Core Concepts for the C++0x Standard Library
(Revision 2)

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).

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 N2572

— Added the Has*Assign concepts, which are required by valarray and friends.
— Split MoveAssignable into the two-parameter HasMoveAssign and the one-parameter MoveAssignable; the same split is also used for CopyAssignable and HasCopyAssign.
— Added the MemberPointeeType concept, and revised ReferentType to permit reference types.
— Renamed HeapAllocatable to FreeStoreAllocatable, which better reflects the terminology of the standard.
— Changed the IntegralLike requirement for a long long constructor into a requirement for a intmax_t constructor.
— ObjectType now refines MemberPointeeType.
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>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20.1 Requirements</td>
<td>&lt;concepts&gt;</td>
</tr>
<tr>
<td>[utility] Utility components</td>
<td>&lt;utility&gt;</td>
</tr>
<tr>
<td>[tuple] Tuples</td>
<td>&lt;tuple&gt;</td>
</tr>
<tr>
<td>[meta] Type traits</td>
<td>&lt;type_traits&gt;</td>
</tr>
<tr>
<td>[function.objects] Function objects</td>
<td>&lt;functional&gt;</td>
</tr>
<tr>
<td>[memory] Memory</td>
<td>&lt;memory&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;cstdlib&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;cstring&gt;</td>
</tr>
<tr>
<td>[date.time] 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<typename T> { }
   concept PointeeType<typename T> { }
   concept MemberPointeeType<typename T> see below;
   concept ReferentType<typename T> { }
   concept VariableType<typename T> { }
   concept ObjectType<typename T> see below;
   concept 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>  see below;  
auto concept CopyAssignable<typename T>  see below;  
concept TriviallyCopyAssignable<typename T>  see below;  
auto concept Swappable<typename T>  see below;  

// 20.1.6, memory allocation:
auto concept FreeStoreAllocatable<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 conceptConvertible<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;
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].

```cpp
concept Returnable< typename T > { }  
Draft
```
Note: Describes types that can be used as the return type of a function.

Requires: for every non-array 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> { }

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 PointeeType shall be implicitly defined in namespace std.

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

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

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

concept ReferentType<typename T> { }

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 ReferentType shall be implicitly defined in namespace std.

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 VariableType<T> shall be implicitly defined in namespace std.

concept ObjectType<typename T> : VariableType<T>, MemberPointeeType<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 ObjectType<T> shall be implicitly defined in namespace std.

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 ClassType<T> shall be implicitly defined in namespace std.

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

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

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

concept Union<typename T> : ClassType<T> { }
20.1 Concepts

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

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> { }

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

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> { }

Note: 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.

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 a 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 <.

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 ([alg.sorting]).

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

Draft
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.

class 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]

class 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 all resources owned by the object are reclaimed.

class 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]

class HasConstructor<typename T, typename... Args> : Destructible<T> {
   T::T(Args...);
}

Draft
9 General utilities library

20.1 Concepts

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

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

auto concept MoveConstructible<
typename T> : 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&> { 
  axiom CopyPreservation(T x) {
    T(x) == x;
  }
}

Note: describes types with a public copy constructor.

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.

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

result_type T::operator=(T&& rv); // inherited from HasMoveAssign

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> : MoveAssignable<T> { 
  axiom CopyPreservation(T& x, T y) {
    (x = y, x) == y;
  }
}

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

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

```cpp
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.

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

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

```cpp
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]

```cpp
auto concept FreeStoreAllocatable<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 free store with `new` and `delete`.

## 20.1.7 Regular types

[concept.regular]

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

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

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

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

## 20.1.8 Convertibility

[concept.convertible] Draft
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 T to U.

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

Note: describes types with an implicit conversion from one type to another T to U.

20.1.9 True

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

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

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

20.1.10 Operator concepts

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

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&);
}

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&);
}

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&);
}

Note: describes types with an operator/.
auto concept HasModulus<typename T, typename U = T> {  
  typename result_type;  
  result_type operator%(T const&, U const&); 
} 

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

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

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

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

\textit{Note:} describes types with a unary \texttt{operator-}.

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

\textit{Note:} describes types with an \texttt{operator<}.

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

\textit{Note:} describes types with an \texttt{operator=}.

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

\textit{Note:} describes types with a logical conjunction operator.

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

\textit{Note:} describes types with a logical disjunction operator.

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

\textit{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&.
20.1 Concepts

General utilities library  14

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....

auto concept HasMoveAssign<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 (which may have a different type), potentially altering the rvalue.

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

Note: describes types with the ability to assign to an object (which may have a different type).

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

Note: describes types with an operator+=.

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

Note: describes types with an operator−=.

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

Note: describes types with an operator∗=.

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

Note: describes types with an operator/=.

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

Note: describes types with an operator%=

Draft
Note: describes types with an operator\% =.

```cpp
class HasBitAndAssign<typename T, typename U = T> {
  typename result_type;
  result_type operator\%(T&, const U&);
}
```

Note: describes types with an operator\& =.

```cpp
class HasBitOrAssign<typename T, typename U = T> {
  typename result_type;
  result_type operator\|(T&, const U&);
}
```

Note: describes types with an operator\| =.

```cpp
class HasBitXorAssign<typename T, typename U = T> {
  typename result_type;
  result_type operator\^(T&, const U&);
}
```

Note: describes types with an operator^=.

```cpp
class HasLeftShiftAssign<typename T, typename U = T> {
  typename result_type;
  result_type operator<<=(T&, const U&);
}
```

Note: describes types with an operator<<=.

```cpp
class HasRightShiftAssign<typename T, typename U = T> {
  typename result_type;
  result_type operator>>=(T&, const U&);
}
```

Note: describes types with an operator>>=.

20.1.11 Arithmetic concepts

concept ArithmeticLike<typename T>
  : Regular<T>, LessThanComparable<T>, HasPlus<T>, HasMinus<T>, HasMultiply<T>, HasDivide<T>,
    HasUnaryPlus<T>, HasNegate<T> {
    T::T(long long intmax_t);
    T::T(long double);
    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>
  }
20.1 Concepts

General utilities library

++ 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);

} 

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);

} 

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

concept SignedIntegralLike<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 SignedIntegralLike<T> shall be defined in namespace std.

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

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 UnsignedIntegralLike<T> shall be defined in namespace std.

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

Note: describes floating-point types.
8 \textit{Requires:} for every floating point type T ([basic.fundamental]), an empty concept map \texttt{FloatingPointLike<T>} shall be defined in namespace std.

20.1.12 Predicates [concept.predicate]

\begin{verbatim}
auto concept Predicate<typename F, typename... Args> : Callable<F, Args...> {
  requires Convertible<result_type, bool>;
}
\end{verbatim}

1 \textit{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 \textit{Requires:} predicate function objects shall not apply any non-constant function through the predicate arguments.

Acknowledgments

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