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

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

Document number: N2037=06-0107
Date: 2006-06-21
Project: Programming Language C++, Library Working Group
Reply-to: Douglas Gregor <dgregor@cs.indiana.edu>

Introduction

This document proposes changes to Chapter 17 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 (N1804). 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 17 Library introduction [lib.library]

17.1 Definitions [lib.definitions]

17.1.19 traits class [defns.traits]

~~a class that encapsulates a set of types and functions necessary for class templates and function templates to manipulate objects of types for which they are instantiated. Traits classes defined in clauses 21, 22 and 27 are chararacter traits, which provide the character handling support needed by the string and iostream classes.~~

17.1.20 wide-oriented iostream classes [defns.wide.iostream]

the instantiations of the iostream class templates on the character container class `wchar_t` ~~and the default value of the traits parameter~~ (**??**).

17.3 Method of description (Informative) [lib.description]

17.3.1 Structure of each subclause [lib.structure]

17.3.1.2 Requirements [lib.structure.requirements]

1 The library can be extended by a C++ program. Each clause, as applicable, describes the requirements that such extensions must meet. Such extensions are generally one of the following:

- Template arguments
- Derived classes
- Containers, iterators, and/or algorithms that meet an interface convention

2 ~~The string and iostreams components use an explicit representation of operations required of template arguments. They use a class template char_traits to define these constraints.~~

3 ~~Interface convention requirements are stated as generally as possible. Instead of stating "class X has to define a member function operator++()", the interface requires "for any object x of class X, ++x is defined." That is, whether the operator is a member is unspecified.~~

4 Requirements are stated in terms of ~~well-defined expressions~~concepts, which define ~~valid terms~~capabilities of the types that satisfy the requirements. For every set of requirements there is a ~~table~~concept that specifies ~~an initial set of the valid expressions~~the requirements and their semantics (**??**, **??**, **??**). Any generic algorithm (clause **??**) that uses the ~~requirements~~concepts ~~is described in terms of the valid expressions for~~places requirements on its formal type parameters.

5 ~~Template argument requirements are sometimes referenced by name. See 17.3.2.1~~.

6 In some cases the semantic requirements are presented as C++ code. Such code is intended as a specification of equivalence of a construct to another construct, not necessarily as the way the construct must be implemented.[1)]

17.3.2 Other conventions [lib.conventions]

17.3.2.1 Type descriptions [lib.type.descriptions]

1 ~~The Requirements subclauses may describe names that are used to specify constraints on template arguments.[2)] These names are used in clauses 20, 23, 25, and 26 to describe the types that may be supplied as arguments by a C++ program when instantiating template components from the library.~~

2 Certain types defined in clause **??** are used to describe implementation-defined types. They are based on other types, but with added constraints.

17.4 Library-wide requirements [lib.requirements]

17.4.1 Library contents and organization [lib.organization]

17.4.1.1 Library contents [lib.contents]

1 The C++ Standard Library provides definitions for the following types of entities: Macros, Values, Types, Concepts, Concept maps, Templates, Classes, Functions, Objects.

2 All library entities except macros, `operator new` and `operator delete` are defined within the namespace `std` or namespaces nested within namespace `std`.

3 Whenever a name `x` defined in the standard library is mentioned, the name `x` is assumed to be fully qualified as `::std::x`, unless explicitly described otherwise. For example, if the Effects section for library function `F` is described as calling library function `G`, the function `::std::G` is meant.

17.4.1.2 Headers [lib.headers]

1 The elements of the C++ Standard Library are declared or defined (as appropriate) in a *header*.[3)]

2 The C++ Standard Library provides ~~33~~34 *C++ headers*, as shown in Table 11.

Table 11: C++ Library Headers

`<algorithm>`	`<functional>`	`<limits>`	`<ostream>`	`<streambuf>`
`<bitset>`	`<iomanip>`	`<list>`	`<queue>`	`<string>`
`<complex>`	`<ios>`	`<locale>`	`<set>`	`<typeinfo>`
`<concepts>`	`<iosfwd>`	`<map>`	`<sstream>`	`<utility>`
`<deque>`	`<iostream>`	`<memory>`	`<stack>`	`<valarray>`
`<exception>`	`<istream>`	`<new>`	`<stdexcept>`	`<vector>`
`<fstream>`	`<iterator>`	`<numeric>`	`<strstream>`	

3 The facilities of the Standard C Library are provided in 18 additional headers, as shown in Table 12.

[1)] Although in some cases the code given is unambiguously the optimum implementation.

[2)] ~~Examples from 20.1 include: EqualityComparable, LessThanComparable, CopyConstructable, etc. Examples from 24.1 include: InputIterator, ForwardIterator, Function, Predicate, etc.~~

[3)] A header is not necessarily a source file, nor are the sequences delimited by < and > in header names necessarily valid source file names (**??**).

Table 12: C++ Headers for C Library Facilities

<cassert>	<ciso646>	<csetjmp>	<cstdio>	<ctime>
<cctype>	<climits>	<csignal>	<cstdlib>	<cwchar>
<cerrno>	<clocale>	<cstdarg>	<cstring>	<cwctype>
<cfloat>	<cmath>	<cstddef>		

4 Except as noted in clauses **??** through **??** and Annex **??** the contents of each header c*name* shall be the same as that of the corresponding header *name*.h, as specified in ISO/IEC 9899:1990 Programming Languages C (clause 7), or ISO/IEC:1990 Programming Languages — C AMENDMENT 1: C Integrity, (clause 7), as appropriate, as if by inclusion. In the C++ Standard Library, however, the declarations and definitions (except for names which are defined as macros in C) are within namespace scope (**??**) of the namespace std.

5 Names which are defined as macros in C shall be defined as macros in the C++ Standard Library, even if C grants license for implementation as functions. [*Note:* the names defined as macros in C include the following: assert, offsetof, setjmp, va_arg, va_end, and va_start. — *end note*]

6 Names that are defined as functions in C shall be defined as functions in the C++ Standard Library.[4)]

7 Identifiers that are keywords or operators in C++ shall not be defined as macros in C++ standard library headers.[5)]

8 **??**, Standard C library headers, describes the effects of using the *name*.h (C header) form in a C++ program.[6)]

17.4.1.3 Reserved names [lib.reserved.names]

1 It is undefined for a C++ program to add declarations or definitions to namespace std or namespaces within namespace std unless otherwise specified. A program may add template specializations for any standard library template to namespace std. A program may add concept maps for any standard library concept to namespace std, unless explicitly prohibited. Such a specialization or concept map (complete or partial) of a standard library template results in undefined behavior unless the declaration depends on a user-defined type of external linkage and unless the specialization meets the standard library requirements for the original template.[7)] A program may explicitly instantiate any templates in the standard library only if the declaration depends on the name of a user-defined type of external linkage and the instantiation meets the standard library requirements for the original template.

2 The C++ Standard Library reserves the following kinds of names:

— Macros

— Global names

— Names with external linkage

[4)] This disallows the practice, allowed in C, of providing a "masking macro" in addition to the function prototype. The only way to achieve equivalent "inline" behavior in C++ is to provide a definition as an extern inline function.

[5)] In particular, including the standard header <iso646.h> or <ciso646> has no effect.

[6)] The ".h" headers dump all their names into the global namespace, whereas the newer forms keep their names in namespace std. Therefore, the newer forms are the preferred forms for all uses except for C++ programs which are intended to be strictly compatible with C.

[7)] Any library code that instantiates other library templates must be prepared to work adequately with any user-supplied specialization that meets the minimum requirements of the Standard.

3 If the program declares or defines a name in a context where it is reserved, other than as explicitly allowed by this clause, the behavior is undefined.

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.