Concepts for the C++0x Standard Library: Utilities

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

Document number: N2038=06-0108
Date: 2006-06-23
Project: Programming Language C++, Library Working Group
Reply-to: Douglas Gregor <dgregor@cs.indiana.edu>

Introduction

This document proposes changes to Chapter 20 of the C++ Standard Library in order to make full use of concepts [1]. Unless otherwise specified, all 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. **??** 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);
}
```

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, U);
};
```

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();
};
```

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.

[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`

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 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&> {};
```

20.1.8 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 ∗/ };
```

20.1.9 True [lib.true]

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

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

20.1.10 Deferenceable [lib.dereferenceable]

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

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

20.1.11 Integral [lib.integral]

1 Concept `Integral` requires all of the operations available on built-in integral types.

```
concept Integral<typename T>
  : DefaultConstructible<T>, CopyConstructible<T>,
```

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

```
    LessThanComparable<T>, EqualityComparable<T> {
  T::T(long long);

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

  bool operator>(T, T);
  bool operator<=(T, T);
  bool operator>=(T, T);
  bool operator!=(T, T);

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

2 Concept `SignedIntegral` requires all of the operations of built-in signed integral types.

```
concept SignedIntegral<typename T> : Integral<T> {
  T operator-(T);
};
```

3 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> { };
```

4 Concept `UnsignedIntegral` requires all of the operations of built-in unsigned integral types.

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

5 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> { };
```

6 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>`.

7 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>`.

20.1.12 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);
};
```

20.1.13 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);
};
```

20.1.14 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);
};
```

20.1.15 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.

20.2 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&);
      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&);
    }

    // ??, pairs:
    template <class T1, class T2> struct pair;
    template <class T1, class T2>
      bool operator==(const pair<T1,T2>&, const pair<T1,T2>&);
    template <class T1, class T2>
      bool operator< (const pair<T1,T2>&, const pair<T1,T2>&);
    template <class T1, class T2>
      bool operator!=(const pair<T1,T2>&, const pair<T1,T2>&);
    template <class T1, class T2>
      bool operator> (const pair<T1,T2>&, const pair<T1,T2>&);
    template <class T1, class T2>
      bool operator>=(const pair<T1,T2>&, const pair<T1,T2>&);
    template <class T1, class T2>
      bool operator<=(const pair<T1,T2>&, const pair<T1,T2>&);
    template <class T1, class 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.

Bibliography

[1] Douglas Gregor and Bjarne Stroustrup. Concepts. Technical Report N2042=06-0112, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, June 2006.