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]

The following clauses describe utility and allocator requirements, 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

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20.1 Concepts</td>
<td>&lt;concepts&gt;</td>
</tr>
<tr>
<td>Utility components</td>
<td>&lt;utility&gt;</td>
</tr>
<tr>
<td>Tuples</td>
<td>&lt;tuple&gt;</td>
</tr>
<tr>
<td>Type traits</td>
<td>&lt;type_traits&gt;</td>
</tr>
<tr>
<td>Function objects</td>
<td>&lt;functional&gt;</td>
</tr>
<tr>
<td>Memory</td>
<td>&lt;memory&gt;</td>
</tr>
<tr>
<td>&lt;cstdlib&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;cstring&gt;</td>
<td></td>
</tr>
</tbody>
</table>

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

20.1 Concepts [utility.concepts]

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 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 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

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> { }

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

Requires: for every type T that is cv void or that meets the requirement MoveConstructible<T> [20.1.8], the concept map Returnable<T> shall be implicitly defined in namespace std.

concept PointeeType<typename T> { }

Note: describes types to which a pointer can be created.
Requires: for every type \( T \) that is an object type, function type, or \( cv\) void, a concept map \( \text{PointeeType} \) shall be implicitly defined in namespace \( \text{std} \).

```cpp
concept ReferentType<typename T> : PointeeType<T> { }
```

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

Requires: for every type \( T \) that is an object type or function type, a concept map \( \text{ReferentType} \) shall be implicitly defined in namespace \( \text{std} \).

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

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

Requires: for every type \( T \) that is an object type or reference type, a concept map \( \text{VariableType<T> \} \) shall be implicitly defined in namespace \( \text{std} \).

```cpp
concept ObjectType<typename T> : VariableType<T> { }
```

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

Requires: for every type \( T \) that is an object type, a concept map \( \text{ObjectType<T> \} \) shall be implicitly defined in namespace \( \text{std} \).

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

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

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

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

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

Requires: for every type \( T \) that is a class or struct, a concept map \( \text{Class<T> \} \) shall be implicitly defined in namespace \( \text{std} \).

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

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

Requires: for every type \( T \) that is a union, a concept map \( \text{Union<T> \} \) shall be implicitly defined in namespace \( \text{std} \).

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

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

Requires: for every type \( T \) that is a trivial type, a concept map \( \text{TrivialType<T> \} \) shall be implicitly defined in namespace \( \text{std} \).

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

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

General utilities library

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> { }

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

    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> { }

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

    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> { }

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

    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> { }

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

    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> { }

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

    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> { }

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

    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> { }

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

    concept DerivedFrom<typename Derived, typename Base> { }

    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

Note: describes types with an operator `<`.

```cpp
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;
}
}
```

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

```cpp
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;
}
```

Note: describes types whose values can be compared for equality with operator `==`, which is an equivalence relation.
concept TriviallyEqualityComparable<typename T> : EqualityComparable<T> { }

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]

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();
} 

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

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

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

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

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...);
} 

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

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

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&&> { }

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&&>

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; 
  } 
} 

Note: describes types with a public copy constructor.

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

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

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&&); 
} 

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

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

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; 
  } 
} 

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

The CopyAssignble 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.

custom concept TriviallyCopyAssignable<typename T> : CopyAssignble<T> { } 

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

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 TriviallyCopyAssignble<T> shall be implicitly defined in namespace std.
auto concept Swappable<typename T> {
    void swap(T&, T&);
}

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

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

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

20.1.6 Memory allocation

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*);
}

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

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

Note: collects several common requirements supported by most types.

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

Note: describes semi-regular types that are default constructible, have equality comparison operators, and can be allocated on the heap.

20.1.8 Convertibility

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

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&);
}

Note: describes types with an implicit conversion from one type to another.
20.1.9 True

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

1 \textit{Note:} used to express the requirement that a particular integral constant expression evaluate true.

2 \textit{Requires:} a program shall not provide a concept map for the True concept.

20.1.10 Operator concepts

auto concept HasPlus<
  \texttt{typename } T, \texttt{typename } U = T > \{ 
  \texttt{typename } result\_type; 
  result\_type operator\+(T \texttt{const \\&}, U \texttt{const \\&}); 
\}

1 \textit{Note:} describes types with a binary operator\+. 

auto concept HasMinus<
  \texttt{typename } T, \texttt{typename } U = T > \{ 
  \texttt{typename } result\_type; 
  result\_type operator\-(T \texttt{const \\&}, U \texttt{const \\&}); 
\}

2 \textit{Note:} describes types with a binary operator\-. 

auto concept HasMultiply<
  \texttt{typename } T, \texttt{typename } U = T > \{ 
  \texttt{typename } result\_type; 
  result\_type operator\*(T \texttt{const \\&}, U \texttt{const \\&}); 
\}

3 \textit{Note:} describes types with a binary operator\*. 

auto concept HasDivide<
  \texttt{typename } T, \texttt{typename } U = T > \{ 
  \texttt{typename } result\_type; 
  result\_type operator\/(T \texttt{const \\&}, U \texttt{const \\&}); 
\}

4 \textit{Note:} describes types with an operator\/. 

auto concept HasModulus<
  \texttt{typename } T, \texttt{typename } U = T > \{ 
  \texttt{typename } result\_type; 
  result\_type operator%\(T \texttt{const \\&}, U \texttt{const \\&}); 
\}

5 \textit{Note:} describes types with an operator\%. 

auto concept HasUnaryPlus<
  \texttt{typename } T > \{ 
  \texttt{typename } result\_type; 
  result\_type operator\+(T \texttt{const \\&}); 
\}

6 \textit{Note:} describes types with a unary operator\+. 

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

    Note: describes types with a unary operator-.

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

    Note: describes types with an operator<.

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

    Note: describes types with an operator==.

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

    Note: describes types with a logical conjunction operator.

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

    Note: describes types with a logical disjunction operator.

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

    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&);
}

    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&);
}

    Note: describes types with an operator|.

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

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

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

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

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

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

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

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

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

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

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

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

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

**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> {

Draft
20.1 Concepts

General utilities library

\[ T::T(long\ long); \]
\[ T&\ \text{operator++}(T&); \]
\[ T\ \text{operator++}(T&\ t,\ int)\ \{\ T\ \text{tmp}(t);\ \text{++t};\ \text{return}\ \text{tmp};\ \} \]
\[ T&\ \text{operator--}(T&); \]
\[ T\ \text{operator--}(T&\ t,\ int)\ \{\ T\ \text{tmp}(t);\ \text{--t};\ \text{return}\ \text{tmp};\ \} \]

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>;

\[ T&\ \text{operator\%=}(T&,\ T); \]
\[ T&\ \text{operator\&=}(T&,\ T); \]
\[ T&\ \text{operator\^=}(T&,\ T); \]
\[ T&\ \text{operator\|=}(T&,\ T); \]
\[ T&\ \text{operator\ll=}(T&,\ T); \]
\[ T&\ \text{operator\rl=}(T&,\ T); \]

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

concept IntegralLike<
\[ \text{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>;

\[ T&\ \text{operator\%=}(T&,\ T); \]
\[ T&\ \text{operator\&=}(T&,\ T); \]
\[ T&\ \text{operator\^=}(T&,\ T); \]
\[ T&\ \text{operator\|=}(T&,\ T); \]
\[ T&\ \text{operator\ll=}(T&,\ T); \]
\[ T&\ \text{operator\rl=}(T&,\ T); \]

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

concept SignedIntegralLike<
\[ \text{typename}\ T\ ]
: IntegralLike<T> { }

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

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<
\[ \text{typename}\ T\ ]
: IntegralLike<T> { }

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

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> { }

Note: describes floating-point types.

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>; }

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

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.

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]]

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;
}
20.1 Concepts

General utilities library

```cpp
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;
```

```cpp
UnsignedIntegral size_type;
```

```cpp
Type: a type that can represent the size of the largest object in the allocation model
```

```cpp
SignedIntegral difference_type;
```

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

```cpp
template<ObjectType T> class rebind;
```

```cpp
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.
```

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

```cpp
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`.

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

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

**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]

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

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

**Preconditions:** All `n` `T` 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.

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

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

```cpp
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.

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

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

**Effects:** `((T*)p)~T()`

**The `AllocatorGenerator` concept describes the requirements on types that can be used to generate Allocators.**

```cpp
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”.

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

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

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

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

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

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

Implementations of containers described in this International Standard are permitted to assume that their `Allocator` 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>`.

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.