>`	`<iostream>`	`<regex>`	`<typeindex>`
`<compare>`	`<istream>`	`<scoped_allocator>`	`<typeinfo>`
`<complex>`	`<iterator>`	`<set>`	`<unordered_map>`
`<concepts>`	`<limits>`	`<shared_mutex>`	`<unordered_set>`
`<condition_variable>`	`<list>`	`<sstream>`	`<utility>`
`<deque>`	`<locale>`	`<stack>`	`<valarray>`
`<exception>`	`<map>`	`<stdexcept>`	`<variant>`
`<execution>`	`<memory>`	`<streambuf>`	
`<filesystem>`	`<memory_resource>`	`<vector>`	
`<forward_list>`	`<mutex>`	`<string>`	

20.5.4 Constraints on programs [constraints]

20.5.4.8 Other functions [res.on.functions]

[Editor's note: Modify paragraph 2 as follows:]

2 In particular, the effects are undefined in the following cases:

(2.1) — for replacement functions (21.6.2), if the installed replacement function does not implement the semantics of the applicable *Required behavior:* paragraph.

(2.2) — for handler functions (21.6.3.3, 21.8.4.1), if the installed handler function does not implement the semantics of the applicable *Required behavior:* paragraph

(2.3) — for types used as template arguments when instantiating a template component, if the operations on the type do not implement the semantics of the applicable *Requirements* subclause (20.5.3.5, 26.2, 27.2, 28.3, 29.3). Operations on such types can report a failure by throwing an exception unless otherwise specified.

(2.4) — if any replacement function or handler function or destructor operation exits via an exception, unless specifically allowed in the applicable *Required behavior:* paragraph.

(2.5) — if an incomplete type (6.7) is used as a template argument when instantiating a template component or evaluating a concept, unless specifically allowed for that component.

[Editor's note: Add a new subclause after [res.on.required]:]

1) The C standard library headers (D.5) also define names within the global namespace, while the C++ headers for C library facilities (20.5.1.2) may also define names within the global namespace.

2) This gives implementers freedom to use inline namespaces to support multiple configurations of the library.

20.5.4.12 Semantic requirements [res.on.requirements]

1 If the semantic requirements of a declaration's constraints (20.4.1.3) are not satisfied at the point of use, the program is ill-formed, no diagnostic required.

22 Concepts library [concepts.lib]

[Editor's note: Add new Clause "Concepts library"]

22.1 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 callable concepts as summarized in Table 33.

Table 33 — Fundamental concepts library summary

	Subclause	Header(s)
22.3	Core language concepts	`<concepts>`
22.4	Comparison concepts	
22.5	Object concepts	
22.6	Callable concepts	

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

[Editor's note: Consider relocating this subclause into [description], somewhere near [structure.requirements].]

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 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*]

22.2 Header `<concepts>` synopsis [concepts.lib.synopsis]

```
namespace std {
  // 22.3, core language concepts:
  // 22.3.2, Same:
  template <class T, class U>
  concept Same = see below;

  // 22.3.3, DerivedFrom:
  template <class Derived, class Base>
  concept DerivedFrom = see below;

  // 22.3.4, ConvertibleTo:
  template <class From, class To>
  concept ConvertibleTo = see below;

  // 22.3.5, CommonReference:
  template <class T, class U>
  concept CommonReference = see below;

  // 22.3.6, Common:
  template <class T, class U>
  concept Common = see below;

  // 22.3.7, Integral:
  template <class T>
  concept Integral = see below;
```

```
// 22.3.8, SignedIntegral:
template <class T>
concept SignedIntegral = see below;

// 22.3.9, UnsignedIntegral:
template <class T>
concept UnsignedIntegral = see below;

// 22.3.10, Assignable:
template <class LHS, class RHS>
concept Assignable = see below;

// 22.3.11, Swappable:
template <class T>
concept Swappable = see below;

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

// 22.3.12, Destructible:
template <class T>
concept Destructible = see below;

// 22.3.13, Constructible:
template <class T, class... Args>
concept Constructible = see below;

// 22.3.14, DefaultConstructible:
template <class T>
concept DefaultConstructible = see below;

// 22.3.15, MoveConstructible:
template <class T>
concept MoveConstructible = see below;

// 22.3.16, CopyConstructible:
template <class T>
concept CopyConstructible = see below;

// 22.4, comparison concepts:
// 22.4.2, Boolean:
template <class B>
concept Boolean = see below;

// 22.4.3, EqualityComparable:
~~template <class T, class U>~~
~~concept WeaklyEqualityComparableWith = see below;~~

template <class T>
concept EqualityComparable = see below;

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

// 22.4.4, StrictTotallyOrdered:
template <class T>
concept StrictTotallyOrdered = see below;

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

// 22.5, object concepts:
// 22.5.1, Movable:
```

```
  template <class T>
  concept Movable = see below;

  // 22.5.2, Copyable:
  template <class T>
  concept Copyable = see below;

  // 22.5.3, Semiregular:
  template <class T>
  concept Semiregular = see below;

  // 22.5.4, Regular:
  template <class T>
  concept Regular = see below;

  // 22.6, callable concepts:
  // 22.6.2, Invocable:
  template <class F, class... Args>
  concept Invocable = see below;

  // 22.6.3, RegularInvocable:
  template <class F, class... Args>
  concept RegularInvocable = see below;

  // 22.6.4, Predicate:
  template <class F, class... Args>
  concept Predicate = see below;

  // 22.6.5, Relation:
  template <class R, class T, class U>
  concept Relation = see below;

  // 22.6.6, StrictWeakOrder:
  template <class R, class T, class U>
  concept StrictWeakOrder = see below;
}
```

22.3 Core language concepts [concepts.lib.corelang]

22.3.1 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.

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

```
template <class T, class U>
concept Same = is_same_v<T, U>; // see below
```

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

2 ~~*Remarks:* For the purposes of constraint checking, `Same<T, U>` implies `Same<U, T>`~~ `Same<T, U>` subsumes `Same<U, T>` and vice versa.

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

```
template <class Derived, class Base>
concept DerivedFrom = is_base_of_v<Base, Derived> &&
  is_convertible_v<remove_cv_t<Derived>*, remove_cv_t<Base>*>;
  is_convertible_v<const volatile Derived*, const volatile Base*>; // see below
```

1 There need not be any subsumption relationship between `DerivedFrom<Derived, Base>` and either `is_base_of_v<Base, Derived>` or `is_convertible_v<remove_cv_t<Derived>*, remove_cv_t<Base>*>`.

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

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

```
template <class From, class To>
concept ConvertibleTo = is_convertible_v<From, To> && // 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 (20.5.5.15).

(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_v<From, To>`.

22.3.5 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 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 (22.1.1).

(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 (23.15.7.6). — *end note*]

22.3.6 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 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 (22.1.1).

(2.2) — `C(u())` equals `C(u())` if and only if `u()` is an equality preserving expression (22.1.1).

3 [*Note:* Users can customize the behavior of `Common` by specializing the `common_type` class template (23.15.7.6). — *end note*]

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

```
template <class T>
concept Integral = is_integral_v<T>; // see below
```

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

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

```
template <class T>
concept SignedIntegral = Integral<T> && is_signed_v<T>; // see below
```

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

2 [*Note:* `SignedIntegral<T>` may be satisfied even for types that are not signed integral types (6.7.1); for example, `char`. — *end note*]

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

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

1 [*Note:* `UnsignedIntegral<T>` may be satisfied even for types that are not unsigned integral types (6.7.1); for example, `char`. — *end note*]

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

```
template <class LHS, class RHS>
concept Assignable =
  is_lvalue_reference_v<LHS> && // see below
  CommonReference<const remove_reference_t<LHS>&, const remove_reference_t<RHS>&> &&
  requires(LHS lhs, RHS&& rhs) {
    { lhs = std::forward<RHS>(rhs) } -> Same<LHS>&&;
    lhs = std::forward<RHS>(rhs);
    requires Same<decltype(lhs = std::forward<RHS>(rhs)), LHS>;
  };
```

1 Let `lhs` be an lvalue that refers to an object `lcopy` such that `decltype((lhs))` is `LHS`, and `rhs` an expression such that `decltype((rhs))` is `RHS`. Let `rcopy` be a distinct object that is equal to `rhs`. `Assignable<LHS, RHS>` is satisfied only if

(1.1) — `addressof(lhs = rhs) == addressof(lcopy)`.

(1.2) — After evaluating `lhs = rhs`:

(1.2.1) — `lhs` is equal to `rcopy`, unless `rhs` is a non-const xvalue that refers to `lcopy`.

(1.2.2) — If `rhs` is a non-`const` xvalue, the resulting state of the object to which it refers is valid but unspecified (20.5.5.15).

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

2 There need not be any subsumption relationship between `Assignable<LHS, RHS>` and `is_lvalue_reference_v<LHS>`.

3 [*Note:* Assignment need not be a total function (20.4.1.3); 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*]

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

```
template <class T>
concept Swappable = requires(T& a, T& b) { std2::swap(a, b); };

template <class T, class U>
concept SwappableWith =
  CommonReference<const remove_reference_t<T>&, const remove_reference_t<U>&> &&
  requires(T&& t, U&& u) {
    std2::swap(std::forward<T>(t), std::forward<T>(t));
    std2::swap(std::forward<U>(u), std::forward<U>(u));
    std2::swap(std::forward<T>(t), std::forward<U>(u));
    std2::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 `std2::swap(t, u)` or `std2::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~~SwappableWith<T> U>
void value_swap(T&& t, U&& u) {
  std2::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~~Swappable T>
void lv_swap(T& t1, T& t2) {
  std2::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) {
    std2::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*]

22.3.12 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 Destructible = is_nothrow_destructible_v<T>; // see below
```

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

3 [*Note:* Unlike the `Destructible` ~~library concept in the C++ Standard~~requirements (Table 27), this concept forbids destructors that are ~~`noexcept(false)`~~potentially throwing, even if non-throwing. — *end note*]

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

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

```
template <class T, class... Args>
concept Constructible = Destructible<T> && is_constructible_v<T, Args...>; // see below
```

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

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

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

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

```
template <class T>
concept 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 (20.5.5.15); otherwise, it is unchanged.

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

```
template <class T>
concept 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`.

22.4 Comparison concepts [concepts.lib.compare]

22.4.1 General [concepts.lib.compare.general]

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

22.4.2 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 Boolean = Movable<remove_cvref_t<B>> && // (see 22.5.1)
  requires(const remove_reference_t<B>& b1,
           const remove_reference_t<B>& b2, const bool a) {
    { b1 } -> ConvertibleTo<bool>&&;
    requires ConvertibleTo<const remove_reference_t<B>&, bool>;
    { !b1 } -> ConvertibleTo<bool>&&;
    !b1; requires ConvertibleTo<decltype(!b1), bool>;
    { b1 && a } -> Same<bool>&&;
    b1 && a; requires Same<decltype(b1 && a), bool>;
    { b1 || a } -> Same<bool>&&;
    b1 || a; requires Same<decltype(b1 || a), bool>;
    { b1 && b2 } -> Same<bool>&&;
    b1 && b2; requires Same<decltype(b1 && b2), bool>;
    { a && b2 } -> Same<bool>&&;
    a && b2; requires Same<decltype(a && b2), bool>;
    { b1 || b2 } -> Same<bool>&&;
    b1 || b2; requires Same<decltype(b1 || b2), bool>;
    { a || b2 } -> Same<bool>&&;
    a || b2; requires Same<decltype(a || b2), bool>;
    { b1 == b2 } -> ConvertibleTo<bool>&&;
    b1 == b2; requires ConvertibleTo<decltype(b1 == b2), bool>;
    { b1 == a } -> ConvertibleTo<bool>&&;
    b1 == a; requires ConvertibleTo<decltype(b1 == a), bool>;
    { a == b2 } -> ConvertibleTo<bool>&&;
    a == b2; requires ConvertibleTo<decltype(a == b2), bool>;
    { b1 != b2 } -> ConvertibleTo<bool>&&;
    b1 != b2; requires ConvertibleTo<decltype(b1 != b2), bool>;
    { b1 != a } -> ConvertibleTo<bool>&&;
    b1 != a; requires ConvertibleTo<decltype(b1 != a), bool>;
    { a != b2 } -> ConvertibleTo<bool>&&;
    a != b2; requires ConvertibleTo<decltype(a != b2), bool>;
  };
```

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

(2.1) — bool(b1) == !bool(!b1).

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

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

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

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

3 [*Example:* The types bool, true_type (23.15.2), and bitset<*N*>::reference (23.9.2) are Boolean types. Pointers, smart pointers, and types with only explicit conversions to bool are not Boolean types. — *end example*]

22.4.3 Concept EqualityComparable [concepts.lib.compare.equalitycomparable]

```
template <class T, class U>
concept __WeaklyEqualityComparableWith = // exposition only
  requires(const remove_reference_t<T>& t,
           const remove_reference_t<U>& u) {
    { t == u } -> Boolean&&;
    t == u; requires Boolean<decltype(t == u)>;
    { t != u } -> Boolean&&;
```

```
    t != u; requires Boolean<decltype(t != u)>;
    { u == t } -> Boolean&&;
    u == t; requires Boolean<decltype(u == t)>;
    { u != t } -> Boolean&&;
    u != t; requires Boolean<decltype(u != t)>;
  };
```

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

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

(1.2) — bool(u == t) == bool(t == u).

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

(1.4) — bool(u != t) == bool(t != u).

```
template <class T>
concept EqualityComparable = __WeaklyEqualityComparableWith<T, T>;
```

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

(2.1) — bool(a == b) if and only if a is equal to b (22.1.1).

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 EqualityComparableWith =
  EqualityComparable<T> && EqualityComparable<U> &&
  CommonReference<const remove_reference_t<T>&, const remove_reference_t<U>&> &&
  EqualityComparable<common_reference_t<const remove_reference_t<T>&, const remove_reference_t<U>&>> &&
  __WeaklyEqualityComparableWith<T, U>;
```

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

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

EqualityComparableWith<T, U> is satisfied only if:

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

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

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

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 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; requires Boolean<decltype(t < u)>;
    { t > u } -> Boolean&&;
    t > u; requires Boolean<decltype(t > u)>;
    { t <= u } -> Boolean&&;
    t <= u; requires Boolean<decltype(t <= u)>;
    { t >= u } -> Boolean&&;
    t >= u; requires Boolean<decltype(t >= u)>;
    { u < t } -> Boolean&&;
    u < t; requires Boolean<decltype(u < t)>;
    { u > t } -> Boolean&&;
    u > t; requires Boolean<decltype(u > t)>;
    { u <= t } -> Boolean&&;
    u <= t; requires Boolean<decltype(u <= t)>;
    { u >= t } -> Boolean&&;
    u >= t; requires Boolean<decltype(u >= t)>;
  };
```

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))`.

22.5 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.

22.5.1 Concept `Movable` [concepts.lib.object.movable]

```
template <class T>
concept Movable = is_object_v<T> && MoveConstructible<T> && Assignable<T&, T> && Swappable<T>;
```

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

22.5.2 Concept `Copyable` [concepts.lib.object.copyable]

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

22.5.3 Concept `Semiregular` [concepts.lib.object.semiregular]

```
template <class T>
concept 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*]

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

```
template <class T>
concept 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*]

22.6 Callable concepts [concepts.lib.callable]

22.6.1 General [concepts.lib.callable.general]

1 The concepts in this section describe the requirements on function objects (23.14) and their arguments.

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

1 The `Invocable` concept specifies a relationship between a callable type (23.14.2) `F` and a set of argument types `Args...` which can be evaluated by the library function `invoke` (23.14.4).

```
template <class F, class... Args>
concept 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 (22.1.1), a function that generates random numbers may satisfy `Invocable`. — *end note*]

22.6.3 Concept RegularInvocable [concepts.lib.callable.regularinvocable]

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

1 The `invoke` function call expression shall be equality-preserving and shall not modify the function object or the arguments (22.1.1). [*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*]

22.6.4 Concept Predicate [concepts.lib.callable.predicate]

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

22.6.5 Concept Relation [concepts.lib.callable.relation]

```
template <class R, class T, class U>
concept 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)))`.

22.6.6 Concept StrictWeakOrder [concepts.lib.callable.strictweakorder]

```
template <class R, class T, class U>
concept 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*]

23 General utilities library [utilities]

23.2 Utility components [utility]

23.2.1 Header <utility> synopsis [utility.syn]

[Editor's note: Add new declarations at the end of the synopsis:]

```
  template<size_t I> inline constexpr in_place_index_t<I> in_place_index{};
}

namespace std2 { inline namespace v1 {
  // 23.6, swap customization point:
  inline constexpr unspecified swap = unspecified;
}}
```

[Editor's note: Add new subclause before [optional]:]

23.6 swap customization point [utility.swap2]

1 The name `std2::swap` denotes a customization point object (20.4.2.1.6). The ~~effect of the~~ expression `std2::swap(E1, E2)` for some subexpressions `E1` and `E2` is expression-equivalent (20.3.11) to:

(1.1) — `(void)swap(E1, E2)`[3], 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 declarations of `std::swap` or `std2::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)`~~ `(void)__std2_swap_ranges(std::begin(E1), std::end(E1), std::begin(E2))` if `E1` and `E2` are lvalues of array types (6.7.2) of equal extent and `std2::swap(*(E1), *(E2))` is a valid expression, except that ~~`noexcept(std2::swap(E1, E2))` is equal to `noexcept(std2::swap(*(E1),`~~ `E1` is evaluated only once. `__std2_swap_ranges` is the exposition-only function:

```
template <class T, class U>
  constexpr void __std2_swap_ranges(T* first1, T* last1, U* first2)
    noexcept(noexcept(std2::swap(*(E1), *(E2))))
{
  for (; first1 != last1; ++first, ++first2) {
    std2::swap(*first1, *first2);
  }
}
```

3) The name `swap` is used here unqualified.

(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. `std2::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(std2::swap(E1, E2))` is equal to `is_nothrow_move_constructible_v<T> && is_nothrow_move_assignable_v<T>`. If either `MoveConstructible` or `Assignable` is not satisfied, the program is ill-formed with no diagnostic required.

(1.4) — Otherwise, `std2::swap(E1, E2)` is ill-formed.

2 *Remark:* Whenever `std2::swap(E1, E2)` is a valid expression, it exchanges the values referenced by `E1` and `E2` and has type `void`.

23.14 Function objects [function.objects]

[Editor's note: Add a new declaration to the `<functional>` synopsis:]

23.14.1 Header `<functional>` synopsis [functional.syn]

```
[...]
template<> struct bit_xor<void>;
template<> struct bit_not<void>;

// 23.14.10, identity:
struct identity;

// 23.14.10, function template not_fn
template<class F> unspecified not_fn(F&& f);

[...]
```

[Editor's note: Add a new subclause before [func.not_fn]:]

23.14.10 Class identity [func.identity]

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

  using is_transparent = unspecified ;
};
```

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

1 ~~*Returns:*~~ *Effects:* Equivalent to: `return std::forward<T>(t);`

23.15 Metaprogramming and type traits [meta]

23.15.2 Header `<type_traits>` synopsis [meta.type.synop]

[Editor's note: Add new declarations to the `<type_traits>` synopsis:]

```
[...]
template <class... T> struct common_type;
template <class T, class U, template <class> class TQual, template <class> class UQual>
  struct basic_common_reference { };
template <class... T> struct common_reference;
template<class T> struct underlying_type;
[...]
template <class... T>
  using common_type_t = typename common_type<T...>::type;
template <class... T>
  using common_reference_t = typename common_reference<T...>::type;
template<class T>
  using underlying_type_t = typename underlying_type<T>::type;
[...]
```

23.15.7.6 Other transformations [meta.trans.other]

[Editor's note: Add new traits to Table 50]

Table 50 — Other transformations

Template	Comments
...	...
`template<class... T> struct common_type;`	Unless this trait is specialized (as specified in Note B, below), the member `type` shall be defined or omitted as specified in Note A, below. If it is omitted, there shall be no member `type`. Each type in the parameter pack `T` shall be complete, *cv* `void`, or an array of unknown bound.
`template <class, class, template <class> class, template <class> class> struct basic_common_reference;`	~~The primary template shall have no member typedef `type`.~~ A program may specialize this trait if at least one template parameter in the specialization depends on a user-defined type. In such a specialization, a member typedef `type` may be defined or omitted. If it is omitted, there shall be no member `type`. [*Note:* Such specializations may be used to influence the result of `common_reference`. — *end note*]
`template <class... T> struct common_reference;`	The member typedef `type` shall be defined or omitted as specified below. If it is omitted, there shall be no member `type`. Each type in the parameter pack `T` shall be complete or (possibly *cv*) `void`.
...	...

[Editor's note: Insert this new paragraph before paragraph 3:]

3 Let `CREF(A)` be `add_lvalue_reference_t<const remove_reference_t<A>>`. Let `XREF(A)` denote a unary template `T` such that ~~`T<remove_cvref_t<A>>` denotes the same type as `A`~~ `T<U>` denotes the same type as `U` with the addition of `A`'s cv and reference qualifiers, for a type `U` such that `is_same_v<U, remove_cvref_t<U>>` is `true`. 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:

(3.1) — If `A` and `B` are both lvalue reference types, `COMMON_REF(A, B)` is `COND_RES(COPYCV(X, Y) &, COPYCV(Y, X) &)` if that type exists and is a reference type.

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

(3.3) — 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_v<A, D>` is `true`, then `COMMON_REF(A, B)` is `D`.

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

(3.5) — Otherwise, `COMMON_REF(A, B)` is ~~`decay_t<COND_RES(CREF(A), CREF(B))>`~~ ill-formed.

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

[Editor's note: Modify the following "Note A" paragraph as follows:]

4 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:

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

(4.2) — If `sizeof...(T)` is one, let `T0` denote the sole type constituting the pack `T`. The member *typedef-name* `type` shall denote the same type, if any, as `common_type_t<T0, T0>`; otherwise there shall be no member `type`.

(4.3) — If `sizeof...(T)` is two, let the first and second types constituting `T` be denoted by `T1` and `T2`, respectively, and let `D1` and `D2` denote the same types as `decay_t<T1>` and `decay_t<T2>`, respectively.

(4.3.1) — If `is_same_v<T1, D1>` is `false` or `is_same_v<T2, D2>` is `false`, let `C` denote the same type, if any, as `common_type_t<D1, D2>`.

(4.3.2) — [*Note:* None of the following will apply if there is a specialization `common_type<D1, D2>`. — *end note*]

(4.3.3) — Otherwise, ~~let `C` denote the same type, if any, as~~ if

```
decay_t<decltype(false ? declval<D1>() : declval<D2>())>
```

~~[*Note:* This will not apply if there is a specialization common_type<D1, D2>. — *end note*]~~ denotes a valid type, let `C` denote its type.

(4.3.4) — Otherwise, let `C` denote the same type as `decay_t<COND_RES(CREF(A), CREF(B))>`, if any.

In either case, the member *typedef-name* `type` shall denote the same type, if any, as `C`. Otherwise, there shall be no member `type`.

(4.4) — If `sizeof...(T)` is greater than two, let `T1`, `T2`, and `R`, respectively, denote the first, second, and (pack of) remaining types constituting `T`. Let `C` denote the same type, if any, as `common_type_t<T1, T2>`. If there is such a type `C`, the member *typedef-name* `type` shall denote the same type, if any, as `common_type_t<C, R...>`. Otherwise, there shall be no member `type`.

[Editor's note: Add new paragraphs following the paragaph that begins "Note B":]

5 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:

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

(5.2) — Otherwise, if `sizeof...(T)` is one, let `T0` denote the sole type in the pack `T`. The member typedef `type` shall denote the same type as `T0`.

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

(5.3.1) — 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.

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

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

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

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

(5.4) — 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:

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

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

6 Notwithstanding the provisions of 23.15.2, and pursuant to 20.5.4.2.1, a program may specialize `basic_common_reference<T, U, TQual, UQual>` for types `T` and `U` such that `is_same_v<T, decay_t<T>>` and `is_same_v<U, decay_t<U>>` 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.

29 Numerics library [numerics]

29.6 Random number generation [rand]

[Editor's note: Relocate "Header /tcode<random> synopsis" [rand.synopsis] before 29.6.1 "Requirements" [rand.req]]

29.6.1 Header `<random>` synopsis [rand.synopsis]

[Editor's note: Modify the `<random>` synopsis as follows:]

```
#include <initializer_list>

namespace std {
  // 29.6.1.1, concept UniformRandomBitGenerator
  template <class G>
  concept UniformRandomNumberBitGenerator = see below;

  // 29.6.3.1, class template linear_congruential_engine
  template<class UIntType, UIntType a, UIntType c, UIntType m>
    class linear_congruential_engine;

  [...]
```

29.6.1.1 Uniform random bit generator requirements [rand.req.urng]

[Editor's note: Add new paragraphs after the existing content:]

```
template <class G>
concept UniformRandomNumberBitGenerator =
  Invocable<G&> && UnsignedIntegral<result_of_t<G&()>invoke_result_t<G&>> &&
  requires {
    { G::min() } -> Same<result_of_t<G&()>>&&;
    G::min(); requires Same<decltype(G::min()), invoke_result_t<G&>>;
    { G::max() } -> Same<result_of_t<G&()>>&&;
    G::max(); requires Same<decltype(G::max()), invoke_result_t<G&>>;
  };
```

4 Let `g` be an object of type `G`. `UniformRandom~~Number~~BitGenerator<G>` is satisfied only if

(4.1) — Both `G::min()` and `G::max()` are constant expressions (8.6).

(4.2) — `G::min() < G::max()`.

(4.3) — `G::min() <= g()`.

(4.4) — `g() <= G::max()`.

(4.5) — `g()` has amortized constant complexity.

Index

Index of library names