Core Concepts for the C++0x Standard Library

Douglas Gregor and Andrew Lumsdaine
Open Systems Laboratory
Indiana University
Bloomington, IN 47405
dgregor, lums}@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).

This proposal adds new functionality into the Standard Library for concepts, and is meant as a companion to the language wording for concepts, since some of the language wording depends on this library wording. More thorough changes to the Standard Library will follow in future revisions of the “Concepts for the C++0x Standard Library” proposals.

The concepts in this proposal replace the requirements tables currently in [utility.requirements]. This leaves the Standard Library in an interesting (but consistent!) state, where the template requirements of the Standard Library are described in terms of actual concepts, but the templates themselves are not constrained templates. For example, a type T in a Standard Library algorithm might be stated to be CopyConstructible in the current text: previously, that text referred to a requirements table for CopyConstructible, whereas now it refers to the CopyConstructible concept itself. When the Standard Library is fully evolved to use concepts, this informally-stated requirement that T is CopyConstructible will be made formal by a requires clause stating CopyConstructible<T>. Thus, the approach of this proposal is designed to provide an evolutionary step toward complete concepts support in the library, while improving the description of the library and support for concepts with each step.

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. Non-editorial changes from the previous wording are highlighted in green.

Purely editorial comments will be written in a separate, shaded box.
Chapter 20  General utilities library

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  Requirements 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>?? Date and time</td>
<td>&lt;ctime&gt;</td>
</tr>
</tbody>
</table>

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

20.1  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<type name T> { }
    concept PointeeType<type name T> { }
    concept ReferentType<type name T> see below;
    concept VariableType<type name T>
    concept Object<Typename T> see below;
    concept CType<Typename T> see below;
    concept Class<type name T> see below;
    concept Union<type name T> see below;
    concept TrivialType<type name T> see below;
    concept StandardLayoutType<type name T> see below;
    concept LiteralType<type name T> see below;
concept ScalarType<typename T> see below;
concept NonTypeTemplateParameterType<typename T> see below;
concept IntegralConstantExpressionType<typename T> see below;
concept BuiltInIntegralType<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, typename U = T> see below;
auto concept EqualityComparable<typename T, typename U = 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 Constructible<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 Newable<typename T, typename... Args> see below;
auto concept Deletable<typename T> see below;
auto concept ArrayNewable<typename T, typename... Args> see below;
auto concept ArrayDeletable<typename T> see below;
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 Concepts

// 20.1.10 operator concepts:
auto concept Addable<
    typename T,
    typename U = T
> see below;
auto concept Subtractable<
    typename T,
    typename U = T
> see below;
auto concept Multiplicable<
    typename T,
    typename U = T
> see below;
auto concept Divisible<
    typename T,
    typename U = T
> see below;
auto concept Remainder<
    typename T,
    typename U = T
> see below;
auto concept HasUnaryPlus<
    typename T,
    typename U = T
> see below;
auto concept Negatable<
    typename 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

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 Remark: 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> [20.1.8], the concept map Returnable<T> shall be implicitly defined in namespace std.

concept PointeeType<
    typename T
> { }

Draft
4  Remark: 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$T$` shall be implicitly defined in namespace std.

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

6  Remark: 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$T$` shall be implicitly defined in namespace std.

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

8  Remark: 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.

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

10  Remark: 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.

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

12  Remark: 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.

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

14  Remark: 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.

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

16  Remark: 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.

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

18  Remark: 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.

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

Draft
20.1 Concepts

General utilities library

20

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

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

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

**Remark:** 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.

```cpp
class TrivialType
{
    // Trivial type definitions
}
```

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

```cpp
class StandardLayoutType
{
    // Standard layout type definitions
}
```

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

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

**Remark:** 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.

```cpp
class NonTypeTemplateParameterType<typename T> : VariableType { }
```

**Remark:** 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.

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

**Remark:** 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.

```cpp
class BuiltInIntegralType<typename T> : IntegralConstantExpressionType<T> { }
```

**Remark:** 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.

```cpp
class EnumerationType<typename T> : IntegralConstantExpressionType<T> { }
```

**Remark:** 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.

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

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

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

Draft
20.1.2 Comparisons

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

Remark: describes types whose values can be ordered via an inequality operator.

Requires: operator< is a strict weak ordering relation (?).  

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

Remark: describes types whose values can be compared for equality with operator==.

Requires: when T and U are identical, operator== is an equivalence relation, that is, it has the following properties:
   — For all a, a == a.
   — If a == b, then b == a.
   — If a == b and b == c, then a == c.

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

Remark: 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

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

Remark: 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.
20.1 Concepts

General utilities library

TriviallyDestructible<

class TriviallyDestructible<typename T> : Destructible<T> { }

Remark: 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

auto concept Constructible<typename T, typename... Args> : Destructible<T> { }

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

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

Remark: describes types that can be default-constructed.

20.1.5 Copy and move

auto concept MoveConstructible<typename T> : Destructible<T> { }

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

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> { }

axiom CopyPreservation(T x) {
    T(x) == x;
}

Remark: describes types with a public copy constructor.

TriviallyCopyConstructible<

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

Remark: 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 operator=(T&, U&&);
}

Remark: describes types with the ability 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&);
    axiom CopyPreservation(T& x, U y) {
        (x = y, x) == y;
    }
}

Remark: 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> { }

Remark: 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 TriviallyCopyAssignable<T> shall be implicitly defined in namespace std.

auto concept Swappable<typename T> {
    void swap(T&, T&);
}

Remark: 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 Newable<typename T, typename... Args> : ObjectType<T> {
    void* T::operator new(size_t size, Args...);
}

Remark: describes types for which objects can be allocated on the heap via new.
20.1 Concepts

auto concept Deletable<typeparam T> : ObjectType<T> {
    void T::operator delete(void*);
}

Remark: describes types for which objects on the heap can be freed with delete.

auto concept ArrayNewable<typeparam T, typename... Args> : ObjectType<T> {
    void* T::operator new[](size_t size, Args...);
}

Remark: describes types for which arrays of objects can be allocated on the heap via array new.

auto concept ArrayDeletable<typeparam T> : ObjectType<T> {
    void T::operator delete[](void*);
}

Remark: describes types for which an array objects on the heap can be freed with delete[].

auto concept HeapAllocatable<typeparam T> : Newable<T>, Deletable<T>, ArrayNewable<T>, ArrayDeletable<T> { }

Remark: 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<typeparam T> : CopyConstructible<T>, CopyAssignable<T> {
    requires SameType<CopyAssignable<T>::result_type, T&>;
}

Remark: collects several common requirements supported by most types.

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

Remark: 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<typeparam T, typename U> : VariableType<T> {
    explicit operator U(T const&);
}

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

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

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

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

1. **Remark:** 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

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

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

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

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

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

3. **Remark:** describes types with an operator `*`.

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

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

Draft
20.1 Concepts

General utilities library

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

`Remark:` describes types with an operator `%`.

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

`Remark:` describes types with a unary operator `+`.

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

`Remark:` describes types with a unary operator `-`.

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

`Remark:` describes types with a logical conjunction operator.

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

`Remark:` describes types with a logical disjunction operator.

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

`Remark:` describes types with a logical negation operator.

auto concept `HasBitAnd`<typename T, typename U = T> {

Draft
typedef result_type;
result_type operator&(T const& , U const&);
result_type operator&(T const& , U &&);
result_type operator&(T && , U const&);
result_type operator&(T && , U &&);
}

Remark: describes types with a binary operator&.

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

Remark: describes types with an operator|.

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

Remark: describes types with an operator^.

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

Remark: describes types with an operator~.

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

Remark: describes types with an operator<<.

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

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

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

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

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

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

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

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

### 20.1.11 Arithmetic concepts

```cpp
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);
    T& operator/=(T& t, T);
}
```

Draft
T& operator+=(T&, T);
T& operator-=(T&, T);
}

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

concept IntegralLike<
type name 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);
}

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

concept SignedIntegralLike<
type name T> : IntegralLike<T> { }

Remark: 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<
type name T> : IntegralLike<T> { }

Remark: 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<
type name T> : ArithmeticLike<T> { }

Remark: 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<
type name F, type name... Args> : Callable<F, Args...> { requires Convertible<result_type, bool>;

Draft
20.1 Concepts

General utilities library

Remark: 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

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.

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.

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.

```cpp
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;

    requiresConvertible<pointer, const_pointer> &&
    requiresConvertible<pointer, void*> &&
    requiresSameType<pointer::value_type, value_type> &&
    requiresSameType<pointer::reference, value_type&> &&
    requiresSameType<pointer::reference, reference>;

    requiresConvertible<const_pointer, const void*> &&
    requiresConvertible<const_pointer, const value_type&> &&
    requiresSameType<const_pointer::value_type, value_type> &&
    requiresSameType<const_pointer::reference, const value_type&> &&
    requiresSameType<const_pointer::reference, const_reference>;

    requiresSameType rebind<value_type>, X;
}
```

Draft
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;

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 Some-Allocator<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);
Effects: Memory is allocated for \( n \) objects of type \( T \) value_type but objects are not constructed.  

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.

\[
\text{void X::deallocate(pointer p, size_type n);} 
\]

Preconditions: All \( n \) 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. — end note ]

Throws: Does not throw exceptions.

\[
\text{size_type X::max_size()} \text{const;}
\]

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

\[
\text{template<typename V}> \\
\text{ \text{requires Constructible<value_type, V&>}} \\
\text{ \text{void X::construct(pointer p, V&);}}
\]

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.

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

\[
\text{void X::destroy(pointer p);} 
\]

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

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

\[
\text{concept AllocatorGenerator<typename X> : Regular<X> {}} \\
\text{ \text{typename value_type = typename X::value_type;}} \\
\text{ \text{template<typename T> class rebind = see below;}} \\
\text{ \text{requires SameType<rebind<value_type>, X>;}} \\
\text{}}
\]

\[
\text{template<typename T> class rebind;}
\]

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\( <\text{value_type}> \), then Allocator::rebind\( <U> :: \text{other rebind}<U> \) is the same type as SomeAllocator\( <U> \). The default value for rebind is a template \( R \) for which \( R<U> \) is X::template rebind\( <U> :: \text{other} \).

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

\[1\]It is intended that a.\( \text{allocate} \) be an efficient means of allocating a single object of type \( T \), even when \( \text{sizeof}(T) \) is small. That is, there is no need for a container to maintain its own "free list".

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

```cpp
template<Allocator X>
concept 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.

Acknowledgments

Thanks to Walter Brown and Daniel Krügler for providing corrections to and noting omissions from this document.