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

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

Document number: N2082=06-0152
Revises document number: N2038=06-0108
Date: 2006-09-09
Project: Programming Language C++, Library Working Group
Reply-to: Douglas Gregor <dgregor@osl.iu.edu>

Introduction

This document proposes changes to Chapter 20 of the C++ Standard Library in order to make full use of concepts [1]. Most of the changes in this document have been verified to work with ConceptGCC and its modified Standard Library implementation. We make every attempt to provide complete backward compatibility with the pre-concept Standard Library, and note each place where we have knowingly changed semantics.

This document is formatted in the same manner as the working draft of the C++ standard (N2009). Future versions of this document will track the working draft and the concepts proposal as they evolve. Wherever the numbering of a (sub)section matches a section of the working paper, the text in this document should be considered replacement text, unless editorial comments state otherwise. All editorial comments will have a gray background. Changes to the replacement text are categorized and typeset as additions, ~~removals~~, or ~~changes~~modifications..

Chapter 20 General utilities library [lib.utilities]

2 The following clauses describe utility and allocator ~~requirements~~concepts, utility components, tuples, type traits templates, function objects, dynamic memory management utilities, and date/time utilities, as summarized in Table 27.

Table 27: General utilities library summary

Subclause	Header(s)
20.1 Requirements	`<concepts>`
20.2 Utility components	`<utility>`
?? Tuples	`<tuple>`
?? Type traits	`<type_traits>`
?? Function objects	`<functional>`
?? Memory	`<memory>` `<cstdlib>` `<cstring>`
?? Date and time	`<ctime>`

20.1 Requirements [lib.utility.requirements]

1 20.1 describes requirements on template arguments as concepts. 20.1.1 through 20.1.4 ~~describe requirements on types~~define concepts used to ~~instantiate~~constrain templates. 20.1.18 describes ~~the requirements on storage allocators~~storage allocator concepts.

Header `<concepts>` synopsis

Note: Synchronize this with the rest of the text.

20.1.1 Equality comparison [lib.equalitycomparable]

1 Concept `EqualityComparable` requires that two values be comparable with `operator==`.

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

2 When T and U are identical, `operator==` is an equivalence relation, that is, it satisfies the following properties:

— For all `a`, `a == a`.

— If `a == b`, then `b == a`.

— If `a == b` and `b == c`, then `a == c`.

20.1.2 Less than comparison [lib.lessthancomparable]

1 Concept `LessThanComparable` requires the ability to order values via `operator<`.

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

2 `operator<` is a strict weak ordering relation (**??**)

20.1.3 Copy construction [lib.copyconstructible]

1 Concept `CopyConstructible` requires the ability to create and destroy copies of an object. [1)]

```
auto concept CopyConstructible<typename T> {
  T::T(T);2)
  T::~T();
};
```

20.1.4 Swapping [lib.swappable]

1 Concept `Swappable` requires that two values `t` and `u` can be swapped, after which `t` has the value originally held by `u` and `u` has the value originally held by `t`.

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

2 [[**Remove paragraph 2**]]

20.1.5 Default construction [lib.default.con.req]

1 ~~The default constructor is not required.~~ Certain container class member function signatures specify the default constructor as a default argument. `T()` shall be a well-defined expression (**??**) if one of those signatures is called using the default argument (**??**).

2 Concept `DefaultConstructible` requires the existence of a default constructor.

```
auto concept DefaultConstructible<typename T> {
  T::T();
};
```

[1)] Table 30 also contains the valid expressions &t and &u. However, we omit these requirements because we need references to model CopyConstructible.

[2)] This signature also covers construction from a non-`const` value of type `T`

20.1.6 Assignment [lib.assignable]

We have moved the Assignable requirement from Section 23.1, paragraph 4 and Table 79, here, because assignability has nothing to do with containers.

The Assignable requirements in C++03 specify that `operator=` must return a `T&`. This is too strong a requirement for most of the uses of `Assignable`, so we have weakened `Assignable` 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.

1 Concept `Assignable` requires the existence of a suitable assignment operator.

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

20.1.7 Regular types [lib.regular]

1 Concept `SemiRegular` collects several common requirements for types that support many regular operations (default construction, copy construction, copy assignment).

```
auto concept SemiRegular<typename T>
  : DefaultConstructible<T>, CopyConstructible<T>, Assignable<T> { }
```

2 Concept `Regular` describes semi-regular types that have equality comparison operators.

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

20.1.8 Convertibility [lib.convertible]

1 Concept `Convertible` requires an implicit conversion from one type to another.

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

template<typename T> concept_map Convertible<T, T> {};
template<typename T> concept_map Convertible<T, T&> {};
template<typename T> concept_map Convertible<T, const T&> {};
```

20.1.9 Same type [lib.sametype]

1 Concept `SameType` requires that its two type parameters have precisely the same type.[3)]

```
concept SameType<typename T, typename U> { /* unspecified */ };
template<typename T> concept_map SameType<T, T> { /* unspecified */ };
```

20.1.10 True [lib.true]

1 Concept `True` requires that its argument (a `bool` value that must be an integral constant expression) be true.

3) Compiler support is required to correctly implement the type-checking semantics of the `SameType` concept.

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

20.1.11 Deferenceable [lib.dereferenceable]

1 Concept `Dereferenceable` requires the existence of a dereference operator `*`.

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

20.1.12 Numeric [lib.requirements.numeric]

1 Concept `Arithmic` requires all of the operations available on arithmetic types.

```
concept Arithmetic<typename T>
  : DefaultConstructible<T>, CopyConstructible<T>,
    LessThanComparable<T>, EqualityComparable<T> {
  T::T(long long);

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

  where Assignable<T> && SameType<Assignable<T>::result_type, T&>;
}
```

2 Concept `Integral` describes the requirements for integral types.

```
concept Integral<typename T> : Arithmetic<T> {
  T& operator++(T&);
  T operator++(T&, int);
  T& operator--(T&);
  T operator--(T&, int);

  T operator%(T, 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);
  T& operator^=(T&, T);
  T operator<<(T, T);
  T& operator<<=(T&, T);
  T operator>>(T, T);
  T& operator>>=(T&, T);
}
```

3 Concept `SignedIntegral` describes signed integral types.

```
concept SignedIntegral<typename T> : Integral<T> { };
```

4 For every built-in signed integral type `T`, there exists an empty concept map `SignedIntegral<T>`.

```
  concept_map SignedIntegral<signed char> { };
  concept_map SignedIntegral<short> { };
  concept_map SignedIntegral<int> { };
  concept_map SignedIntegral<long> { };
  concept_map SignedIntegral<long long> { };
```

5 Concept `UnsignedIntegral` describes unsigned integral types.

```
concept UnsignedIntegral<typename T> : Integral<T> { };
```

6 For every built-in unsigned integral type `T`, there exists an empty concept map `UnsignedIntegral<T>`.

```
  concept_map UnsignedIntegral<unsigned char> { };
  concept_map UnsignedIntegral<unsigned short> { };
  concept_map UnsignedIntegral<unsigned int> { };
  concept_map UnsignedIntegral<unsigned long> { };
  concept_map UnsignedIntegral<unsigned long long> { };
```

7 If `char` is a signed integral type, there shall exist an empty concept map `SignedIntegral<char>`; otherwise, there shall exist an empty concept map `UnsignedIntegral<char>`.

8 If `wchar_t` is a signed integral type, there shall exist an empty concept map `SignedIntegral<wchar_t>`; otherwise, there shall exist an empty concept map `UnsignedIntegral<wchar_t>`.

9 The `Floating` concept describes floating-point numbers.

```
concept Floating<typename T> : Arithmetic<T> { }
```

10 For every built-in floating point type `T`, there exists an empty concept map `Floating<T>`.

```
  concept_map Floating<float> { }
  concept_map Floating<double> { }
```

20.1.13 Addable [lib.addable]

1 Concept `Addable` requires that two values be addable via `operator+`.

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

20.1.14 Subtractable [lib.subtractable]

1 Concept `Subtractable` requires that two values be subtractable via `operator-`.

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

20.1.15 Multiplicable [lib.multiplicable]

1 Concept `Multiplicable` requires that two values be addable via `operator*`.

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

20.1.16 Callable [lib.callable]

1 The `Callable` family of concepts–`Callable0`, `Callable1`, ..., `CallableM` requires that the given parameter `F` be callable given arguments of types `T1`, `T2`, ..., `TN`.

```
auto concept CallableN<typename F, typename T1, typename T2, ..., typename TN> {
  typename result_type;
  result_type operator()(F&, T1, T2, ..., TN);
};
```

20.1.17 Predicates [lib.predicate]

1 The `Predicate` concept requires that a function object be callable with a single argument, the result of which can be used in a context that requires a `bool`.

```
auto concept Predicate<typename F, typename T1> : Callable1<F, T1> {
  where Convertible<result_type, bool>;
};
```

2 The `BinaryPredicate` concept requires that a function object be callable with two arguments, the result of which can be used in a context that requires a `bool`.

```
auto concept BinaryPredicate<typename F, typename T1, typename T2> : Callable2<F, T1, T2> {
  where Convertible<result_type, bool>;
};
```

3 Predicate function objects shall not apply any non-constant function through the predicate arguments.

20.1.18 Allocator requirements [lib.allocator.requirements]

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

[[**Remove Table 32: Descriptive variadic definitions**]]

[[Remove Table 33: Allocator requirements]]

2 ~~Table 32 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. ~~Table 33 describes the requirements on allocator types.~~

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

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

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

  X::X(const X&);
  pointer X::allocate(size_type n);
  pointer X::allocate(size_type n, const_pointer p);
  pointer X::deallocate(pointer p, size_type n);
  size_type X::max_size();
  void X::construct(pointer p, value_type);
  void X::destroy(pointer p);
  pointer X::address(reference);
  const_pointer X::address(const_reference);
}
```

3 ~~The member class template rebind in the table above is effectively a typedef template: if the name Allocator is bound to SomeAllocator<T>, then Allocator::rebind<U>::other is the same type as SomeAllocator<U>.~~

At present, we do not support `rebind`, because we are not certain what the intended requirements actually are.

`UnsignedIntegral size_type;`

4 a type that can represent the size of the largest object in the allocation model

`SignedIntegral difference_type;`

5 a type that can represent the difference between any two pointers in the allocation model

`pointer X::allocate(size_type n);`

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

6 Memory is allocated for `n` objects of type `T` but objects are not constructed. `allocate` may raise an appropriate exception. The result is a random access iterator. [4] [*Note:* If `n == 0`, the return value is unspecified. — *end note*]

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

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

`size_type X::max_size();`

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

`void X::construct(pointer p, value_type);`

9 *Effects:* `::new((void*)p) T(t)`

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

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

11 Two allocators compare equal with == iff iff storage allocated from each can be deallocated via the other.

12 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 33~~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 `Allocator` concept may contain the following requirements: `SameType<pointer, value_type*>`, `SameType<const_pointer, const value_type*>`, `SameType<size_type, std::size_t>`, and `SameType<difference_type, std::ptrdiff_t>`.

13 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 33~~concept Allocator, and the semantics of containers and algorithms when allocator instances compare non-equal, are implementation-defined.

20.2 Utility components [lib.utility]

1 This subclause contains some basic function and class templates that are used throughout the rest of the library.

Header `<utility>` synopsis

```
namespace std {
  // 20.2.1, operators:
  namespace rel_ops {
    template<EqualityComparable T> bool operator!=(const T&, const T&);
```

[4] 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".

```
      template<LessThanComparable T> bool operator> (const T&, const T&);
      template<LessThanComparable T> bool operator<=(const T&, const T&);
      template<LessThanComparable T> bool operator>=(const T&, const T&);
    }

    // 20.2.2, pairs:
    template <class T1, class T2> struct pair;
    template <~~class~~EqualityComparable T1, ~~class~~EqualityComparable T2>
      bool operator==(const pair<T1,T2>&, const pair<T1,T2>&);
    template <~~class~~LessThanComparable T1, ~~class~~LessThanComparable T2>
      bool operator< (const pair<T1,T2>&, const pair<T1,T2>&);
    template <~~class~~EqualityComparable T1, ~~class~~EqualityComparable T2>
      bool operator!=(const pair<T1,T2>&, const pair<T1,T2>&);
    template <~~class~~LessThanComparable T1, ~~class~~LessThanComparable T2>
      bool operator> (const pair<T1,T2>&, const pair<T1,T2>&);
    template <~~class~~LessThanComparable T1, ~~class~~LessThanComparable T2>
      bool operator>=(const pair<T1,T2>&, const pair<T1,T2>&);
    template <~~class~~LessThanComparable T1, ~~class~~LessThanComparable T2>
      bool operator<=(const pair<T1,T2>&, const pair<T1,T2>&);
    template <~~class~~CopyConstructible T1, ~~class~~CopyConstructible T2> pair<T1,T2> make_pair(T1, T2);
  }
```

20.2.1 Operators [lib.operators]

By adding concept constraints to the operators in `rel_ops`, we eliminate nearly all of the problems with `rel_ops` that caused them to be banished. We could consider bringing them back into namespace `std`, if they are deemed useful.

1 To avoid redundant definitions of `operator!=` out of `operator==` and operators `>`, `<=`, and `>=` out of `operator<`, the library provides the following:

```
template <EqualityComparable T> bool operator!=(const T& x, const T& y);
```

2 ~~Requires: Type T is EqualityComparable (20.1.1).~~

3 *Returns:* `!(x == y)`.

```
template <LessThanComparable T> bool operator>(const T& x, const T& y);
```

4 ~~Requires: Type T is LessThanComparable (20.1.2).~~

5 *Returns:* `y < x`.

```
template <LessThanComparable T> bool operator<=(const T& x, const T& y);
```

6 ~~Requires: Type T is LessThanComparable (20.1.2).~~

7 *Returns:* `!(y < x)`.

```
template <LessThanComparable T> bool operator>=(const T& x, const T& y);
```

8 ~~Requires: Type T is LessThanComparable (20.1.2).~~

9 *Returns:* `!(x < y)`.

10 In this library, whenever a declaration is provided for an `operator!=`, `operator>`, `operator>=`, or `operator<=`, and requirements and semantics are not explicitly provided, the requirements and semantics are as specified in this clause.

20.2.2 Pairs [lib.pairs]

1 The library provides a template for heterogeneous pairs of values. The library also provides a matching function template to simplify their construction and several templates that provide access to `pair` objects as if they were `tuple` objects (see **??** and **??**).

```
template <class T1, class T2>
struct pair {
  typedef T1 first_type;
  typedef T2 second_type;

  T1 first;
  T2 second;
  where DefaultConstructible<T1> && DefaultConstructible<T2> pair();
  where CopyConstructible<T1> && CopyConstructible<T2> pair(const T1& x, const T2& y);
  template<class U, class V>
    where Convertible<U, T1> && Convertible<U, T2>
    pair(const pair<U, V> &p);
};
```

`where DefaultConstructible<T1> && DefaultConstructible<T2> pair();`

2 *Effects:* Initializes its members as if implemented: `pair() : first(), second() {}`

`where CopyConstructible<T1> && CopyConstructible<T2> pair(const T1& x, const T2& y);`

3 *Effects:* The constructor initializes `first` with *x* and `second` with *y*.

```
template<class U, class V>
  where Convertible<U, T1> && Convertible<U, T2>
  pair(const pair<U, V> &p);
```

4 *Effects:* Initializes members from the corresponding members of the argument, performing implicit conversions as needed.

template <~~class~~EqualityComparable T1, ~~class~~EqualityComparable T2>
  bool operator==(const pair<T1, T2>& x, const pair<T1, T2>& y);

5 *Returns:* `x.first == y.first && x.second == y.second`.

template <~~class~~LessThanComparable T1, ~~class~~LessThanComparable T2>
  bool operator<(const pair<T1, T2>& x, const pair<T1, T2>& y);

6 *Returns:* `x.first < y.first || (!(y.first < x.first) && x.second < y.second)`.

template <~~class~~CopyConstructible T1, ~~class~~CopyConstructible T2>
  pair<T1, T2> make_pair(T1 x, T2 y);

7 *Returns:* `pair<T1, T2>(x, y)`.[5)]

8 [*Example:* In place of:

```
return pair<int, double>(5, 3.1415926);    // explicit types
```

a C++ program may contain:

```
return make_pair(5, 3.1415926);            // types are deduced
```

— end example]

```
tuple_size<pair<T1, T2> >::value
```

9 *Returns:* integral constant expression.

10 *Value:* 2.

```
tuple_element<0, pair<T1, T2> >::type
```

11 *Value:* the type `T1`.

TR1:
```
tuple_element<1, pair<T1, T2> >::type
```

12 *Value:* the type T2.

TR1:
```
template<int I, class T1, class T2>
  P& get(pair<T1, T2>&);
```

TR1:
```
template<int I, class T1, class T2>
  const P& get(const pair<T1, T2>&);
```

13 *Return type:* If `I == 0` then `P` is `T1`, if `I == 1` then `P` is `T2`, and otherwise the program is ill-formed.

14 *Returns:* If `I == 0` returns `p.first`, otherwise returns `p.second`.

Bibliography

[1] Douglas Gregor and Bjarne Stroustrup. Concepts (revision 1). Technical Report N2081=06-0151, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, October 2006.

5) According to (**??**), an implementation is permitted to not perform a copy of an argument, thus avoiding unnecessary copies.