Foundational Concepts for the C++0x Standard Library (Revision 4)

Authors: Douglas Gregor, Indiana University
Mat Marcus, Adobe Systems, Inc.
Thomas Witt, Zephyr Associates, Inc.
Andrew Lumsdaine, Indiana University
Document number: N2737=08-0247
Revises document number: N2677=08-0187
Date: 2008-08-25
Project: Programming Language C++, Library Working Group
Reply-to: Douglas Gregor <dgregor@osl.iu.edu>

Introduction

This document proposes basic support for concepts in the C++0x Standard Library. It describes a new header `<concepts>` that contains concepts that require compiler support (such as `SameType` and `ObjectType`) and concepts that describe common type behaviors likely to be used in many templates, including those in the Standard Library (such as `CopyConstructible` and `EqualityComparable`).

Within the proposed wording, text that has been added will be presented in blue and underlined when possible. Text that has been removed will be presented in red,~~with strike-through when possible~~.

Purely editorial comments will be written in a separate, shaded box.

Changes from N2677

- Added the `Constructible` concept, which refines the syntactic `HasConstructor` concept and `NothrowDestructible`. In Sophia-Antipolis, LWG removed `Destructible` from `HasConstructor` to decouple the semantic coupling within `HasConstructor`, with the understanding that, if `Destructible` requirements cropped up in too many places, we would bring the `Destructible` requirement back. The concepts drafting group realized that (1) `Destructible` was used in many places, and all of those were actually uses of `NothrowDestructible`, and (2) that `HasConstructor` still should not imply any semantics, and thus should not refine from any destruction concept. Hence, the addition of the new `Constructible` concept that implies some semantics by refining `NothrowDestructible`, and will be used throughout most of the standard library.
- Added the `RvalueOf` concept, which is needed to constrain and use `std::move` within the standard library. This concept replaces the `HasStdMove` concept that was part of the iterator concepts proposal.
- In `MoveConstructible` and `MoveAssignable`, added a constructor and copy-assignment operator requirement, respectively, that accept an `RvalueOf<T>::type` on the right-hand side, to account for uses of `std::move`.
- Added some introductory text describing the naming conventions used for concepts (20.1).
- Added a new concept `EquivalenceRelation`.

— Added a new concept `HasVirtualDestructor`.

— In the `FreeStoreAllocatable` concept, default the implementations of operators `new[]` and `delete[]` to the use `new` and `delete`, respectively.

— Added the `ArithmeticType` and `FloatingPointType` concepts.

— Added the `underlying_type` associated type to the `EnumerationType` concept.

— Added the `IdentityOf` concept, which acts like a simple identity metafunction. It is used by `std::forward`.

— In 20.1.4, require that the concept maps in namespace `std` for pointer types for `HasLess`, `HasGreater`, `HasLessEqual`, and `HasGreaterEqual` ensure a total order. This is how concepts resolve issue 532 (already closed as NAD).

— Added the `FunctionType` concept.

Proposed Wording

Issues resolved by concepts

The following LWG are resolved by concepts. These issues should be resolved as NAD following the application of this proposal to the wording paper:

Issue 556. Is Compare a BinaryPredicate? With concepts, we've specified exactly what "convertible to bool" means for predicates, and all Compare objects are predicates (since they refine the Predicate concept). Short-circuiting is taken care of because the Predicate concept forces conversion to bool inside all constrained templates.

Issue 724. DefaultConstructible is not defined. The concepts proposal provides a `DefaultConstructible` concept.

Chapter 20 General utilities library [utilities]

2 The following clauses describe utility ~~and allocator~~ ~~requirements~~concepts, utility components, tuples, type traits templates, function objects, dynamic memory management utilities, and date/time utilities, as summarized in Table 30.

Table 30: General utilities library summary

Subclause	Header(s)
20.1 ~~Requirements~~Concepts	`<concepts>`
[utility] Utility components	`<utility>`
[tuple] Tuples	`<tuple>`
[meta] Type traits	`<type_traits>`
[function.objects] Function objects	`<functional>`
[memory] Memory	`<memory>` `<cstdlib>` `<cstring>`
[date.time] Date and time	`<ctime>`

Replace the section [utility.requirements] with the following section [utility.concepts]

20.1 Concepts [utility.concepts]

1 This subclause describes concepts that specify requirements on template arguments used throughout the C++ Standard Library. Concepts whose name is prefixed with `Has` provide detection of a specific syntax (e.g., `HasConstructor`), but do not imply the semantics of the corresponding operation. Concepts whose name has the `able` or `ible` suffix (e.g., `Constructible`) require both a specific syntax and semantics of the associated operations. These semantic concepts refine the corresponding syntax-detection concepts, for example, the `Constructible` concept refines the `HasConstructor` concept.

Header `<concepts>` synopsis

```
namespace std {
  // 20.1.1, support concepts:
  concept Returnable<typename T> { }
  concept PointeeType<typename T> { }
  concept MemberPointeeType<typename T> see below;
  concept ReferentType<typename T> { }
  concept VariableType<typename T> { }
```

```
concept ObjectType<typename T> see below;
concept FunctionType<typename T> see below;
concept ClassType<typename T> see below;
concept Class<typename T> see below;
concept Union<typename T> see below;
concept TrivialType<typename T> see below;
concept StandardLayoutType<typename T> see below;
concept LiteralType<typename T> see below;
concept ScalarType<typename T> see below;
concept ArithmeticType<typename T> see below;
concept NonTypeTemplateParameterType<typename T> see below;
concept IntegralConstantExpressionType<typename T> see below;
concept IntegralType<typename T> see below;
concept EnumerationType<typename T> see below;
concept FloatingPointType<typename T> see below;
concept SameType<typename T, typename U> {  }
concept DerivedFrom<typename Derived, typename Base> { }

// 20.1.2, type transformations:
auto concept IdentityOf<typename T> see below;
auto concept RvalueOf<typename T> see below;
template<typename T> concept_map RvalueOf<T&> see below;

// 20.1.3, true:
concept True<bool> { }
concept_map True<true> { }

// 20.1.4, operator concepts:
auto concept HasPlus<typename T, typename U> see below;
auto concept HasMinus<typename T, typename U> see below;
auto concept HasMultiply<typename T, typename U> see below;
auto concept HasDivide<typename T, typename U> see below;
auto concept HasModulus<typename T, typename U> see below;
auto concept HasUnaryPlus<typename T> see below;
auto concept HasNegate<typename T> see below;
auto concept HasLess<typename T, typename U> see below;
auto concept HasGreater<typename T, typename U> see below;
auto concept HasLessEqual<typename T, typename U> see below;
auto concept HasGreaterEqual<typename T, typename U> see below;
auto concept HasEqualTo<typename T, typename U> see below;
auto concept HasNotEqualTo<typename T, typename U> see below;
auto concept HasLogicalAnd<typename T, typename U> see below;
auto concept HasLogicalOr<typename T, typename U> see below;
auto concept HasLogicalNot<typename T> see below;
auto concept HasBitAnd<typename T, typename U> see below;
auto concept HasBitOr<typename T, typename U> see below;
auto concept HasBitXor<typename T, typename U> see below;
auto concept HasComplement<typename T> see below;
auto concept HasLeftShift<typename T, typename U> see below;
auto concept HasRightShift<typename T, typename U> see below;
```

```
auto concept HasDereference<typename T> see below;
auto concept HasAddressOf<typename T> see below;
auto concept Callable<typename F, typename... Args> see below;
auto concept HasAssign<typename T, typename U> see below;
auto concept HasPlusAssign<typename T, typename U> see below;
auto concept HasMinusAssign<typename T, typename U> see below;
auto concept HasMultiplyAssign<typename T, typename U> see below;
auto concept HasDivideAssign<typename T, typename U> see below;
auto concept HasModulusAssign<typename T, typename U> see below;
auto concept HasBitAndAssign<typename T, typename U> see below;
auto concept HasBitOrAssign<typename T, typename U> see below;
auto concept HasBitXorAssign<typename T, typename U> see below;
auto concept HasLeftShiftAssign<typename T, typename U> see below;
auto concept HasRightShiftAssign<typename T, typename U> see below;
auto concept HasPreincrement<typename T> see below;
auto concept HasPostincrement<typename T> see below;
auto concept HasPredecrement<typename T> see below;
auto concept HasPostdecrement<typename T> see below;
auto concept HasComma<typename T, typename U> see below;

// 20.1.5, predicates:
auto concept Predicate<typename F, typename... Args> see below;

// 20.1.6, comparisons:
auto concept LessThanComparable<typename T> see below;
auto concept EqualityComparable<typename T> see below;
concept TriviallyEqualityComparable<typename T> see below;
auto concept StrictWeakOrder<typename F, typename T> see below;
auto concept EquivalenceRelation<typename F, typename T> see below;

// 20.1.7, construction:
auto concept HasConstructor<typename T, typename... Args> see below;
auto concept Constructible<typename T, typename... Args> see below;
auto concept DefaultConstructible<typename T> see below;
concept TriviallyDefaultConstructible<typename T> see below;

// 20.1.8, destruction:
auto concept HasDestructor<typename T> see below;
auto concept HasVirtualDestructor<typename T> see below;
auto concept NothrowDestructible<typename T> see below;
concept TriviallyDestructible<typename T> see below;

// 20.1.9, copy and move:
auto concept MoveConstructible<typename T> see below;
auto concept CopyConstructible<typename T> see below;
concept TriviallyCopyConstructible<typename T> see below;
auto concept MoveAssignable<typename T> see below;
auto concept CopyAssignable<typename T> see below;
concept TriviallyCopyAssignable<typename T> see below;
auto concept HasSwap<typename T, typename U> see below;
```

```
    auto concept Swappable<typename T> see below;

    // 20.1.10, memory allocation:
    auto concept HasPlacementNew<typename T> see below;
    auto concept FreeStoreAllocatable<typename T> see below;

    // 20.1.11, regular types:
    auto concept Semiregular<typename T> see below;
    auto concept Regular<typename T> see below;

    // 20.1.12, convertibility:
    auto concept ExplicitlyConvertible<typename T, typename U> see below;
    auto concept Convertible<typename T, typename U> see below;


    // 20.1.13, arithmetic concepts:
    concept ArithmeticLike<typename T> see below;
    concept IntegralLike<typename T> see below;
    concept SignedIntegralLike<typename T> see below;
    concept UnsignedIntegralLike<typename T> see below;
    concept FloatingPointLike<typename T> see below;
  }
```

20.1.1 Support concepts [concept.support]

1 The concepts in [concept.support] provide the ability to state template requirements for C++ type classifications ([basic.types]) and type relationships that cannot be expressed directly with concepts ([concept]). Concept maps for these concepts are implicitly defined. A program shall not provide concept maps for any concept in [concept.support].

```
concept Returnable<typename T> { }
```

2 *Note:* Describes types that can be used as the return type of a function.

3 *Requires:* for every non-array type `T` that is *cv* `void` or that meets the requirement `MoveConstructible<T>` (20.1.9), the concept map `Returnable<T>` shall be implicitly defined in namespace `std`.

```
concept PointeeType<typename T> { }
```

4 *Note:* describes types to which a pointer can be created.

5 *Requires:* for every type `T` that is an object type, function type, or *cv* `void`, a concept map `PointeeType` shall be implicitly defined in namespace `std`.

```
concept MemberPointeeType<typename T> : PointeeType<T> { }
```

6 *Note:* describes types to which a pointer-to-member can be created.

7 *Requires:* for every type `T` that is an object type or function type, a concept map `MemberPointeeType` shall be implicitly defined in namespace `std`.

```
concept ReferentType<typename T> { }
```

8 *Note:* describes types to which a reference can be created, including reference types (since references to references can be formed during substitution of template arguments).

9 *Requires:* for every type `T` that is an object type, a function type, or a reference type, a concept map `ReferentType` shall be implicitly defined in namespace `std`.

```
concept VariableType<typename T> { }
```

10 *Note:* describes types that can be used to declare a variable.

11 *Requires:* for every type `T` that is an object type or reference type, a concept map `VariableType<T>` shall be implicitly defined in namespace `std`.

```
concept ObjectType<typename T> : VariableType<T>, MemberPointeeType<T>
```

12 *Note:* describes object types ([basic.types]), for which storage can be allocated.

13 *Requires:* for every type `T` that is an object type, a concept map `ObjectType<T>` shall be implicitly defined in namespace `std`.

```
concept FunctionType<typename T> : PointeeType<T> { }
```

14 *Note:* describes function types ([dcl.fct]).

15 *Requires:* for every type `T` that is a function type, a concept map `FunctionType<T>` shall be implicitly defined in namespace `std`.

```
concept ClassType<typename T> : ObjectType<T> { }
```

16 *Note:* describes class types (i.e., unions, classes, and structs).

17 *Requires:* for every type `T` that is a class type ([class]), a concept map `ClassType<T>` shall be implicitly defined in namespace `std`.

```
concept Class<typename T> : ClassType<T> { }
```

18 *Note:* describes classes and structs ([class]).

19 *Requires:* for every type `T` that is a class or struct, a concept map `Class<T>` shall be implicitly defined in namespace `std`.

```
concept Union<typename T> : ClassType<T> { }
```

20 *Note:* describes union types ([class.union]).

21 *Requires:* for every type `T` that is a union, a concept map `Union<T>` shall be implicitly defined in namespace `std`.

```
concept TrivialType<typename T> : ObjectType<T> { }
```

22 *Note:* describes trivial types ([basic.types]).

23 *Requires:* for every type `T` that is a trivial type, a concept map `TrivialType<T>` shall be implicitly defined in namespace `std`.

```
concept StandardLayoutType<typename T> : ObjectType<T> { }
```

24 *Note:* describes standard-layout types ([basic.types]).

25 *Requires:* for every type `T` that is a standard-layout type, a concept map `StandardLayoutType<T>` shall be implicitly defined in namespace `std`.

```
concept LiteralType<typename T> : ObjectType<T> { }
```

26 *Note:* describes literal types ([basic.types]).

27 *Requires:* for every type `T` that is a literal type, a concept map `LiteralType<T>` shall be implicitly defined in namespace `std`.

```
concept ScalarType<typename T>
  : TrivialType<T>, LiteralType<T>, StandardLayoutType<T> { }
```

28 *Note:* describes scalar types ([basic.types]).

29 *Requires:* for every type `T` that is a scalar type, a concept map `ScalarType<T>` shall be implicitly defined in namespace `std`.

```
concept ArithmeticType<typename T> : ScalarType<T> { }
```

30 *Note:* describes arithmetic types ([basic.fundamental]).

31 *Requires:* for every type `T` that is an arithmetic type, a concept map `ArithmeticType<T>` shall be implicitly defined in namespace `std`.

```
concept NonTypeTemplateParameterType<typename T> : VariableType<T> { }
```

32 *Note:* describes type that can be used as the type of a non-type template parameter ([temp.param]).

33 *Requires:* for every type `T` that can be the type of a non-type *template-parameter* ([temp.param]), a concept map `NonTypeTemplateParameterType<T>` shall be implicitly defined in namespace `std`.

```
concept IntegralConstantExpressionType<typename T>
  : ScalarType<T>, NonTypeTemplateParameterType<T> { }
```

34 *Note:* describes types that can be the type of an integral constant expression ([expr.const]).

35 *Requires:* for every type `T` that is an integral type or enumeration type, a concept map `IntegralConstantExpressionType<T>` shall be implicitly defined in namespace `std`.

```
concept IntegralType<typename T>
  : IntegralConstantExpressionType<T>, ArithmeticType<T> { }
```

36 *Note:* describes integral types ([basic.fundamental]).

37 *Requires:* for every type `T` that is an integral type, a concept map `IntegralType<T>` shall be implicitly defined in namespace `std`.

```
concept EnumerationType<typename T> : IntegralConstantExpressionType<T> {
  IntegralType underlying_type;
}
```

38 *Note:* describes enumeration types ([dcl.enum]). underlying_type is the underlying type of the enumeration type.

39 *Requires:* for every type T that is an enumeration type, a concept map EnumerationType<T> shall be implicitly defined in namespace std.

```
concept FloatingPointType<typename T> : ArithmeticType<T> { }
```

40 *Note:* describes floating point types ([basic.fundamental]).

41 *Requires:* for every type T that is a floating point type, a concept map FloatingPointType<T> shall be implicitly defined in namespace std.

```
concept SameType<typename T, typename U> { }
```

42 *Note:* describes a same-type requirement ([temp.req]).

```
concept DerivedFrom<typename Derived, typename Base> { }
```

43 *Requires:* for every pair of class types (T, U), such that T is either the same as or publicly and unambiguously derived from U, a concept map DerivedFrom<T, U> shall be implicitly defined in namespace std.

20.1.2 Type transformations [concept.transform]

```
auto concept IdentityOf<typename T> {
  typename type = T;
  requires SameType<type, T>;
}
```

1 *Note:* concept form of the identity type metafunction ([forward]).

```
auto concept RvalueOf<typename T> {
  typename type = T&&;
  requires Convertible<T&, type> && Convertible<T&&, type>;
}
```

2 *Note:* describes the rvalue reference type for an arbitrary type T.

```
template<typename T> concept_map RvalueOf<T&> {
  typedef T&& type;
}
```

3 *Note:* provides the appropriate rvalue reference type for the rvalue an lvalue reference type. [*Note:*this concept map is required to circumvent reference collapsing for lvalue references. *— end note*]

20.1.3 True [concept.true]

```
concept True<bool> { }
concept_map True<true> { }
```

1 *Note:* used to express the requirement that a particular integral constant expression evaluate true.

2 *Requires:* a program shall not provide a concept map for the `True` concept.

20.1.4 Operator concepts [concept.operator]

```
auto concept HasPlus<typename T, typename U> {
  typename result_type;
  result_type operator+(const T&, const U&);
}
```

1 *Note:* describes types with a binary `operator+`.

```
auto concept HasMinus<typename T, typename U> {
  typename result_type;
  result_type operator-(const T&, const U&);
}
```

2 *Note:* describes types with a binary `operator-`.

```
auto concept HasMultiply<typename T, typename U> {
  typename result_type;
  result_type operator*(const T&, const U&);
}
```

3 *Note:* describes types with a binary `operator*`.

```
auto concept HasDivide<typename T, typename U> {
  typename result_type;
  result_type operator/(const T&, const U&);
}
```

4 *Note:* describes types with an `operator/`.

```
auto concept HasModulus<typename T, typename U> {
  typename result_type;
  result_type operator%(const T&, const U&);
}
```

5 *Note:* describes types with an `operator%`.

```
auto concept HasUnaryPlus<typename T> {
  typename result_type;
  result_type operator+(const T&);
}
```

6 *Note:* describes types with a unary `operator+`.

```
auto concept HasNegate<typename T> {
  typename result_type;
  result_type operator-(const T&);
}
```

7 *Note:* describes types with a unary `operator-`.

```
auto concept HasLess<typename T, typename U> {
  bool operator<(const T& a, const U& b);
}
```

8 *Note:* describes types with an `operator<`.

```
auto concept HasGreater<typename T, typename U> {
  bool operator>(const T& a, const U& b);
}
```

9 *Note:* describes types with an `operator>`.

```
auto concept HasLessEqual<typename T, typename U> {
  bool operator<=(const T& a, const U& b);
}
```

10 *Note:* describes types with an `operator<=`.

```
auto concept HasGreaterEqual<typename T, typename U> {
  bool operator>=(const T& a, const U& b);
}
```

11 *Note:* describes types with an `operator>=`.

12 For the concepts `HasLess`, `HasGreater`, `HasLessEqual`, and `HasGreaterEqual`, the concept maps in namespace `std` for any pointer type yield a total order, even if the built-in operators `<`, `>`, `<=`, `>=` do not.

```
auto concept HasEqualTo<typename T, typename U> {
  bool operator==(const T& a, const U& b);
}
```

13 *Note:* describes types with an `operator==`.

```
auto concept HasNotEqualTo<typename T, typename U> {
  bool operator!=(const T& a, const U& b);
}
```

14 *Note:* describes types with an `operator!=`.

```
auto concept HasLogicalAnd<typename T, typename U> {
  bool operator&&(const T&, const U&);
}
```

15 *Note:* describes types with a logical conjunction operator.

```
auto concept HasLogicalOr<typename T, typename U> {
  bool operator||(const T&, const U&);
}
```

16 *Note:* describes types with a logical disjunction operator.

```
auto concept HasLogicalNot<typename T> {
```

```
  bool operator!(const T&);
}
```

17 *Note:* describes types with a logical negation operator.

```
auto concept HasBitAnd<typename T, typename U> {
  typename result_type;
  result_type operator&(const T&, const U&);
}
```

18 *Note:* describes types with a binary `operator&`.

```
auto concept HasBitOr<typename T, typename U> {
  typename result_type;
  result_type operator|(const T&, const U&);
}
```

19 *Note:* describes types with an `operator|`.

```
auto concept HasBitXor<typename T, typename U> {
  typename result_type;
  result_type operator^(const T&, const U&);
}
```

20 *Note:* describes types with an `operator^`.

```
auto concept HasComplement<typename T> {
  typename result_type;
  result_type operator~(const T&);
}
```

21 *Note:* describes types with an `operator~`.

```
auto concept HasLeftShift<typename T, typename U> {
  typename result_type;
  result_type operator<<(const T&, const U&);
}
```

22 *Note:* describes types with an `operator<<`.

```
auto concept HasRightShift<typename T, typename U> {
  typename result_type;
  result_type operator>>(const T&, const U&);
}
```

23 *Note:* describes types with an `operator>>`.

```
auto concept HasDereference<typename T> {
  typename result_type;
  result_type operator*(const T&);
}
```

24 *Note:* describes types with a dereferencing `operator*`.

```
auto concept HasAddressOf<typename T> {
  typename result_type;
  result_type operator&(T&);
}
```

25 *Note:* describes types with an address-of `operator&`.

```
auto concept Callable<typename F, typename... Args> {
  typename result_type;
  result_type operator()(F&&, Args...);
}
```

26 *Note:* describes function object types callable given arguments of types `Args...`.

```
auto concept HasAssign<typename T, typename U> {
  typename result_type;
  result_type T::operator=(U);
}
```

27 *Note:* describes types with an assignment operator.

```
auto concept HasPlusAssign<typename T, typename U> {
  typename result_type;
  result_type operator+=(T&, U);
}
```

28 *Note:* describes types with an `operator+=`.

```
auto concept HasMinusAssign<typename T, typename U> {
  typename result_type;
  result_type operator-=(T&, U);
}
```

29 *Note:* describes types with an `operator-=`.

```
auto concept HasMultiplyAssign<typename T, typename U> {
  typename result_type;
  result_type operator*=(T&, U);
}
```

30 *Note:* describes types with an `operator*=`.

```
auto concept HasDivideAssign<typename T, typename U> {
  typename result_type;
  result_type operator/=(T&, U);
}
```

31 *Note:* describes types with an `operator/=`.

```
auto concept HasModulusAssign<typename T, typename U> {
  typename result_type;
  result_type operator%=(T&, U);
}
```

32 *Note:* describes types with an operator%=.

```
auto concept HasBitAndAssign<typename T, typename U> {
  typename result_type;
  result_type operator&=(T&, U);
}
```

33 *Note:* describes types with an operator&=.

```
auto concept HasBitOrAssign<typename T, typename U> {
  typename result_type;
  result_type operator|=(T&, U);
}
```

34 *Note:* describes types with an operator|=.

```
auto concept HasBitXorAssign<typename T, typename U> {
  typename result_type;
  result_type operator^=(T&, U);
}
```

35 *Note:* describes types with an operator^=.

```
auto concept HasLeftShiftAssign<typename T, typename U> {
  typename result_type;
  result_type operator<<=(T&, U);
}
```

36 *Note:* describes types with an operator<<=.

```
auto concept HasRightShiftAssign<typename T, typename U> {
  typename result_type;
  result_type operator>>=(T&, U);
}
```

37 *Note:* describes types with an operator>>=.

```
auto concept HasPreincrement<typename T> {
  typename result_type;
  result_type operator++(T&);
}
```

38 *Note:* describes types with a pre-increment operator.

```
auto concept HasPostincrement<typename T> {
  typename result_type;
  result_type operator++(T&, int);
}
```

39 *Note:* describes types with a post-increment operator.

```
auto concept HasPredecrement<typename T> {
  typename result_type;
```

```
  result_type operator--(T&);
}
```

40 *Note:* describes types with a pre-decrement operator.

```
auto concept HasPostdecrement<typename T> {
  typename result_type;
  result_type operator--(T&, int);
}
```

41 *Note:* describes types with a post-decrement operator.

```
auto concept HasComma<typename T, typename U> {
  typename result_type
  result_type operator,(const T&, const U&);
}
```

42 *Note:* describes types with a comma operator.

20.1.5 Predicates [concept.predicate]

```
auto concept Predicate<typename F, typename... Args> : Callable<F, const Args&...> {
  requires Convertible<result_type, bool>;
}
```

1 *Note:* describes function objects callable with some set of arguments, the result of which can be used in a context that requires a `bool`.

2 *Requires:* predicate function objects shall not apply any non-constant function through the predicate arguments.

20.1.6 Comparisons [concept.comparison]

```
auto concept LessThanComparable<typename T> : HasLess<T, T> {
  bool operator>(const T& a, const T& b) { return b < a; }
  bool operator<=(const T& a, const T& b) { return !(b < a); }
  bool operator>=(const T& a, const T& b) { return !(a < b); }

  axiom Consistency(T a, T b) {
    (a > b) == (b < a);
    (a <= b) == !(b < a);
    (a >= b) == !(a < b);
  }

  axiom Irreflexivity(T a) { (a < a) == false; }

  axiom Antisymmetry(T a, T b) {
    if (a < b)
      (b < a) == false;
  }
```

```
  axiom Transitivity(T a, T b, T c) {
    if (a < b && b < c)
      (a < c) == true;
  }

  axiom TransitivityOfEquivalence(T a, T b, T c) {
    if (!(a < b) && !(b < a) && !(b < c) && !(c < b))
      (!(a < c) && !(c < a)) == true;
  }
}
```

1 *Note:* describes types whose values can be ordered, where `operator<` is a strict weak ordering relation ([alg.sorting]).

```
auto concept EqualityComparable<typename T> : HasEqualTo<T, T> {
  bool operator!=(const T& a, const T& b) { return !(a == b); }

  axiom Consistency(T a, T b) {
    (a == b) == !(a != b);
  }

  axiom Reflexivity(T a) { a == a; }

  axiom Symmetry(T a, T b) {
    if (a == b)
      b == a;
  }

  axiom Transitivity(T a, T b, T c) {
    if (a == b && b == c)
      a == c;
  }
}
```

2 *Note:* describes types whose values can be compared for equality with `operator==`, which is an equivalence relation.

```
concept TriviallyEqualityComparable<typename T> : EqualityComparable<T> { }
```

3 *Note:* describes types whose equality comparison operators (`==`, `!=`) can be implemented via a bitwise equality comparison, as with `memcmp`. [*Note:* such types should not have padding, i.e. the size of the type is the sum of the sizes of its elements. If padding exists, the comparison may provide false negatives, but never false positives. *— end note*]

4 *Requires:* for every integral type `T` and pointer type, a concept map `TriviallyEqualityComparable<T>` shall be defined in namespace `std`.

```
auto concept StrictWeakOrder<typename F, typename T> : Predicate<F, T, T> {

  axiom Irreflexivity(F f, T a) { f(a, a) == false; }

  axiom Antisymmetry(F f, T a, T b) {
```

```
    if (f(a, b))
      f(b, a) == false;
  }

  axiom Transitivity(F f, T a, T b, T c) {
    if (f(a, b) && f(b, c))
      f(a, c) == true;
  }

  axiom TransitivityOfEquivalence(F f, T a, T b, T c) {
    if (!f(a, b) && !f(b, a) && !f(b, c) && !f(c, b))
      (!f(a, c) && !f(c, a)) == true;
  }
}
```

5 *Note:* describes a strict weak ordering relation ([alg.sorting]), F, on a type T.

```
auto concept EquivalenceRelation<typename F, typename T> : Predicate<F, T, T> {
  axiom Reflexivity(F f, T a) { f(a, a) == true; }

  axiom Symmetry(F f, T a, T b) {
    if (f(a, b))
      f(b, a) == true;
  }

  axiom Transitivity(F f, T a, T b, T c) {
    if (f(a, b) && f(b, c))
      f(a, c) == true;
  }
}
```

6 *Note:* describes an equivalence relation, F, on a type T.

20.1.7 Construction [concept.construct]

```
auto concept HasConstructor<typename T, typename... Args> {
  T::T(Args...);
}
```

1 *Note:* describes types that can be constructed from a given set of arguments.

```
auto concept Constructible<typename T, typename... Args>
  : HasConstructor<T, Args...>, NothrowDestructible<T> { }
}
```

2 *Note:* describes types that can be constructed from a given set of arguments that also have a no-throw destructor.

```
auto concept DefaultConstructible<typename T> : Constructible<T> { }
```

3 *Note:* describes types for which an object can be constructed without initializing the object to any particular value.

```
concept TriviallyDefaultConstructible<typename T> : DefaultConstructible<T> { }
```

4 *Note:* describes types whose default constructor is trivial.

5 *Requires:* for every type `T` that is a trivial type ([basic.types]) or a class type with a trivial default constructor ([class.ctor]), a concept map `TriviallyDefaultConstructible<T>` shall be implicitly defined in namespace `std`.

20.1.8 Destruction [concept.destruct]

```
auto concept HasDestructor<typename T> : VariableType<T> {
  T::~T();
}
```

1 *Note:* describes types that can be destroyed. These are scalar types, references, and class types with a public non-deleted destructor.

```
concept HasVirtualDestructor<typename T> : HasDestructor<T> { }
```

2 *Note:*describes types with a virtual destructor.

3 *Requires:*for every class type `T` that has a virtual destructor, a concept map `HasVirtualDestructor<T>` shall be implicitly defined in namespace `std`.

```
auto concept NothrowDestructible<typename T> : HasDestructor<T> { }
```

`T::~T()` *// inherited from HasDestructor<T>*

4 *Requires:* no exception is propagated.

```
concept TriviallyDestructible<typename T> : NothrowDestructible<T> { }
```

5 *Note:* describes types whose destructors do not need to be executed when the object is destroyed.

6 *Requires:* for every type `T` that is a trivial type ([basic.types]), reference, or class type with a trivial destructor ([class.dtor]), a concept map `TriviallyDestructible<T>` shall be implicitly defined in namespace `std`.

20.1.9 Copy and move [concept.copymove]

```
auto concept MoveConstructible<typename T> : Constructible<T, T&&> {
  requires RvalueOf<T> && Constructible<T, RvalueOf<T>::type>;
}
```

1 *Note:* describes types that can move-construct an object from a value of the same type, possibly altering that value.

`T::T(T&& rv);` *// note: inherited from HasConstructor<T, T&&>*

2 *Postcondition:* the constructed `T` object is equivalent to the value of `rv` before the construction. [*Note:* there is no requirement on the value of `rv` after the construction. — *end note*]

```
auto concept CopyConstructible<typename T> : MoveConstructible<T>, Constructible<T, const T&> {
  axiom CopyPreservation(T x) {
    T(x) == x;
  }
}
```

3 *Note:* describes types with a public copy constructor.

```
concept TriviallyCopyConstructible<typename T> : CopyConstructible<T> { }
```

4 *Note:* describes types whose copy constructor is equivalent to `memcpy`.

5 *Requires:* for every type `T` that is a trivial type ([basic.types]), a reference, or a class type with a trivial copy constructor ([class.copy]), a concept map `TriviallyCopyConstructible<T>` shall be implicitly defined in namespace `std`.

```
auto concept MoveAssignable<typename T> : HasAssign<T, T&&> {
  requires RvalueOf<T> && HasAssign<T, RvalueOf<T>::type>;
}
```

6 *Note:* describes types with the ability to assign to an object from an rvalue, potentially altering the rvalue.

`result_type T::operator=(T&& rv);` *// inherited from HasAssign<T, T&&>*

7 *Postconditions:* the constructed `T` object is equivalent to the value of `rv` before the assignment. [*Note:* there is no requirement on the value of `rv` after the assignment. *— end note*]

```
auto concept CopyAssignable<typename T> : HasAssign<T, const T&>, MoveAssignable<T> {
  axiom CopyPreservation(T& x, T y) {
    (x = y, x) == y;
  }
}
```

8 *Note:* describes types with the ability to assign to an object.

The CopyAssignable requirements in N2461 specify that `operator=` must return a `T&`. This is too strong a requirement for most of the uses of `CopyAssignable`, so we have weakened `CopyAssignable` to not require anything of its return type. When we need a `T&`, we'll add that as an explicit requirement. See, e.g., the `IntegralLike` concept.

```
concept TriviallyCopyAssignable<typename T> : CopyAssignable<T> { }
```

9 *Note:* describes types whose copy-assignment operator is equivalent to `memcpy`.

10 *Requires:* for every type `T` that is a trivial type ([basic.types]) or a class type with a trivial copy assignment operator ([class.copy]), a concept map `TriviallyCopyAssignable<T>` shall be implicitly defined in namespace `std`.

```
auto concept HasSwap<typename T, typename U> {
  void swap(T, U);
}
```

11 *Note:* describes types that have a swap operation.

```
auto concept Swappable<typename T> : HasSwap<T&, T&> { }
```

12 *Note:* describes types for which two values of that type can be swapped.

```
void swap(T& t, T& u); // inherited from HasSwap<T, T>
```

13 *Postconditions:* `t` has the value originally held by `u`, and `u` has the value originally held by `t`.

20.1.10 Memory allocation [concept.memory]

```
auto concept HasPlacementNew<typename T> {
  void* T::operator new(size_t size, void*);
}
```

1 *Note:* Describes types that have a placement new.

```
auto concept FreeStoreAllocatable<typename T> {
  void* T::operator new(size_t size);
  void T::operator delete(void*);

  void* T::operator new[](size_t size) {
    return T::operator new(size);
  }

  void T::operator delete[](void* ptr) {
    T::operator delete(ptr);
  }

  void* T::operator new(size_t size, const nothrow_t&) {
    try {
      return T::operator new(size);
    } catch(...) {
      return 0;
    }
  }

  void* T::operator new[](size_t size, const nothrow_t&) {
    try {
      return T::operator new[](size);
    } catch(...) {
      return 0;
    }
  }

  void T::operator delete(void* ptr, const nothrow_t&) {
    T::operator delete(ptr);
  }

  void T::operator delete[](void* ptr, const nothrow_t&) {
    T::operator delete[](ptr);
  }
}
```

2 *Note:* describes types for which objects and arrays of objects can be allocated on or freed from the free store with `new` and `delete`.

20.1.11 Regular types [concept.regular]

```
auto concept Semiregular<typename T>
  : CopyConstructible<T>, CopyAssignable<T>, FreeStoreAllocatable<T> {
  requires SameType<CopyAssignable<T>::result_type, T&>;
}
```

1 *Note:* collects several common requirements supported by most types.

```
auto concept Regular<typename T>
  : Semiregular<T>, DefaultConstructible<T>, EqualityComparable<T> { }
```

2 *Note:* describes semi-regular types that are default constructible and have equality comparison operators.

20.1.12 Convertibility [concept.convertible]

```
auto concept ExplicitlyConvertible<typename T, typename U> : VariableType<T> {
  explicit operator U(const T&);
}
```

1 *Note:* describes types with a conversion (explicit or implicit) from `T` to `U`.

```
auto concept Convertible<typename T, typename U> : ExplicitlyConvertible<T, U> {
  operator U(const T&);
}
```

2 *Note:* describes types with an implicit conversion from `T` to `U`.

20.1.13 Arithmetic concepts [concept.arithmetic]

```
concept ArithmeticLike<typename T>
  : Regular<T>, HasUnaryPlus<T>, HasNegate<T>,
    HasPlus<T, T>, HasMinus<T, T>, HasMultiply<T, T>, HasDivide<T, T>,
    HasLess<T, T>, HasGreater<T, T>, HasLessEqual<T, T>, HasGreaterEqual<T, T>,
    HasPreincrement<T>, HasPostincrement<T>, HasPredecrement<T>, HasPostdecrement<T>,
    HasPlusAssign<T, const T&>, HasMinusAssign<T, const T&>,
    HasMultiplyAssign<T, const T&>, HasDivideAssign<T, const T&> {
  T::T(intmax_t);
  T::T(uintmax_t);
  T::T(long double);

  requires Convertible<HasUnaryPlus<T>::result_type, T>
        && Convertible<HasNegate<T>::result_type, T>
        && Convertible<HasPlus<T, T>::result_type, T>
        && Convertible<HasMinus<T, T>::result_type, T>
```

```
        && Convertible<HasMultiply<T, T>::result_type, T>
        && Convertible<HasDivide<T, T>::result_type, T>
        && SameType<HasPreincrement<T>::result_type, T&>
        && SameType<HasPostincrement<T>::result_type, T>
        && SameType<HasPredecrement<T>::result_type, T&>
        && SameType<HasPostdecrement<T>::result_type, T>
        && SameType<HasPlusAssign<T, const T&>::result_type, T&>
        && SameType<HasMinusAssign<T, const T&>::result_type, T&>
        && SameType<HasMultiplyAssign<T, const T&>::result_type, T&>
        && SameType<HasDivideAssign<T, const T&>::result_type, T&>;
}
```

1 *Note:* describes types that provide all of the operations available on arithmetic types ([basic.fundamental]).

```
concept IntegralLike<typename T>
  : ArithmeticLike<T>, LessThanComparable<T>,
    HasComplement<T>, HasModulus<T, T>, HasBitAnd<T, T>, HasBitXor<T, T>, HasBitOr<T, T>,
    HasLeftShift<T, T>, HasRightShift<T, T>
    HasModulusAssign<T, const T&>, HasLeftShiftAssign<T, const T&>, HasRightShiftAssign<T, const T&>
    HasBitAndAssign<T, const T&>, HasBitXorAssign<T, const T&>, HasBitOrAssign<T, const T&> {
  requires Convertible<HasComplement<T>::result_type, T>
        && Convertible<HasModulus<T, T>::result_type, T>
        && Convertible<HasBitAnd<T, T>::result_type, T>
        && Convertible<HasBitXor<T, T>::result_type, T>
        && Convertible<HasBitOr<T, T>::result_type, T>
        && Convertible<HasLeftShift<T, T>::result_type, T>
        && Convertible<HasRightShift<T, T>::result_type, T>
        && SameType<HasModulusAssign<T, const T&>::result_type, T&>
        && SameType<HasLeftShiftAssign<T, const T&>::result_type, T&>
        && SameType<HasRightShiftAssign<T, const T&>::result_type, T&>
        && SameType<HasBitAndAssign<T, const T&>::result_type, T&>
        && SameType<HasBitXorAssign<T, const T&>::result_type, T&>
        && SameType<HasBitOrAssign<T, const T&>::result_type, T&>;
}
```

2 *Note:* describes types that provide all of the operations available on integral types.

```
concept SignedIntegralLike<typename T> : IntegralLike<T> { }
```

3 *Note:* describes types that provide all of the operations available on signed integral types.

4 *Requires:* for every signed integral type `T` ([basic.fundamental]), including signed extended integral types, an empty concept map `SignedIntegralLike<T>` shall be defined in namespace `std`.

```
concept UnsignedIntegralLike<typename T> : IntegralLike<T> { }
```

5 *Note:* describes types that provide all of the operations available on unsigned integral types.

6 *Requires:* for every unsigned integral type `T` ([basic.fundamental]), including unsigned extended integral types, an empty concept map `UnsignedIntegralLike<T>` shall be defined in namespace `std`.

```
concept FloatingPointLike<typename T> : ArithmeticLike<T> { }
```

7 *Note:* describes floating-point types.

8 *Requires:* for every floating point type `T` ([basic.fundamental]), an empty concept map `FloatingPointLike<T>` shall be defined in namespace `std`.

Acknowledgments

Daniel Krügler made many valuable suggestions that helped improve this document.