Core Concepts for the C++0x Standard Library (Revision 1)

Douglas Gregor and Andrew Lumsdaine
Open Systems Laboratory
Indiana University
Bloomington, IN 47405
{dgregor, lums}@osl.iu.edu

Document number: N2572=08-0082
Revises document number: N2502=08-0012
Date: 2008-03-16
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 N2502

— Applied the proposed resolutions for concept issues 1, 3, 4, 6, 9, 10, 12, 14, and 17.

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 components	`<utility>`
?? Tuples	`<tuple>`
?? Type traits	`<type_traits>`
?? Function objects	`<functional>`

?? Memory	`<memory>`
	`<cstdlib>`
	`<cstring>`

?? Date and time	`<ctime>`

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

20.1 Concepts [utility.concepts]

1 The `<concepts>` header describes requirements on template arguments used throughout the C++ Standard Library.

Header `<concepts>` synopsis

```
namespace std {
  // 20.1.1, support concepts:
  concept Returnable<typename T> { }
  concept PointeeType<typename T> { }
  concept ReferentType<typename T> see below;
  concept VariableType<typename T> { }
  concept ObjectType<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 NonTypeTemplateParameterType<typename T> see below;
concept IntegralConstantExpressionType<typename T> see below;
concept IntegralType<typename T> see below;
concept EnumerationType<typename T> see below;
concept SameType<typename T, typename U> {  }
concept DerivedFrom<typename Derived, typename Base> { }

// 20.1.2, comparisons:
auto concept LessThanComparable<typename T> see below;
auto concept EqualityComparable<typename T> see below;
auto concept TriviallyEqualityComparable<typename T> see below;

// 20.1.3, destruction:
auto concept Destructible<typename T> see below;
concept TriviallyDestructible<typename T> see below;

// 20.1.4, construction:
auto concept HasConstructor<typename T, typename... Args> see below;
auto concept DefaultConstructible<typename T> see below;
concept TriviallyDefaultConstructible<typename T> see below;

// 20.1.5, 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, typename U = T> see below;
auto concept CopyAssignable<typename T, typename U = T> see below;
concept TriviallyCopyAssignable<typename T> see below;
auto concept Swappable<typename T> see below;

// 20.1.6, memory allocation:
auto concept HeapAllocatable<typename T> see below;

// 20.1.7, regular types:
auto concept Semiregular<typename T> see below;
auto concept Regular<typename T> see below;

// 20.1.8, convertibility:
auto concept ExplicitlyConvertible<typename T, typename U> see below;
auto concept Convertible<typename T, typename U> see below;

// 20.1.9, true:
concept True<bool> { }
concept_map True<true> { }

// 20.1.10, operator concepts:
auto concept HasPlus<typename T, typename U = T> see below;
auto concept HasMinus<typename T, typename U = T> see below;
auto concept HasMultiply<typename T, typename U = T> see below;
```

```
  auto concept HasDivide<typename T, typename U = T> see below;
  auto concept HasModulus<typename T, typename U = T> see below;
  auto concept HasUnaryPlus<typename T> see below;
  auto concept HasNegate<typename T> see below;
  auto concept HasLess<typename T, typename U = T> see below;
  auto concept HasEqualTo<typename T, typename U = T> see below;
  auto concept HasLogicalAnd<typename T, typename U = T> see below;
  auto concept HasLogicalOr<typename T, typename U = T> see below;
  auto concept HasLogicalNot<typename T> see below;
  auto concept HasBitAnd<typename T, typename U = T> see below;
  auto concept HasBitOr<typename T, typename U = T> see below;
  auto concept HasBitXor<typename T, typename U = T> see below;
  auto concept HasComplement<typename T> see below;
  auto concept HasLeftShift<typename T, typename U = T> see below;
  auto concept HasRightShift<typename T, typename U = T> see below;
  auto concept Dereferenceable<typename T> see below;
  auto concept Addressable<typename T> see below;
  auto concept Callable<typename F, typename... Args> see below;

  // 20.1.11, 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;

  // 20.1.12, predicates:
  auto concept Predicate<typename F, typename... Args> see below;

  // 20.1.13, allocators:
  concept Allocator<typename X> see below;
  concept AllocatorGenerator<typename X> see below;
  template<Allocator X> concept_map AllocatorGenerator<X> 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 type `T` that is *cv* `void` or that meets the requirement `MoveConstructible<T>` (20.1.5), 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 ReferentType<typename T> : PointeeType<T> { }
```

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

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

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

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

9 *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> { }
```

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

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

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

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

13 *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<Class> { }
```

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

15 *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> { }
```

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

17 *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> { }
```

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

19 *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> { }
```

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

21 *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> { }
```

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

23 *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> { }
```

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

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

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

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

27 *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> { }
```

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

29 *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> { }
```

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

31 *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> { }
```

32 *Note:* describes enumeration types ([dcl.enum]).

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

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

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

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

35 *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 Comparisons [concept.comparison]

1 *Note:* describes types with an operator <.

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

2 *Note:* describes types whose values can be ordered, where `operator<` is a strict weak ordering relation (**??**).

```
auto concept EqualityComparable<typename T> : EqualTo<T> {
  bool operator!=(T const& a, U const& 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;
  }
}
```

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

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

4 *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*]

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

20.1.3 Destruction [concept.destruct]

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

1 *Note:* describes types that can be destroyed, including scalar types, references, and class types with a public destructor.

2 *Requires:* following destruction of an object, ~~All~~all resources owned by the object are reclaimed.

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

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

4 *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`.

20.1.4 Construction [concept.construct]

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

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

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

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

20.1.5 Copy and move [concept.copymove]

```
auto concept MoveConstructible<typename T> : HasConstructor<T, T&&> { }
```

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>, HasConstructor<T, const T&> {
  requires EqualityComparable<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, typename U = T> {
  typename result_type;
  result_type T::operator=(U&&);
}
```

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

```
result_type T::operator=(U&& rv);
```

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, typename U = T> : MoveAssignable<T, U> {
  typename result_type;
  result_type T::operator=(const U&);

  requires EqualityComparable<T, U>
  axiom CopyPreservation(T& x, U 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 `Integral` 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 Swappable<typename T> {
  void swap(T&, T&);
}
```

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

```
void swap(T& t, T& u);
```

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

20.1.6 Memory allocation [concept.memory]

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

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

20.1.7 Regular types [concept.regular]

```
auto concept Semiregular<typename T> : CopyConstructible<T>, CopyAssignable<T>, HeapAllocatable<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, have equality comparison operators, and can be allocated on the heap.

20.1.8 Convertibility [concept.convertible]

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

1 *Note:* describes types with a conversion (explicit or implicit) from one type to another.

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

2 *Note:* describes types with an implicit conversion from one type to another.

20.1.9 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.

20.1.10 Operator concepts [concept.operator]

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
auto concept HasLogicalNot<typename T> {
  bool operator!(T const&);
}
```

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

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

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

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

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

```
auto concept HasBitXor<typename T, typename U = T> {
  typename result_type;
```

```
  result_type operator^(T const&, U const&);
}
```

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

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

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

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

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

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

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

```
auto concept Dereferenceable<typename T> {
  typename reference;
  reference operator*(T);
}
```

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

```
auto concept Addressable<typename T> {
  typename pointer;
  pointer operator&(T&);
}
```

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

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

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

20.1.11 Arithmetic concepts [concept.arithmetic]

```
concept ArithmeticLike<typename T>
  : Regular<T>, LessThanComparable<T>, HasPlus<T>, HasMinus<T>, HasMultiply<T>, HasDivide<T>,
    HasUnaryPlus<T>, HasNegate<T> {
```

```
  T::T(long long);

  T& operator++(T&);
  T operator++(T& t, int) { T tmp(t); ++t; return tmp; }
  T& operator--(T&);
  T operator--(T& t, int) { T tmp(t); --t; return tmp; }

  requires Convertible<HasUnaryPlus<T>::result_type, T>
        && Convertible<HasNegate<T>::result_type, T>
        && Convertible<HasPlus<T>::result_type, T>
        && Convertible<HasMinus<T>::result_type, T>
        && Convertible<HasMultiply<T>::result_type, T>
        && Convertible<HasDivide<T>::result_type, T>;

  T& operator*=(T&, T);
  T& operator/=(T&, T);
  T& operator+=(T&, T);
  T& operator-=(T&, T);
}
```

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

```
concept IntegralLike<typename T>
  : ArithmeticLike<T>, HasComplement<T>, HasModulus<T>, HasBitAnd<T>, HasBitXor<T>, HasBitOr<T>,
    HasLeftShift<T>, HasRightShift<T> {
  requires Convertible<HasComplement<T>::result_type, T>
        && Convertible<HasModulus<T>::result_type, T>
        && Convertible<HasBitAnd<T>::result_type, T>
        && Convertible<HasBitXor<T>::result_type, T>
        && Convertible<HasBitOr<T>::result_type, T>
        && Convertible<HasLeftShift<T>::result_type, T>
        && Convertible<HasRightShift<T>::result_type, T>;

  T& operator%=(T&, T);
  T& operator&=(T&, T);
  T& operator^=(T&, T);
  T& operator|=(T&, T);
  T& operator<<=(T&, T);
  T& operator>>=(T&, 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 `SignedIntegral<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 `UnsignedIntegral<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 `FloatingPoint<T>` shall be defined in namespace `std`.

20.1.12 Predicates [concept.predicate]

```
auto concept Predicate<typename F, typename... Args> : Callable<F, 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.

20.1.13 Allocators [concept.allocator]

We have kept most of the text of [allocator.requirements] here, although much of it has been moved from tables into numbered paragraphs when translating the allocator requirements into concepts. With the introduction of scoped allocations, this text is somewhat out of date, and will be revised significantly for the next mailing.

1 The library describes a standard set of requirements for *allocators*, which are objects that encapsulate the information about an allocation model. This information includes the knowledge of pointer types, the type of their difference, the type of the size of objects in this allocation model, as well as the memory allocation and deallocation primitives for it. All of the containers (clause **??**) are parameterized in terms of allocators.

[[**Remove Table 39: Descriptive variable definitions**]]

[[**Remove Table 40: Allocator requirements**]]

2 ~~Table 40 describes the requirements on types manipulated through allocators.~~The `Allocator` concept describes the requirements on allocators. ~~All the operations on the allocators are expected to be amortized constant time.~~Each allocator operation shall have amortized constant time complexity. ~~Table 33 describes the requirements on allocator types.~~

```
concept Allocator<typename X> : DefaultConstructible<X>, CopyConstructible<X> {
  ObjectType value_type               = typename X::value_type;
  MutableRandomAccessIterator pointer = typename X::pointer;
  RandomAccessIterator const_pointer  = typename X::const_pointer;
  typename reference                  = typename X::reference;
  typename const_reference            = typename X::const_reference;
  SignedIntegral difference_type      = typename X::difference_type;
  UnsignedIntegral size_type          = typename X::size_type;
```

```
  template<ObjectType T> class rebind = see below;

  requires Convertible<pointer, const_pointer> &&
           Convertible<pointer, value_type*> &&
           SameType<pointer::value_type, value_type> &&
           SameType<pointer::reference, value_type&> &&
           SameType<pointer::reference, reference>;

  requires Convertible<const_pointer, const value_type*> &&
           SameType<const_pointer::value_type, value_type> &&
           SameType<const_pointer::reference, const value_type&> &&
           SameType<const_pointer::reference, const_reference>;

  requires SameType<rebind<value_type>, X>;

  pointer X::allocate(size_type n);
  pointer X::allocate(size_type n, const_pointer p);
  void X::deallocate(pointer p, size_type n);

  size_type X::max_size() const;

  template<ObjectType T>
    X::X(const rebind<T>& y);

  ~~void X::construct(pointer p, const value_type&);~~
  template<typename V>
    requires Convertible<V, value_type>
    void X::construct(pointer p, V&&);

  void X::destroy(pointer p);

  pointer X::address(reference) const;
  const_pointer X::address(const_reference) const;
}
```

`UnsignedIntegral size_type;`

3 *Type:* a type that can represent the size of the largest object in the allocation model

`SignedIntegral difference_type;`

4 *Type:* a type that can represent the difference between any two pointers in the allocation model

`template<ObjectType T> class rebind;`

5 *Type:* The ~~member class~~associated template `rebind` ~~in the table above is effectively a typedef template~~is a template that produces allocators in the same family as `X`: if the name ~~`Allocator`~~`X` is bound to ~~`SomeAllocator<T>`~~ `SomeAllocator<value_type>`, then ~~`Allocator::rebind<U>::other`~~`rebind<U>` is the same type as `SomeAllocator<U>`. The resulting type `SameAllocator<U>` shall meet the requirements of the `Allocator` concept.

The default value for `rebind` is a template `R` for which `R<U>` is `X::template rebind<U>::other`.

The aforementioned default value for `rebind` can be implemented as follows:

```
template<typename Alloc>
struct rebind_allocator {
  template<typename U>
  using rebind = typename Alloc::template rebind<U>::other;
};
```

The default value for `rebind` in the `Allocator` concept is, therefore, `rebind_allocator<X>::template rebind`.

```
pointer X::allocate(size_type n);
pointer X::allocate(size_type n, const_pointer p);
```

6 *Effects:* Memory is allocated for `n` objects of type ~~`T`~~`value_type` but objects are not constructed. [1)]

7 *Returns:* ~~The result is a random access iterator.~~ A pointer to the allocated memory. [*Note:* If `n == 0`, the return value is unspecified. — *end note*]

8 *Throws:* `allocate` may raise an appropriate exception.

```
void X::deallocate(pointer p, size_type n);
```

9 *Preconditions:* All `n` ~~`T`~~value_type objects in the area pointed to by `p` shall be destroyed prior to this call. `n` shall match the value passed to `allocate` to obtain this memory. [*Note:* `p` shall not be ~~null~~singular. — *end note*]

Throws: Does not throw exceptions.

```
size_type X::max_size() const;
```

10 *Returns:* the largest value that can meaningfully be passed to `X::allocate()`

```
template<typename V>
  requires HasConstructor<value_type, V>
  void X::construct(pointer p, V&&);
```

The non-templated `X::construct` has been removed from the `Allocator` requirements because it implies that the `value_type` is `CopyConstructible` (which we do not want as a requirement in the `Allocator` concept). The templated version is more general, allowing in-place and move construction.

11 *Effects:* `::new((void*)p) T(forward<V>(v))`

```
void X::destroy(pointer p);
```

12 *Effects:* `((T*)p)->~T()`

13 The `AllocatorGenerator` concept describes the requirements on types that can be used to generate `Allocator`s.

```
concept AllocatorGenerator<typename X> : Regular<X> {
  typename value_type                = typename X::value_type;
  template<typename T> class rebind = see below;
```

1) It is intended that `a.allocate` be an efficient means of allocating a single object of type `T`, even when `sizeof(T)` is small. That is, there is no need for a container to maintain its own “free list”.

```
  requires SameType<rebind<value_type>, X>;
}
```

```
template<typename T> class rebind;
```

14 *Type:* The ~~member class~~associated template `rebind` ~~in the table above is effectively a typedef template~~is a template that produces allocator generators in the same family as `X`: if the name ~~`Allocator`~~`X` is bound to ~~`SomeAllocator<T>`~~ `SomeAllocator<value_type>`, then ~~`Allocator::rebind<U>::other`~~`rebind<U>` is the same type as `SomeAllocator<U>`. The default value for `rebind` is a template `R` for which `R<U>` is `X::template rebind<U>::other`.

15 Two allocators or allocator generators compare equal with == iff storage allocated from each can be deallocated via the other.

16 Every `Allocator` also meets the requirements of the `AllocatorGenerator` concept:

```
template<Allocator X>
concept_map AllocatorGenerator<X> {
  typedef Allocator<X>::value_type value_type;
  template<typename U> using rebind = Allocator<X>::rebind<U>;
}
```

17 Implementations of containers described in this International Standard are permitted to assume that their `Alloc`~~`ator`~~ template parameter meets the following two additional requirements beyond those in ~~Table 40~~the Allocator concept.

- All instances of a given allocator type are required to be interchangeable and always compare equal to each other.
- ~~The typedef members pointer, const_pointer, size_type, and difference_type are required to be T*, T const*, std::size_t, and std::ptrdiff_t, respectively.~~ The requirements clause may contain the following additional requirements: `SameType<Alloc::pointer, Alloc::value_type*>`, `SameType<Alloc::const_pointer, const Alloc::value_type*>`, `SameType<Alloc::size_type, std::size_t>`, and `SameType<Alloc::difference_type, std::ptrdiff_t>`.

18 Implementors are encouraged to supply libraries that can accept allocators that encapsulate more general memory models and that support non-equal instances. In such implementations, any requirements imposed on allocators by containers beyond those requirements that appear in ~~Table 40~~concept Allocator, and the semantics of containers and algorithms when allocator instances compare non-equal, are implementation-defined.