Proposed Wording for Concepts

Authors: Douglas Gregor, Indiana University
Bjarne Stroustrup, Texas A&M University
Document number: N2193=07-0053
Revises document number: N2081=06-0151
Date: 2007-03-12
Project: Programming Language C++, Evolution Working Group
Reply-to: Douglas Gregor <doug.gregor@gmail.com>

Introduction

This document provides proposed wording for concepts. Readers unfamiliar with concepts are encouraged to read the complete proposal [3]. This document provides wording for changes to the core language. Changes to the standard library are discussed in separate documents:

— Concepts for the C++0x Standard Library: Approach [N2036=06-0106]

— Concepts for the C++0x Standard Library: Introduction [N2037=06-0107]

— Concepts for the C++0x Standard Library: Utilities (Revision 1) [N2038=06-0108]

— Concepts for the C++0x Standard Library: Containers [N2085=06-0155]

— Concepts for the C++0x Standard Library: Iterators (Revision 1) [N2083=06-0153]

— Concepts for the C++0x Standard Library: Algorithms (Revision 1) [N2084=06-0154]

— Concepts for the C++0x Standard Library: Numerics [N2041=06-0111]

Changes from N2081

The wording in this document reflects several changes to the formulation of concepts presented in N2081, most of which were discussed at the ad hoc concepts meeting in February, 2007 at Google. The following changes are reflected in this wording:

— Keywords: the `where` keyword has been replaced with `requires`, due to the prevalence of `where` as an identifier in existing C++ code [1]. Thus, the "where clause" is now a "requires clause" or "requirements clause."

— Terminology: we now collectively refer to the members of concepts as "associated" members, e.g., "associated types," "associated functions," "associated requirements," etc.

— Signatures: we now de-emphasize the existence of "forwarding functions" in the implementation of signatures, because such forwarding functions should not exist in a quality implementation (for performance reasons). We have renamed "signatures" to "associated functions".

— Type checking: it is now ill-formed to call an unconstrained template from a constrained template.

— Late checking: the `late_check` keyword now applies to entire templates, and is part of the declaration.

— Inline requirements: inline requirements are now called the "simple form" of requirements, while requirements stated in the requires clause are called the "general form" of requirements. The two are now semantically equivalent (including having the same rules for name lookup).

— Requirements: "||" constraints have been moved to a separate proposal [5], due to lack of experience in their implementation and use [2]. As part of this change, requirements clauses are now comma-separated lists rather than conjunctions.

— Associated values: This feature has been removed. Instead, one can use the `constexpr` facility [4] with signatures.

Typographical conventions

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.

The wording in this document is based on the latest C++0x draft, currently N2135. We have done our best to synchronize section, paragraph, and table numbers with N2135. However, we have omitted many sections (those that did not require any changes), leaving some dangling references in the final document. These references will be resolved automatically when the LaTeX source of this proposal is merged with the LaTeX source of the working paper.

Chapter 2 Lexical conventions [lex]

2.11 Keywords [key]

1 The identifiers shown in Table 3 are reserved for use as keywords (that is, they are unconditionally treated as keywords in phase 7):

Table 3: keywords

asm	default	goto	reinterpret_cast	true
auto	delete	if	requires	try
axiom	do	inline	return	typedef
bool	double	int	short	typeid
break	dynamic_cast	late_check	signed	typename
case	else	long	sizeof	union
catch	enum	mutable	static	unsigned
char	explicit	namespace	static_assert	using
class	export	new	static_cast	virtual
concept	extern	operator	struct	void
concept_map	false	private	switch	volatile
const	float	protected	template	wchar_t
const_cast	for	public	this	while
continue	friend	register	throw	

Chapter 3 Basic concepts [basic]

6 Some names denote types, classes, concepts, enumerations, or templates. In general, it is necessary to determine whether or not a name denotes one of these entities before parsing the program that contains it. The process that determines this is called *name lookup* (3.4).

3.2 One definition rule [basic.def.odr]

1 No translation unit shall contain more than one definition of any variable, function, class type, concept, concept map, enumeration type or template.

5 There can be more than one definition of a class type (clause 9), concept (clause 7.6), concept map (clause 7.6.2), enumeration type (**??**), inline function with external linkage (**??**), class template (clause 14), non-static function template (14.5.5), static data member of a class template (**??**), member function of a class template (**??**), or template specialization for which some template parameters are not specified (14.7, 14.5.4) in a program provided that each definition appears in a different translation unit, and provided the definitions satisfy the following requirements. Given such an entity named `D` defined in more than one translation unit, then

3.3 Declarative regions and scopes [basic.scope]

3.3.1 Point of declaration [basic.scope.pdecl]

10 The point of declaration for a concept (clause 7.6) is immediately after the identifier in the *concept*. The point of declaration for a concept map (clause **??**) is immediately after the *template-id* in the *concept-map*.

Add the following new sections to 3.3 [basic.scope]:

3.3.7 Concept scope [basic.scope.concept]

1 The following rules describe the scope of names declared in concepts and concept maps.

1) The potential scope of a name declared in a concept or concept map consists not only of the declarative region following the name's point of declaration, but also of all associated function definitions in that concept or concept map.

2) If reordering declarations in a concept or concept map yields an alternate valid program under (1), the program is ill-formed, no diagnostic is required.

3) A name declared within an associated function definition hides a declaration of the same name whose scope extends to or past the end of the associated function's concept or concept map.

4) The potential scope of a declaration that extends to or past the end of a concept or concept map definition also extends to the regions defined by its associated function definitions, even if the associated functions are defined

lexically outside the concept or concept map.

2 The name of a concept member shall only be used as follows:

— in the scope of its concept (as described above) or a concept refining (clause 7.6.3) from its concept,

— after the `::` scope resolution operator (5.2) applied to the name of a concept map or a *template-parameter* in a constrained template (14.9).

3.3.8 Requirements scope [basic.scope.req]

1 In a template that contains a requirements clause (14.9.1), the names of all associated functions inside the concepts named by the *concept-id requirements* in the requirements clause (14.9.1) are declared in the scope of the template declaration. [*Example:*

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

concept RAIterator<typename Iter> {
  Integral difference_type;
  difference_type operator-(Iter, Iter);
}

template<RAIterator Iter>
RAIterator<Iter>::difference_type distance(Iter first, Iter last) {
  return -(first - last); // okay: name lookup for - finds RAIterator<Iter>::operator-
                          // and Integral<RAIterator<Iter>::difference_type>::operator-
                          // overload resolution picks the appropriate operator for both uses of -
}
```

— end example]

3.3.9 Name hiding [basic.scope.hiding]

Add the following new paragraph:

6 In an associated function definition, the declaration of a local name hides the declaration of a member of the concept or concept map with the same name; see 3.3.7.

3.4 Name lookup [basic.lookup]

1 The name lookup rules apply uniformly to all names (including *typedef-names* (**??**), *namespace-names* (**??**), *concept-names* (7.6) and *class-names* (**??**)) wherever the grammar allows such names in the context discussed by a particular rule. Name lookup associates the use of a name with a declaration (**??**) of that name. Name lookup shall find an unambiguous declaration for the name (see **??**). Name lookup may associate more than one declaration with a name if it finds the name to be a function name; the declarations are said to form a set of overloaded functions (**??**). Overload resolution (**??**) takes place after name lookup has succeeded. The access rules (clause **??**) are considered only once name lookup and function overload resolution (if applicable) have succeeded. Only after name lookup, function overload resolution

(if applicable) and access checking have succeeded are the attributes introduced by the name's declaration used further in expression processing (clause 5).

3.4.1 Unqualified name lookup [basic.lookup.unqual]

Add the following new paragraphs:

16 A name used in the definition of a concept or concept map `X` outside of an associated function body shall be declared in one of the following ways:

— before its use in the concept or concept map `X` or be a member of a refined concept of `X`, or

— if `X` is a member of namespace `N`, before the definition of concept or concept map `X` in namespace `N` or in one of `N` 's enclosing namespaces.

17 A name used in the definition of an associated function (7.6.1.1) of a concept or concept map `X` following the associated function's *declarator-id* shall be declared in one of the following ways:

— before its use in the block in which it is used or in an enclosing block (**??**), or

— shall be a member of concept or concept map `X` or be a member of a refined concept of `X`, or

— if `X` is a member of namespace `N`, before the associated function definition, in namespace `N` or in one of `N` 's enclosing namespaces.

3.4.3 Qualified name lookup [basic.lookup.qual]

1 The name of a class, concept map or namespace member can be referred to after the `::` scope resolution operator (5.2) applied to a *nested-name-specifier* that nominates its class, concept map or namespace. During the lookup for a name preceding the `::` scope resolution operator, object, function, and enumerator names are ignored. If the name found is not a *class-name* (clause 9), *concept-id* or *namespace-name* (**??**), the program is ill-formed.

Add the following paragraph to Qualified name lookup [basic.lookup.qual]

6 In a constrained template (14.9), the name of an associated type (7.6.1.2) can be referred to after the `::` scope resolution operator (5.2) applied to a *typedef-name* introduced for a *type-parameter* (14.1). Qualified name lookup searches for the associated type within each concept named by a *concept-id requirement* in the requirements clause (14.9.1). If qualified name lookup finds more than one such associated type, and the associated types are not equivalent (14.4), the lookup is ambiguous. [*Example:*

```
concept Callable1<typename F, typename T1> {
  typename result_type;
  result_type operator()(F&, T1);
}

template<typename F, typename T1>
requires Callable1<F, T1>
F::result_type      // okay: refers to Callable<F, T1>::result_type
forward(F& f, const T1& t1) {
  return f(t1);
}
```

— *end example*]

Add the following subsection to Qualified name lookup [basic.lookup.qual]

3.4.3.3 Concept map members [concept.qual]

1 If the *nested-name-specifier* of a *qualified-id* nominates a *concept-id*, the name specified after the *nested-name-specifier* is looked up in the scope of the concept map (**??**), except for the cases listed below. The name shall represent one or more members of that concept map. [*Note:* a concept map member can be referred to using a *qualified-id* at any point in its potential scope (3.3.7). — *end note*]

2 A concept map member name hidden by a name in a nested declarative region can still be found if qualified by the name of its concept map followed by the `::` operator.

Chapter 5 Expressions [expr]

5.1 Primary expressions [expr.prim]

7 An *identifier* is an *id-expression* provided it has been suitably declared (clause 7). [*Note:* for *operator-function-id*s, see **??**; for *conversion-function-id*s, see **??**; for *template-id*s, see **??**. A *class-name* prefixed by ~ denotes a destructor; see 12.4. Within the definition of a non-static member function, an *identifier* that names a non-static member is transformed to a class member access expression (**??**). — *end note*] The type of the expression is the type of the *identifier*. The result is the entity denoted by the identifier. The result is an lvalue if the entity is a function, variable, or data member.

qualified-id:
  `::`$_{opt}$ *nested-name-specifier* `template`$_{opt}$ *unqualified-id*
  `::` *identifier*
  `::` *operator-function-id*
  `::` *template-id*
  `::` *concept-id*

nested-name-specifier:
  type-name `::`
  namespace-name `::`
  nested-name-specifier identifier `::`
  nested-name-specifier `template`$_{opt}$ *template-id* `::`

A *nested-name-specifier* that names a class, optionally followed by the keyword `template` (**??**), and then followed by the name of a member of either that class (9.2) or one of its base classes (clause **??**), is a *qualified-id*; **??** describes name lookup for class members that appear in *qualified-ids*. The result is the member. The type of the result is the type of the member. The result is an lvalue if the member is a static member function or a data member. [*Note:* a class member can be referred to using a *qualified-id* at any point in its potential scope (**??**). — *end note*] The result of a *qualified-id* for which name lookup finds a member of a concept map (3.4.3.3) is an rvalue. Where *class-name* `::` *class-name* is used, and the two *class-name*s refer to the same class, this notation names the constructor (12.1). Where *class-name* `::~` *class-name* is used, the two *class-name*s shall refer to the same class; this notation names the destructor (12.4). [*Note:* a *typedef-name* that names a class is a *class-name* (**??**). — *end note*]

5.2 Primary expressions [expr.prim]

5.2.2 Function call [expr.call]

Add the following new paragraph to [expr.call]:

11 In a constrained template (14.9), a function call shall not call an unconstrained template.

5.3 Unary expressions [expr.unary]

5.3.1 Unary operators [expr.unary.op]

2 The result of the unary & operator is a pointer to its operand. The operand shall be an lvalue or a *qualified-id*. The operand shall not refer to a member of a concept or concept map. In the first case, if the type of the expression is “T,” the type of the result is “pointer to T.” In particular, the address of an object of type “*cv* T” is “pointer to *cv* T,” with the same cv-qualifiers. For a *qualified-id*, if the member is a static member of type “T”, the type of the result is plain “pointer to T.” If the member is a non-static member of class C of type T, the type of the result is “pointer to member of class C of type T.” [*Example:*

```
struct A { int i; };
struct B : A { };
... &B::i ...                  // has type int A::*
```

— end example] [*Note:* a pointer to member formed from a mutable non-static data member (**??**) does not reflect the mutable specifier associated with the non-static data member. *— end note*]

Chapter 7 Declarations [dcl.dcl]

1 Declarations specify how names are to be interpreted. Declarations have the form

declaration-seq:
  declaration
  declaration-seq declaration

declaration:
  block-declaration
  function-definition
  template-declaration
  explicit-instantiation
  explicit-specialization
  linkage-specification
  namespace-definition
  concept-definition
  concept-map-definition

block-declaration:
  simple-declaration
  asm-definition
  namespace-alias-definition
  using-declaration
  using-directive
  static_assert-declaration

simple-declaration:
  decl-specifier-seq$_{opt}$ init-declarator-list$_{opt}$;

static_assert-declaration:
  `static_assert` (*constant-expression* , *string-literal*) ;

[*Note: asm-definition*s are described in **??**, and *linkage-specification*s are described in **??**. *Function-definition*s are described in **??** and *template-declaration*s are described in clause 14. *Namespace-definition*s are described in **??**, *Concept-definition*s are described in 7.6.1, *Concept-map-definition*s are described in 7.6.2, *using-declaration*s are described in **??** and *using-directive*s are described in **??**. — *end note*] The *simple-declaration*

decl-specifier-seq$_{opt}$ init-declarator-list$_{opt}$;

is divided into two parts: *decl-specifier*s, the components of a *decl-specifier-seq*, are described in **??** and *declarator*s, the components of an *init-declarator-list*, are described in clause 8.

2 A declaration occurs in a scope (3.3); the scope rules are summarized in 3.4. A declaration that declares a function or defines a class, concept, namespace, template, or function also has one or more scopes nested within it. These nested

scopes, in turn, can have declarations nested within them. Unless otherwise stated, utterances in clause 7 about components in, of, or contained by a declaration or subcomponent thereof refer only to those components of the declaration that are *not* nested within scopes nested within the declaration.

Add the following new section to 7 [dcl.dcl]:

7.6 Concepts [concept]

1 Concepts describe an abstract interface that can be used to constrain templates 14.9. Concepts state certain syntactic and semantic requirements (7.6.1) on a set of template type, non-type, and template parameters.

> *concept-id:*
> *concept-name* < *template-argument-list*$_{opt}$ >
>
> *concept-name:*
> *identifier*

2 A *concept-id* refers to a specific concept map (7.6.2) by its *concept-name* and a specific set of concept arguments. [*Example:* `CopyConstructible<int>` is a *concept-id* if name lookup (3.4) determines that the identifier `CopyConstructible` refers to a *concept-name*. — *end example*]

7.6.1 Concept definitions [concept.def]

1 The grammar for a *concept-definition* is:

> *concept-definition:*
> `auto`$_{opt}$ `concept` *identifier* < *template-parameter-list* > *refinement-clause*$_{opt}$ *concept-body* `;`$_{opt}$

2 *Concept-definition*s are used to make *concept-name*s. A *concept-name* is inserted into the scope in which it is declared immediately after the *concept-name* is seen. A concept is considered defined after the closing brace of its *concept-body* has been seen.

3 Concepts shall only be defined at namespace or global scope.

4 A concept with a preceding `auto` is an *implicit concept* (7.6.4). A concept without a preceding `auto` is an *explicit concept*.

5 The *template-parameter-list* of a *concept-definition* shall not contain any requirements specified in the simple form (14.9.1).

6

> *concept-body:*
> `{` *concept-member-specification*$_{opt}$ `}`
>
> *concept-member-specification:*
> *associated-function concept-member-specification*$_{opt}$
> *associated-parameter concept-member-specification*$_{opt}$
> *associated-requirements concept-member-specification*$_{opt}$
> *axiom-definition concept-member-specification*$_{opt}$

The body of a concept contains associated functions (7.6.1.1), associated parameters (7.6.1.2), associated requirements

(7.6.1.3), and axioms (7.6.1.3) that describe the behavior of the concept parameters in its *template-parameter-list*.

7.6.1.1 Associated functions [concept.fct]

1 Associated functions describe functions, member functions, or operators that describe the functional behavior of the concept arguments and associated arguments (7.6.1.2). Concept maps (7.6.2) for a given concept must provide, either implicitly (7.6.2.3) or explicitly (7.6.2.1, definitions for each associated function in the concept.

associated-function:
simple-declaration
function-definition
template-declaration

2 Associated functions can specify requirements for free functions and operators. [*Example:*

```
concept Monoid<typename T> {
  T operator+(T, T);
  T identity();
}
```

— end example]

3 With the exception of the assignment operator (**??**), associated functions shall specify requirements for operators as free functions. [*Note:* This restriction applies even to the operators `()`, `[]`, and `->`, which can otherwise only be overloaded via non-static member functions (**??**): [*Example:*

```
concept Convertible<typename T, typename U> {
  operator U(T); // okay: conversion from T to U
  T::operator U*() const; // error: cannot specify requirement for member operator
}
```

— end example]

4 Associated functions can specify requirements for non-static member functions, constructors and destructors. [*Example:*

```
concept Container<typename X> {
  X::X(int n);
  X::~X();
  bool X::empty() const;
}
```

— end example]

5 Associated functions can specify requirements for function templates and member function templates. [*Example:*

```
concept Sequence<typename X> {
  typename value_type;

  template<InputIterator Iter>
    requires Convertible<InputIterator<Iter>::value_type, Sequence<X>::value_type>
    X::X(Iter first, Iter last);
};
```

— *end example*]

6 Concepts may contain overloaded associated functions, but the associated functions shall have distinct types. [*Example:*

```
concept C<typename X> {
  void f(X);
  void f(X, X); // okay
  void f(X); // error: redeclaration of function 'f'
  int f(X, X); // error: differs only by return type
};
```

— *end example*]

7 All associated function arguments are passed by reference. The actual type `P` of a type parameter whose declared type is `T` is `T`, if `T` is a reference, or `T const&`, if `T` is not a reference. [*Example:*

```
concept C<typename X> {
  void f(X);
};

struct Y {};
concept_map C<Y> {
  void f(const Y&); // okay: matches requirement for f
};

concept_map C<Y&&> {
  void f(Y&&); // okay: matches requirement for f
};
```

— *end example*]

8 Associated non-member functions may have a default implementation. This implementation will be instantiated when implicit definition of an implementation (7.6.4) for the associated function (7.6.2.1) fails. [*Example:*

```
concept EqualityComparable<typename T> {
  bool operator==(T, T);
  default bool operator!=(T x, T y) { return !(x == y); }
};

class X{};
bool operator==(const X&, const X&);

concept_map EqualityComparable<X> { }; // okay, operator!= uses default
```

— *end example*]

7.6.1.2 Associated parameters [concept.param]

1 Associated parameters are types, values, and templates that are "implicit" parameters defined in the concept body and used in the description of the concept.

associated-parameter:
type-parameter ;

2 An associated type parameter (or *associated type*) is an associated parameter that specifies a type in a concept body. Associated types are typically used to express the parameter and return types of associated functions. [*Example:*

```
concept Callable1<typename F, typename T1> {
  typename result_type;
  result_type operator()(F, T1);
}
```

— end example]

3 Associated parameters may be provided with a default value. The default value will be used to define the associated parameter when no corresponding definiion is provided in a concept map (7.6.2.2). [*Example:*

```
concept Iterator<typename Iter> {
  typename difference_type = int;
}

concept_map Iterator<int*> { } // okay, difference_type is int
```

— end example] (**??**).

4 Associated parameters may use the simple form to specify requirements (14.9.1) on the associated parameters. The simple form is equivalent to a declaration of the associated parameter followed by an associated requirement (7.6.1.3) stated using the general form (14.9.1). [*Example:*

```
concept InputIterator<typename Iter> { /* ... */ }

concept Container<typename X> {
  InputIterator iterator;
  // ...
}
```

— end example]

7.6.1.3 Associated requirements [concept.req]

1 Associated requirements place additional requirements on concept parameters and associated parameters. Associated requirements have the same form and behavior as a requirements clause for constrained templates (14.9).

associated-requirements:
requires-clause ;

[*Example:*

```
concept Iterator<typename Iter> {
  typename difference_type;
  requires SignedIntegral<difference_type>;
}
```

— *end example*]

7.6.1.4 Axioms [concept.axiom]

1 Axioms allow the expression of the semantic properties of concepts.

axiom-definition:
  `axiom` *identifier* `(` *parameter-declaration-clause* `)` *axiom-body*

axiom-body:
  `{` *axiom-seq*$_{opt}$ `}`

axiom-seq:
  axiom axiom-seq$_{opt}$

axiom:
  expression-statement
  `if (` *condition* `)` *expression-statement*

An *axiom-definition* defines a new semantic axiom whose name is specified by its *identifier*. [*Example:*

```
concept Semigroup<typename Op, typename T> {
  T operator()(Op, T, T);

  axiom Associativity(Op op, T x, T y, T z) {
    op(x, op(y, z)) == op(op(x, y), z);
  }
}

concept Monoid<typename Op, typename T> : Semigroup<Op, T> {
  T identity_element(Op);

  axiom Identity(Op, T x) {
    op(x, identity_element(op)) == x;
    op(identity_element(op), x) == x;
  }
}
```

— *end example*]

2 Within the body of an *axiom-definition*, equality (`==`) and inequality (`!=`) operators are available for each concept type parameter and associated type `T`. These implicitly-defined operators have the form:

```
bool operator==(const T&, const T&);
bool operator!=(const T&, const T&);
```

[*Example:*

```
concept CopyConstructible<typename T> {
  T::T(const T&);

  axiom CopyEquivalence(T x) {
    T(x) == x; // okay, uses implicit ==
```

```
  }
}
```

— *end example*]

3 Name lookup within an axiom will only find the implicitly-declared == and != operators if the corresponding operation is not declared as an associated function (7.6.1.1) in the concept, one of the concepts it refines (7.6.3), or in an associated requirement (7.6.1.3). [*Example:*

```
concept EqualityComparable<typename T> {
  bool operator==(T, T);
  bool operator!=(T, T);

  axiom Reflexivity(T x) {
    x == x; // okay: refers to EqualityComparable<T>::operator==
  }
}
```

— *end example*] The != operator is semantically equivalent to the logical negation of the == operator, whether the == and != operators are explicitly or implicitly defined.

4 Where axioms state the equality of two expressions, implementations are permitted to replace one expression with the other. [*Example:*

```
concept Monoid<typename Op, typename T> {
  T identity_element(Op);

  axiom Identity(Op, T x) {
    op(x, identity_element(op)) == x;
    op(identity_element(op), x) == x;
  }
}

template<typename Op, typename T> requires Monoid<Op, T>
  T identity(const Op& op, const T& t) {
    return op(t, identity_element(op)); // equivalent to "return t;"
  }
```

— *end example*]

5 Axioms can state conditional semantics using `if` statements. When the condition can be proven true, and the *expression-statement* states the equality of two expressions, implementations are permitted to replace one expression with the other. [*Example:*

```
concept TotalOrder<typename Op, typename T> {
  bool operator()(Op, T, T);

  axiom Reflexivity(Op op, T x) { op(x, x); }
  axiom Antisymmetry(Op op, T x, T y) { if (op(x, y) && op(y, x)) x == y; }
  axiom Transitivity(Op op, T x, T y, T z) { if (op(x, y) && op(y, z)) op(x, z) == true; }
}
```

— *end example*]

7.6.2 Concept maps [concept.map]

1 The grammar for a *concept-map-definition* is:

> *concept-map-definition:*
> `concept_map` *concept-id* { *concept-map-member-specification*$_{opt}$ } `;`$_{opt}$
>
> *concept-map-member-specification:*
> *simple-declaration concept-map-member-specification*$_{opt}$
> *function-definition concept-map-member-specification*$_{opt}$
> *template-declaration concept-map-member-specification*$_{opt}$

2

Concept maps describe how a set of template arguments meet the requirements stated in the body of a concept definition (7.6.1). Whenever a constrained template (14.9) is named, there must be a concept map corresponding to each *concept-id* requirement in the requirements clause (14.9.1). This concept map may be written explicitly (7.6.2), instantiated from a concept map template (14.5.6), or generated implicitly (7.6.4). [*Example:*

```
class student_record {
public:
  string id;
  string name;
  string address;
};

concept EqualityComparable<typename T> {
  bool operator==(T, T);
}

concept_map EqualityComparable<student_record> {
  bool operator==(const student_record& a, const student_record& b) {
    return a.id == b.id;
  }
};

template<typename T> requires EqualityComparable<T> void f(T);

f(student_record()); // okay, have concept_map EqualityComparable<student_record>
```

— *end example*]

3 Concept maps shall provde, either implicitly (7.6.2.3) or explicitly (7.6.2.1, 7.6.2.2), definitions for every associated function (7.6.1.1) and associated parameter (7.6.1.2) of the concept named by its *concept-id* and any of its refined concepts (7.6.3). [*Example:*

```
concept C<typename T> { T f(T); }

concept_map C<int> {
  int f(int); // okay: matches requirement for f in concept C
}
```

— end example]

4 Concept maps shall be defined in the same namespace as their corresponding concept.

5 Concept maps shall not contain declarations that do not match any requirement in their corresponding concept or its refined concepts. [*Example:*

```
concept C<typename T> { }

concept_map C<int> {
  int f(int); // error: no requirement for function f
}
```

— end example]

6 At the point of definition of a concept map, all associated requirements (7.6.1.3) of the corresponding concept and its refined concepts (7.6.3) shall be satisfied. [*Example:*

```
concept SignedIntegral<typename T> { /* ... */ }

concept ForwardIterator<typename Iter> {
  typename difference_type;
  requires SignedIntegral<difference_type>;
}

concept_map SignedIntegral<ptrdiff_t> { };

concept_map ForwardIterator<int*> {
  typedef ptrdiff_t difference_type;
} // okay: there exists a concept_map SignedIntegral<ptrdiff_t>

concept_map ForwardIterator<file_iterator> {
  typedef long difference_type;
} // error: no concept_map SignedIntegral<long>
```

— end example]

7 The definition of a concept map asserts that the axioms (7.6.1.4) defined in the corresponding concept and its refined concepts (7.6.3) are true. The implementation is permitted to assume that these axioms hold without verifying them [*Note:* the intent is to permit axioms to be used for optimization *— end note*].

8 If a concept map is provided for a particular *concept-id*, then that concept map shall be defined before the corresponding *concept-id* is required.

7.6.2.1 Associated function definitions [concept.map.fct]

1 Associated non-member function requirements (7.6.1.1) are satisfied by function definitions in the body of a concept map. These definitions can be used to adapt the syntax of the concept arguments to the syntax expected by the concept. [*Example:*

```
concept Stack<typename S> {
  typename value_type;
  bool empty(S);
```

```
  void push(S&, value_type);
  void pop(S&);
  value_type& top(S&);
}

// Make a vector behave like a stack
template<Regular T>
concept_map Stack<std::vector<T> > {
  typedef T value_type;
  bool empty(std::vector<T> vec) { return vec.empty(); }
  void push(std::vector<T>& vec, value_type value) { vec.push_back(value); }
  void pop(std::vector<T>& vec) { vec.pop_back(); }
  value_type& top(std::vector<T>& vec) { return vec.back(); }
}
```

— end example]

2 A function declaration in a concept map matches an associated function of the same name when function type is equivalent after substitution of concept arguments and transformation of parameter types to references (7.6.1.1). [*Example:*

```
concept C<typename X> {
  void f(const X&);
};

concept_map C<int> {
  void f(int) { } // okay, "int" parameter becomes a "const int&" parameter
};
```

— end example]

3 Functions declared within a concept map may be defined outside the concept map. [*Example:*

```
// c.h
concept C<typename X> {
  void f(X);
};

concept_map C<int> {
  void f(int);
};

// c.cpp
void C<int>::f(int) {
  // ...
}
```

— end example]

4 Function templates declared within a concept map match an associated function template when the associated function template is at least as specialized (14.5.5.2) as the function template. [*Example:*

```
concept C<typename X> {
  typename value_type;

  template<ForwardIterator Iter>
    requires Convertible<Iter::value_type, value_type>
    void X::X(Iter first, Iter last);
}

concept_map C<MyContainer> {
  typedef int value_type;

  template<InputIterator Iter>
    requires Convertible<Iter::value_type, int>
    void X::X(Iter first, Iter last) { ... } // okay: associated function is more specialized
}
```

— end example]

5 Associated member function requirements (7.6.1.1), including constructors and destructors, are satisfied by member functions in the corresponding concept map parameter (call it `X`). Let `parm1`, `parm2`, ..., `parmN` be the parameters of the associated member function, then

— if the associated member function requirement is a constructor requirement, the requirement is satisfied if `X` is initializable with arguments `parm1`, `parm2`, ..., `parm`*N*, [*Example:*

```
concept IntConstructible<typename T> {
  T::T(int);
}

concept_map IntConstructible<float> { } // okay: float can be initialized with an int

struct X { X(long); }
concept_map IntConstructible<X> { } // okay: X has a constructor that can accept an int (converted to a long)
```

— end example]

— if the associated member function requirement is a destructor requirement, the requirement is satisfied if `X` is a built-in type or has a public destructor, [*Example:*

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

concept_map Destructible<int> { } // okay: int is a built-in type

struct X { };
concept_map Destructible<X> { } // okay: X has implicitly-declared, public destructor

struct Y { private: ~Y(); }
concept_map Destructible<Y> { } // error: Y's destructor is inaccessible
```

— *end example*]

— otherwise, the associated member function requirement requires a member function `f`. If `x` is an lvalue of type *cv* `X`, where *cv* are the cv-qualifiers on the associated member function requirement, the requirement is satisfied if the expression `x.f(parm1, parm2, ..., parmN)` is well-formed and its type is implicitly convertible to the return type of the associated member function requirement. [*Example:*

```
concept MemberSwap<typename T> {
  void T::swap(T&);
}

struct X {
  X& swap(X&);
}
concept_map MemberSwap<X> { } // okay: X has a member function swap and its return type is convertible to void
```

— *end example*]

7.6.2.2 Associated parameter definitions [concept.map.associated]

1 Definitions in the concept map provide types, values, and templates that satisfy requirements for associated parameters (7.6.1.2).

2 Associated type parameter requirements are satisfied by type definitions in teh body of a concept map. [*Example:*

```
concept ForwardIterator<typename Iter> {
  typename difference_type;
}

concept_map ForwardIterator<int*> {
  typedef ptrdiff_t difference_type;
}
```

— *end example*]

3 Associated template parameter requirements are satisfied by class template definitions in the body of the concept map. [*Example:*

```
concept Allocator<typename Alloc> {
  template<class T> class rebind;
}

template<typename T>
concept_map Allocator<my_allocator<T>> {
  template<class U>
    class rebind {
    public:
      typedef my_allocator<U> type;
    };
};
```

— *end example*]

7.6.2.3 Implicit definitions [concept.map.implicit]

1 Any of the requirements of a concept and its refined concepts (7.6.3) that are not satisfied by the definitions in the body of a concept map (7.6.2.1, 7.6.2.2) are *unsatisfied requirements*.

2 Definitions for unsatisfied requirements in a concept map are implicitly defined from the requirements and their default values as specified by the matching of implicit concepts (7.6.4). If any unsatisfied requirement is not matched by this process, the concept map is ill-formed.

7.6.3 Concept refinement [concept.refinement]

1 The grammar for a *refinement-clause* is:

refinement-clause:
`:` *refinement-specifier-list*

refinement-specifier-list:
refinement-specifier **,** *refinement-specifier-list*

refinement-specifier:
`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-id*

2 Refinements specify an inheritance relationship among concepts. Concept refinement inherits all requirements in the body of a concept (7.6.1, such that the requirements of the refining concept are a superset of the requirements of the refined concept. When a concept `B` refines a concept `A`, every set of template arguments that meets the requirements of `B` also meets the requirements of `A`. `A` is called the refined concept and `B` is the refining concept. [*Example:* In the following example, `EquilateralPolygon` refines `Polygon`. Thus, every `EquilateralPolygon` is a `Polygon`, and constrained templates (14.9) that are well-formed with a `Polygon` constraint are well-formed when given a `EquilateralPolygon`.

```
concept Polygon<typename P> { /* ... */ }

concept EquilateralPolygon<typename P> : Polygon<P> { /* ... */ }
```

— *end example*]

3 The *concept-id*s referred to in the refinement clause shall correspond to defined concepts. [*Example:*

```
concept C<typename T> : C<vector<T>> {/* ... */ } // error: concept C is not defined
```

— *end example*]

4 A *concept-id* in the refinement clause shall not refer to associated types.

5 A *concept-id* in the refinement clause shall refer to at least one of the concept parameters. [*Example:*

```
concept InputIterator<typename Iter>
  : Incrementable<int>   // error: Incrementable<int> uses no concept parameters
{
  // ...
}
```

— *end example*]

7.6.3.1 Concept member lookup [concept.member.lookup]

1 Concept member lookup determines the meaning of a name (*id-expression*) in concept scope (3.3.7). The following steps define the result of name lookup for a member name `f` in concept scope `C`. $\mathtt{C}_R$ is the set of concept scopes corresponding to the concepts refined by the concept whose scope is `C`.

2 If the name `f` is declared in concept scope `C`, and `f` refers to an associated parameter (7.6.1.2), then the result of name lookup is the associated parameter.

3 If the name `f` is declared in concept scope `C`, and `f` refers to one or more associated functions (7.6.1.1), then the result of name lookup is an overload set containing the associated functions in `C` in addition to the overload sets in each concept scope in $\mathtt{C}_R$ for which name lookup of `f` results in an overload set. [*Example:*

```
concept C<typename T> {
  T f(T); // #1
}

concept D<typename T> {
  T f(T, T); // #2
}

template<typename T>
requires D<T>
void f(T x)
{
  D<T>::f(x); // name lookup finds #1 and #2, overload resolution selects #1
}
```

— *end example*]

4 If the name `f` is not declared in `C`, name lookup searches for `f` in the scopes of each of the refined concepts ($\mathtt{C}_R$). If name lookup of `f` is ambiguous in any concept scope $\mathtt{C}_R$, name lookup of `f` in `C` is ambiguous. Otherwise, the set of concept scopes $\mathtt{C}_{R'}$ is a subset of $\mathtt{C}_R$ containing only those concept scopes for which name lookup finds `f`. The result of name lookup for `f` in `C` is defined by:

— if $\mathtt{C}_{R'}$ is empty, name lookup of `f` in `C` returns no result, or

— if $\mathtt{C}_{R'}$ contains only a single concept scope, name lookup for `f` on `C` is the result of name lookup for `f` in $\mathtt{C}_{R'}$, or

— if `f` refers to an overload set in all concept scopes in $\mathtt{C}_{R'}$, then `f` refers to an overload set containing all associated functions from each of these overload sets, or

— if `f` refers to an associated type in all concept scopes in $\mathtt{C}_{R'}$, and all of the associated types are equivalent (14.9.1), the result is the associated type `f` found first by a depth-first traversal of the refinement clause,

— otherwise, name lookup of `f` in `C` is ambiguous.

5 When name lookup in a concept scope `C` results in an overload set, duplicate associated functions are removed from the overload set. If more than one associated function in the overload set has the same signature (**??**), the associated function found first by a depth-first traversal of the refinements of `C` starting at `C` will be retained and the other associated functions will be removed as duplicates. [*Example:*

```
concept A<typename T> {
  T f(T); // #1a
}

concept B<typename T> {
  T f(T); // #1b
  T g(T); // #2a
}

concept C<typename T> : A<T>, B<T> {
  T g(T); // #2b
}

template<typename T>
requires C<T>
void h(T x) {
  C<T>::f(x); // overload set contains #1a; #1b was removed as a duplicate
  C<T>::g(x); // overload set contains #2b; #2a was removed as a duplicate
}
```

— end example]

7.6.3.2 Implicit concept maps for refined concepts [concept.implicit.maps]

1 When a concept map is defined for a concept `C` that has a refinement clause, concept maps for each of the concepts refined by `C` are implicitly defined. [*Example:*

```
concept A<typename T> { }
concept B<typename T> : A<T> { }

concept_map B<int> { } // implicitly defines concept map A<int>
```

— end example]

2 When a concept map is implicitly defined for a refined concept, definitions in the concept map can be used to satisfy the requirements of the refined concept. [*Example:*

```
concept C<typename T> {
  T f(T);
}

concept D<typename T> : C<T> { }

concept_map D<int> {
  int f(int x); // satisfies requirement for C<int>::f
}
```

— end example]

3 Concept map templates (14.5.6) are implicitly defined only for refinements for which the template parameters of the original concept map are deducible from the refinement. Concept maps for which the template parameters of the original concept map are not all deducible shall have been defined either implicitly or explicitly, and associated functions and parameters for these refined concepts shall not be defined in the original concept map. [*Example:*

```
concept Ring<typename AddOp, typename MulOp, typename T>
  : Group<AddOp, T>, Monoid<MulOp, T> { /* ... */ }

template<Integral T>
concept_map Ring<std::plus<T>, std::multiplies<T>, T> { }
// okay, implicitly generates:
template<Integral T> concept_map Group<std::plus<T>, T> { }
template<Integral T> concept_map Monoid<std::multiplies<T>, T> { }

template<Integral T, Integral V>
  requires MutuallyConvertible<T, V>,
  concept_map Group<std::plus<T>, V> { }
// okay, used to instead of implicitly-generated Group refinement in the following concept map

template<Integral T, Integral U, Integral V>
  requires MutuallyConvertible<T, U>, MutuallyConvertible<T, V>,
           MutuallyConvertible<U, V>
  concept_map Ring<std::plus<T>, std::multiplies<U>, V> { }
// ill-formed, cannot implicitly define:
template<Integral T, Integral U, Integral V>
  requires MutuallyConvertible<T, U>, MutuallyConvertible<T, V>,
           MutuallyConvertible<U, V>
  concept_map Monoid<std::multiplies<U>, V> { }
```

— end example]

7.6.4 Implicit concepts [concept.implicit]

1 Concept maps for implicit concepts (i.e., those concepts preceded by the `auto` keyword) are implicitly defined when they are required to satisfy the requirements of a constrained template (14.9), the associated requirements of a concept (7.6.1.3), or a concept map of a refined concept that cannot be implicitly defined from the concept map for the refining concept (7.6.3.2). [*Example:*

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

template<typename T>
requires Addable<T>
T add(T x, T y) {
  return x + y;
}

int f(int x, int y) {
  return add(x, y); // okay: concept map Addable<int> implicitly defined
}
```

— *end example*]

2 The implicit definition of a concept map involves the implicit definition of concept map members for each associated non-member function (7.6.1.1) and associated parameter (7.6.1.2) requirement, described below. If the implicit definition of a concept map member would produce an invalid definition, or if any of the requirements of the concept would be unsatisfied by the implicitly-defined (7.6.2), the implicit definition of the concept map fails [*Note:* failure to implicitly define a concept map does not imply that the program is ill-formed. — *end note*] [*Example:*

```
auto concept F<typename T> {
  void f(T);
}

auto concept G<typename T> {
  void g(T);
}

template<typename T> requires F<T> void h(T); // #1
template<typename T> requires G<T> void h(T); // #2

struct X { };
void g(X);

void func(X x) {
  h(x); // okay: implicit concept map F<X> fails, causing template argument deduction to fail for #1; calls #2
}
```

— *end example*]

3 The implicit concept map member defined for an associated non-member function requirement (7.6.1.1) has the same type and name as the associated function, after the concept map parameters have been substituted into the associated function [*Note:* the implicitly-defined function matches the associated function requirement (7.6.2.1) — *end note*]İf the return type of the function is `void`, the body of the function contains a single *expression-statement*; otherwise, the body of the function contains a single `return` statement. The *expression* in the *expression-statement* or `return` statement is defined as follows:

- if the associated function requirement is a prefix unary operator `Op`, the *expression* is `Op parm`, where `parm` is the name of the parameter, or
- if the associated function requirement is a postfix unary operator `Op`, the *expression* is `parm Op`, where `parm` is the name of the parameter, or
- if the associated function requirement is a binary operator `Op`, the *expression* is `parm1 Op parm2`, where `parm1` and `parm2` are the first and second parameters, respectively, or
- if the associated function requirement is the function call operator, the *expression* is `parm1(parm2, parm3, ..., parm`*N*`)`, where `parm1` is the first parameter and `parm2` through `parm`*N* are the remaining parameters,
- otherwise, the associated function requirement is a function (call it `f`). The *expression* is an unqualified call (5.2.2) to `f` whose arguments are the parameters `parm1`, `parm2`, ..., `parm`*N*.

If the *expression* is ill-formed, and the associated non-member function requirement has a default implementation

(7.6.1.1), the implicit concept map member is defined by substituting the concept map arguments into the default implementation.

4 Implicitly-defined associated function definitions cannot have their addresses taken (5.3.1). It is unspecified whether these functions have linkage. [*Note:* Implementations are encouraged to optimize away implicit-defined associated function definitions, so that the use of constrained templates does not incur any overhead relative to unconstrained templates. *— end note*]

5 The implicit concept map member defined for an associated type parameter `type` can have its value deduced from the return type of an associated function requirement defined implicitly (7.6.4) or explicitly (7.6.2.1). If for each of the associated function requirements whose return type is `type`, the return type of the corresponding function in the concept is the same type `T`, then the implicit concept map member `type` is defined as `T`. [*Example:*

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

template<typename T> requires Addable<T>
Addable<T>::result_type f(T x, T y);

int g(int x, int y) {
  return f(x, y); // okay: Addable<int> implicitly defined
                  // implicitly-defined Addable<int>::operator+ calls built-in + for integers
                  // implicitly-defined Addable<int>::result_type is int
}
```

— end example]

6 If an associated parameter (7.6.1.2) has a default argument, a concept map member satisfying the associated parameter requirement shall be implicitly defined by substituting the concept map arguments into the default argument. If this substitution does not produce a valid type or template (14.8.2), the concept map member is not implicitly defined. [*Example:*

```
auto concept A<typename T> {
  typename result_type = typename T::result_type;
}

auto concept B<typename T> {
  T::T(const T&);
}

template<typename T> requires A<T> void f(const T&); // #1
template<typename T> requires B<T> void f(const T&); // #2

struct X {};
void g(X x) {
  f(x); // okay: A<X> cannot satisfy result_type requirement, and is not implicitly defined, calls #2
}
```

— end example]

Chapter 8 Declarators [dcl.decl]

8.3 Meaning of declarators [dcl.meaning]

8.3.6 Default arguments [dcl.fct.default]

Add the following new paragraph:

11 An associated function (7.6.1.1) or associated function definition (7.6.2.1) shall not have default arguments.

Chapter 9 Classes [class]

9.2 Class members [class.mem]

member-specification:
  member-declaration member-specification$_{opt}$
  access-specifier `:` *member-specification$_{opt}$*

member-declaration:
  decl-specifier-seq$_{opt}$ member-declarator-list$_{opt}$ `;`
  function-definition `;`$_{opt}$
  `::`$_{opt}$ *nested-name-specifier* `template`$_{opt}$ *unqualified-id* `;`
  using-declaration
  static_assert-declaration
  template-declaration
  requires-clause member-declaration

member-declarator-list:
  member-declarator
  member-declarator-list `,` *member-declarator*

member-declarator:
  declarator pure-specifier$_{opt}$
  declarator constant-initializer$_{opt}$
  identifier$_{opt}$ `:` *constant-expression*

pure-specifier:
  `= 0`

constant-initializer:
  `=` *constant-expression*

Add the following new paragraphs to 14 [temp]

19 A *member-declaration* that contains a *requires-clause* (14.9.1) is a constrained template (14.9) and shall only occur in a class template (14.5.1). A *member-declaration* with a *requires-clause* shall not be a friend declaration (**??**). A *member-declaration* shall have no more than one *requires-clause*.

Chapter 12 Special member functions [special]

12.1 Constructors [class.ctor]

5 A *default* constructor for a class `X` is a constructor of class `X` that can be called without an argument. If there is no user-declared constructor for class `X`, and if all of the non-static data members and base classes of `X` can be default-initialized (**??**). a default constructor is implicitly declared. An implicitly-declared default constructor is an `inline` `public` member of its class. A default constructor is *trivial* if it is implicitly-declared and if:

— its class has no virtual functions (**??**) and no virtual base classes (**??**), and

— all the direct base classes of its class have trivial default constructors, and

— for all the non-static data members of its class that are of class type (or array thereof), each such class has a trivial default constructor.

12.4 Destructors [class.dtor]

3 If a class has no user-declared destructor, and if every non-static data member of class type and every base class has an accessible destructor, a destructor is declared implicitly. An implicitly-declared destructor is an `inline` `public` member of its class. A destructor is *trivial* if it is implicitly-declared and if:

— all of the direct base classes of its class have trivial destructors and

— for all of the non-static data members of its class that are of class type (or array thereof), each such class has a trivial destructor.

12.8 Copying class objects [class.copy]

5 The implicitly-declared copy constructor for a class `X` will have the form

```
X::X(const X&)
```

if

— each direct or virtual base class `B` of `X` has a copy constructor whose first parameter is of type `const B&` or `const volatile B&`, and

— for all the non-static data members of `X` that are of a class type `M` (or array thereof), each such class type has a copy constructor whose first parameter is of type `const M&` or `const volatile M&`.[1]

[1] This implies that the reference parameter of the implicitly-declared copy constructor cannot bind to a `volatile` lvalue; see **??**.

Otherwise, the implicitly declared copy constructor will have the form

```
X::X(X&)
```

An implicitly-declared copy constructor is an `inline` ~~`public`~~ member of its class. If all of the direct and virtual base classes of `X` and all of the non-static members of class type in `X` have accessible copy constructors; the implicitly-declared copy constructor is `public`, otherwise it is inaccessible.

10 If the class definition does not explicitly declare a copy assignment operator, one is declared *implicitly*. The implicitly-declared copy assignment operator for a class `X` will have the form

```
X& X::operator=(const X&)
```

if

- — each direct base class `B` of `X` has a copy assignment operator whose parameter is of type `const B&`, `const volatile B&` or `B`, and
- — for all the non-static data members of `X` that are of a class type `M` (or array thereof), each such class type has a copy assignment operator whose parameter is of type `const M&`, `const volatile M&` or `M`.[2]

Otherwise, the implicitly declared copy assignment operator will have the form

```
X& X::operator=(X&)
```

The implicitly-declared copy assignment operator for class `X` has the return type `X&`; it returns the object for which the assignment operator is invoked, that is, the object assigned to. An implicitly-declared copy assignment operator is an `inline` ~~`public`~~ member of its class. If all of the direct and virtual base classes of `X` and all of the non-static members of class type in `X` have accessible copy assignment operators; the implicitly-declared copy assignment operator is `public`, otherwise it is inaccessible. Because a copy assignment operator is implicitly declared for a class if not declared by the user, a base class copy assignment operator is always hidden by the copy assignment operator of a derived class (**??**). A *using-declaration* (**??**) that brings in from a base class an assignment operator with a parameter type that could be that of a copy-assignment operator for the derived class is not considered an explicit declaration of a copy-assignment operator and does not suppress the implicit declaration of the derived class copy-assignment operator; the operator introduced by the *using-declaration* is hidden by the implicitly-declared copy-assignment operator in the derived class.

[2] This implies that the reference parameter of the implicitly-declared copy assignment operator cannot bind to a `volatile` lvalue; see **??**.

Chapter 14 Templates [temp]

1 A *template* defines a family of classes or functions.

template-declaration:
  `export`$_{opt}$ `late_check`$_{opt}$ `template` < *template-parameter-list* > *requires-clause*$_{opt}$ *declaration*

template-parameter-list:
  template-parameter
  template-parameter-list **,** *template-parameter*

The *declaration* in a *template-declaration* shall

— declare or define a function or a class, or

— define a member function, a member class or a static data member of a class template or of a class nested within a class template, or

— define a member template of a class or class template.

A *template-declaration* is a *declaration*. A *template-declaration* is also a definition if its *declaration* defines a function, a class, a concept map, or a static data member.

5 A class template shall not have the same name as any other template, class, concept, function, object, enumeration, enumerator, namespace, or type in the same scope (3.3), except as specified in (14.5.4). Except that a function template can be overloaded either by (non-template) functions with the same name or by other function templates with the same name (**??**), a template name declared in namespace scope or in class scope shall be unique in that scope.

Add the following new paragraphs to 9.2 [class.mem]:

11 A *template-declaration* with a *requires-clause* that is not preceded by the `late_check` keyword is a *constrained template*. Constrained templates are described in 14.9.

12 A *template-declaration* with a *requires-clause* may be preceded by the `late_check` keyword. Such a template is said to be *late-checked*. Late-checked templates are described in 14.9.4.

14.1 Template parameters [temp.param]

1 The syntax for *template-parameter*s is:

template-parameter:
  type-parameter
  parameter-declaration

type-parameter:
`class` *identifier$_{opt}$*
`class` *identifier$_{opt}$* = *type-id*
`typename` *identifier$_{opt}$*
`typename` *identifier$_{opt}$* = *type-id*
`template` < *template-parameter-list* > `class` *identifier$_{opt}$*
`template` < *template-parameter-list* > `class` *identifier$_{opt}$* = *id-expression*
`::`$_{opt}$ *nested-name-specifier$_{opt}$ concept-name identifier*
`::`$_{opt}$ *nested-name-specifier$_{opt}$ concept-name identifier* = *type-id*
`::`$_{opt}$ *nested-name-specifier$_{opt}$ concept-id identifier*
`::`$_{opt}$ *nested-name-specifier$_{opt}$ concept-id identifier* = *type-id*

Add the following new paragraphs to 14.1 [temp.param]

17 A *type-parameter* declared with a *concept-name* or *concept-id* is a template type parameter that specifies a template requirement (14.9.1) using the *simple form* of template requirements. A template type parameter written `::`$_{opt}$ *nested-name-specifier$_{opt}$ C T*, where `C` is a *concept-name*, is equivalent to a template type parameter written as `typename T` with the template requirement `::`$_{opt}$ *nested-name-specifier$_{opt}$ C<T>* added to the requirements clause (14.9.1). A template type parameter written `::`$_{opt}$ *nested-name-specifier$_{opt}$ C<T2, T3, ..., TN> T*, where `C<T2, T3, ..., TN>` is a *concept-id*, is equivalent to a a template type prameter written as `typename T` with the template requirement `::`$_{opt}$ *nested-name-specifier$_{opt}$ C<T, T2, T3, ..., TN>* added to the requirements clause (14.9.1). The first concept parameter of concept `C` shall be a type parameter, and all other concept parameters of `C` shall have default values. [*Example:*

```
concept C<typename T> { }
concept D<typename T, typename U> { }

template<C T1, D<T1> T2> void f(T1, T2);
// equivalent to
template<typename T1, typename T2> requires C<T1>, D<T2, T1> void f(T1, T2);
```

— end example]

14.4 Type equivalence [temp.type]

Add the following new paragraphs to 14.4 [temp.type]

2 In constrained template (14.9), two types are the same type if some *same-type requirement* makes them equivalent (14.9.1).

14.5 Template declarations [temp.decls]

14.5.1 Class templates [temp.class]

Add the following new paragraphs to 14.5.1 [temp.class]

5 A *member-declaration* (9.2) in a class template that is preceded by a *requires-clause* (14.9.1) is only available in instantiations in which the *requires-clause* is satisfied. [*Example:*

```
auto concept LessThanComparable<typename T> {
  bool operator<(T, T);
}

template<typename T>
class list {
```

```
  requires LessThanComparable<T> void sort();
}

struct X { };

void f(list<int> li, list<X> lX)
{
  li.sort(); // okay: LessThanComparable<int> implicitly defined
  lX.sort(); // error: no 'sort' member in list<X>
}
```

— end example]

14.5.4 Class template partial specializations [temp.class.spec]

9 Within the argument list of a class template partial specialization, the following restrictions apply:

— A partially specialized non-type argument expression shall not involve a template parameter of the partial specialization except when the argument expression is a simple *identifier*. [*Example:*

```
template <int I, int J> struct A {};
template <int I> struct A<I+5, I*2> {}; // error

template <int I, int J> struct B {};
template <int I> struct B<I, I> {};     // OK
```

— end example]

— The type of a template parameter corresponding to a specialized non-type argument shall not be dependent on a parameter of the specialization. [*Example:*

```
template <class T, T t> struct C {};
template <class T> struct C<T, 1>;              // error

template< int X, int (*array_ptr)[X] > class A {};
int array[5];
template< int X > class A<X,&array> { };        // error
```

— end example]

— The argument list of the specialization shall not be identical to the implicit argument list of the primary template, unless the specialization contains a requirements clause (14.9.1) that is more specific than the primary template's requirements clause. [*Example:*

```
concept Hashable<typename T> { int hash(T); }

template<typename T> class X { /* ... */ }; // #6
template<typename T> requires Hashable<T> class X<T> { /* ... */ }; //#7, okay
```

— end example]

The template parameter list of a specialization shall not contain default template argument values.[3)]

[3)] There is no way in which they could be used.

14.5.4.1 Matching of class template partial specializations [temp.class.spec.match]

2 A partial specialization matches a given actual template argument list if the template arguments of the partial specialization can be deduced from the actual template argument list (14.8.2) and the deduced template arguments meet the requirements in the partial specialization's requirements clause (14.9.1), if it has one. [*Example:*

```
A<int, int, 1>   a1;          // uses #1
A<int, int*, 1>  a2;          // uses #2, T is int, I is 1
A<int, char*, 5> a3;          // uses #4, T is char
A<int, char*, 1> a4;          // uses #5, T1 is int, T2 is char, I is 1
A<int*, int*, 2> a5;          // ambiguous: matches #3 and #5

int hash(int);
struct Y { };

X<int> x1;                    // uses #6
X<Y> x2;                      // uses #5
```

— end example]

4 In a type name that refers to a class template specialization, (e.g., `A<int, int, 1>`) the argument list must match the template parameter list of the primary template. If the the primary template has a requirements clause (14.9.1), the arguments must satisfy the requirements of the primary template. The template arguments of a specialization are deduced from the arguments of the primary template. The template arguments of a specialization must meet the requirements in the requirements clause (reftemp.req), if it has one [*Note:* in practice, this means that the requirements clause on a partial specialization must at least as specialized (14.5.5.2) as the requirements clause on the primary template. *— end note*]

14.5.4.2 Partial ordering of class template specializations [temp.class.order]

2 [*Example:*

```
concept C<typename T> { }
concept D<typename T> :  C<T> { }
template<int I, int J, class T> class X { };
template<int I, int J>          class X<I, J, int> { }; // #1
template<int I>                 class X<I, I, int> { }; // #2
template<int I, int J, class T> requires C<T> class X<I, J, T>; // #3
template<int I, int J, class T> requires D<T> class X<I, J, T>; // #4

template<int I, int J> void f(X<I, J, int>);            // #A
template<int I>        void f(X<I, I, int>);            // #B
template<int I, int J, class T> requires C<T> void f(X<I, J, T>); // #C
template<int I, int J, class T> requires D<T> void f(X<I, J, T>); // #D
```

The partial specialization `#2` is more specialized than the partial specialization `#1` because the function template `#B` is more specialized than the function template `#A` according to the ordering rules for function templates. The partial specialization `#4` is more specialized than the partial specialization `#3` because the function template `#D` is more specialized than the function template #C according to the ordering rules for function templates. — *end example*]

14.5.5 Function templates [temp.fct]

14.5.5.1 Function template overloading [temp.over.link]

4 The signature of a function template consists of its function signature, its return type, its requirements clause (14.9.1) and its template parameter list. The names of the template parameters are significant only for establishing the relationship between the template parameters and the rest of the signature. [*Note:* two distinct function templates may have identical function return types and function parameter lists, even if overload resolution alone cannot distinguish them.

7 Two function templates are *equivalent* if they are declared in the same scope, have the same name, have identical template parameter lists, have identical requirements clauses (14.9.1.1) and have return types and parameter lists that are equivalent using the rules described above to compare expressions involving template parameters. Two function templates are *functionally equivalent* if they are equivalent except that one or more expressions that involve template parameters in the return types and parameter lists are functionally equivalent using the rules described above to compare expressions involving template parameters. If a program contains declarations of function templates that are functionally equivalent but not equivalent, the program is ill-formed; no diagnostic is required.

14.5.5.2 Partial ordering of function templates [temp.func.order]

2 Partial ordering selects which of two function templates is more specialized than the other by transforming each template in turn (see next paragraph) and performing template argument deduction using the function parameter types, or in the case of a conversion function the return type. If template argument deduction succeeds, the deduced arguments are used to determine if the requirements of the template (14.9.1) are satisfied. The deduction process and verification of template requirements (14.9.1) determines whether one of the templates is more specialized than the other. If so, the more specialized template is the one chosen by the partial ordering process.

3 To produce the transformed template, for each type, non-type, or template template parameter synthesize a unique type (subject to the same-type requirements (14.9.1) of the template), value, or class template respectively and substitute it for each occurrence of that parameter in the function type of the template. For each requirement in the requirements clause (14.9.1), synthesize a concept map (7.6.2) for the synthesized types, values, and class templates.

4 Using the transformed function template's function parameter list, or in the case of a conversion function its transformed return type, perform type deduction against the function parameter list (or return type) of the other function. The mechanism for performing these deductions is given in **??**.

[*Example:*

```
template<class T> struct A { A(); };

template<class T> void f(T);
template<class T> void f(T*);
template<class T> void f(const T*);

template<class T> void g(T);
template<class T> void g(T&);

template<class T> void h(const T&);
template<class T> void h(A<T>&);
void m() {
```

```
  const int *p;
  f(p);                    // f(const T*) is more specialized than f(T) or f(T*)
  float x;
  g(x);                    // Ambiguous: g(T) or g(T&)
  A<int> z;
  h(z);                    // overload resolution selects h(A<T>&)
  const A<int> z2;
  h(z2);                   // h(const T&) is called because h(A<T>&) is not callable
}
```

— end example]

If the signatures of two function templates are identical modulo the requirements clause (14.9.1), partial ordering of function templates compares the requirements clauses. If one of the function templates has a requirements clause and the other does not, the function template with a requirements clause is more specialized. If both templates have requirements clauses, partial ordering determines whether the transformed function type (with its synthesized concept maps) meets the requirements in the other template's requirements clause. [*Example:*

```
template<class T> struct A { A(); };

concept C<typename T> { }
concept D<typename T> : C<T> { }
concept_map C<const int*> { }
concept_map D<float> { }
template<typename T> concept_map D<A<T>> { }

template<class T> requires C<T> void f(const T&) { } // #1
template<class T> requires D<T> void f(const T&) { } // #2
template<class T> requires C<A<T>> void f(const A<T>&) { } // #3

void m() {
  const int *p;
  f(p);            // #1 is called because #2 and #3 are not callable
  float x;
  f(x);            // #2 is called because #3 is not callable and #2 is more specialized than #1
  A<int> z;
  f(z);            // #3 is called because partial ordering based on requirements clauses does not come into effect

}
```

— end example]

Add the following new subsection to Template declarations [temp.decls]

14.5.6 Concept map templates [temp.concept.map]

1 A concept map *template* defines an unbounded set of concept maps (7.6.2) with a common set of associated function (7.6.2.1) and associated parameter (**??**) definitons. [*Example:*

```
concept Iterator<typename Iter> {
```

```
    typename value_type;
    Iter operator*(Iter);
  }

  template<typename T>
  concept_map Iterator<T*> {
    typedef T value_type;
    T* operator*(T*); // can be implicitly or explicitly declared
  }
```

— end example]

2 A concept map template not preceded by the `late_check` keyword is a *constrained template* (14.9) [*Note:* a concept map template can be a constrained template even if it does not have a requirements clause (14.9.1). *— end note*]

3 When a particular concept map is required (as specified with a *concept-id*), concept map matching determines whether a particular concept map template can be used. Concept map matching matches the concept arguments in the *concept-id* to the concept arguments in the concept map template, using matching of class template partial specializations (14.5.4.1).

4 If more than one concept map template matches a specific *concept-id*, partial ordering of concept map templates proceeds as partial ordering of class template specializations (14.5.4.2).

5 A concept map template that is not a *late-checked template* (14.9.4) shall meet the requirements of its corresponding concept (7.6.2) at the time of definition of the concept map template. [*Example:*

```
  concept F<typename T> {
    void f(T);
  }

  template<F T> struct X;

  template<typename T>
  concept_map F<X<T>> { } // error: requirement for f(X<T>) not satisfied

  template<F T> void f(X<T>); // #1

  template<typename T>
  concept_map F<X<T>> { } // okay: uses #1 to satisfy requirement for f(X<T>)
```

— end example]

6 If the definition of a concept map template depends on a primary class template or class template partial specialization (14.5.4), and instantiation of the concept map template results in a different (partial or full) specialization of that class template with an incompatible definition, the program is ill-formed. [*Example:*

```
  concept Stack<typename X> {
    typename value_type;
    value_type& top(X&);
    // ...
  }

  template<typename T> struct dynarray {
```

```
  T& top();
}

template<> struct dynarray<bool> {
  bool top();
}

template<typename T>
concept_map Stack<dynarray<T>> {
  typedef T value_type;
  T& top(dynarray<T>& x) { return x.top(); }
}

template<Stack X>
void f(X& x) {
  X::value_type& t = top(x);
}

void g(dynarray<int>& x1, dynarray<bool>& x2) {
  f(x1); // okay
  f(x2); // error: Stack<dynarray<bool>> uses the dynarray<bool> class specialization
         // rather than the dynarray primary class template, and the two
         // have incompatible signatures for top()
}
```

— end example]

14.6 Name resolution [temp.res]

No changes in this section; it is here only to allow cross-references

14.6.2 Dependent names [temp.dep]

14.6.2.1 Dependent types [temp.dep.type]

14.6.2.2 Type-dependent expressions [temp.dep.expr]

14.6.2.3 Value-dependent expressions [temp.dep.constexpr]

14.6.2.4 Dependent template arguments [temp.dep.temp]

14.6.3 Non-dependent names [temp.nondep]

1 Non-dependent names used in a template definition are found using the usual name lookup and bound at the point they are used. [*Example:*

```
void g(double);
void h();

template<class T> class Z {
public:
```

```
    void f() {
      g(1);              // calls g(double)
      h++;               // ill-formed: cannot increment function;
          // this could be diagnosed either here or
          // at the point of instantiation
    }
  };

  void g(int);                  // not in scope at the point of the template
          // definition, not considered for the call g(1)
```

— end example]

Add the following new paragraph to Non-dependent names [temp.nondep]

2 If a template contains a requirements clause (14.9.1), name lookup of non-dependent names in the template definition can find the names of associated functions in the requirements scope (3.3.8).

14.7 Template instantiation and specialization [temp.spec]

1 The act of instantiating a function, a class, a concept map, a member of a class template or a member template is referred to as *template instantiation*.

2 A function instantiated from a function template is called an instantiated function. A class instantiated from a class template is called an instantiated class. A concept map instantiated from a concept map template is called an instantiated concept map. A member function, a member class, or a static data member of a class template instantiated from the member definition of the class template is called, respectively, an instantiated member function, member class or static data member. A member function instantiated from a member function template is called an instantiated member function. A member class instantiated from a member class template is called an instantiated member class.

14.7.1 Implicit instantiation [temp.inst]

5 If the overload resolution process can determine the correct function to call without instantiating a class template definition or concept map template definition, it is unspecified whether that instantiation actually takes place. [*Example:*

```
  template <class T> struct S {
    operator int();
  };

  void f(int);
  void f(S<int>&);
  void f(S<float>);

  void g(S<int>& sr) {
    f(sr);                   // instantiation of S<int> allowed but not required
          // instantiation of S<float> allowed but not required
  };
```

— end example]

9 An implementation shall not implicitly instantiate a function template, a member template, a non-virtual member function, a member class or a static data member of a class template that does not require instantiation. [*Note:* An imple-

mentation is permitted to instantiate concept map templates that do not require instantiation, so long as instantiation of an ill-formed concept map template does not make a well-formed program ill-formed. — *end note*] It is unspecified whether or not an implementation implicitly instantiates a virtual member function of a class template if the virtual member function would not otherwise be instantiated. The use of a template specialization in a default argument shall not cause the template to be implicitly instantiated except that a class template may be instantiated where its complete type is needed to determine the correctness of the default argument. The use of a default argument in a function call causes specializations in the default argument to be implicitly instantiated.

10 Implicitly instantiated class, concept map and function template specializations are placed in the namespace where the template is defined. Implicitly instantiated specializations for members of a class template are placed in the namespace where the enclosing class template is defined. Implicitly instantiated member templates are placed in the namespace where the enclosing class or class template is defined. [*Example:*

```
namespace N {
  template<class T> class List {
  public:
    T* get();
  // ...
  };
}

template<class K, class V> class Map {
  N::List<V> lt;
  V get(K);
  // ...
};

void g(Map<char*,int>& m)
{
  int i = m.get("Nicholas");
  // ...
}
```

a call of `lt.get()` from `Map<char*,int>::get()` would place `List<int>::get()` in the namespace `N` rather than in the global namespace. — *end example*]

Add the following new paragraph to [temp.inst]

15 Unless a concept map has been explicitly defined, the concept map is implicitly instantiated when the concept map is referenced in a context that requires the concept map definition, either to satisfy a concept requirement (14.9.1) or when it is used in a *qualified-id*.

14.7.2 Explicit instantiation [temp.explicit]

1 A class, a concept map, a function or member template specialization can be explicitly instantiated from its template. A member function, member class or static data member of a class template can be explicitly instantiated from the member definition associated with its class template.

14.7.3 Explicit specialization [temp.expl.spec]

Add the following new paragraph to [temp.expl.spec]:

23 The template arguments provided for an explicit specialization shall meet the requirements (14.9.1) of the primary template. [*Example:*

```
concept C<typename T> { }
concept_map C<float> { }

template<typename T> requires C<T> void f(T);

template<> void f<float>(float); // okay: concept_map C<float> satisfies requirement
template<> void f<int>(int); // ill-formed: no concept map C<int>
```

— end example]

14.8 Function template specializations [temp.fct.spec]

14.8.2 Template argument deduction [temp.deduct]

2 When an explicit template argument list is specified, the template arguments must be compatible with the template parameter list and must result in a valid function type as described below; otherwise type deduction fails. Specifically, the following steps are performed when evaluating an explicitly specified template argument list with respect to a given function template:

- The specified template arguments must match the template parameters in kind (i.e., type, non-type, template), and there must not be more arguments than there are parameters; otherwise type deduction fails.
- Non-type arguments must match the types of the corresponding non-type template parameters, or must be convertible to the types of the corresponding non-type parameters as specified in **??**, otherwise type deduction fails.
- All references in the function type of the function template to the corresponding template parameters are replaced by the specified template argument values. If a substitution in a template parameter or in the function type of the function template results in an invalid type, type deduction fails. [Note: The equivalent substitution in exception specifications is done only when the function is instantiated, at which point a program is ill-formed if the substitution results in an invalid type.] Type deduction may fail for the following reasons:
  - Attempting to create an array with an element type that is `void`, a function type, a reference type, or an abstract class type, or attempting to create an array with a size that is zero or negative. [*Example:*

    ```
    template <class T> int f(T[5]);
    int I = f<int>(0);
    int j = f<void>(0);             // invalid array
    ```

    — end example]
  - Attempting to use a type that is not a class type in a qualified name. [*Example:*

    ```
    template <class T> int f(typename T::B*);
    int i = f<int>(0);
    ```

    — end example]
  - Attempting to use a type in a nested-name-specifier of a qualified-id when that type does not contain the specified member, or

— the specified member is not a type where a type is required, or

— the specified member is not a template where a template is required, or

— the specified member is not a non-type where a non-type is required., or

— the member is an associated type but no concept map has been defined, either implicitly (7.6.4) or explicitly (7.6.2).

[*Example:*

```
template <int I> struct X { };
template <template <class T> class> struct Z { };
template <class T> void f(typename T::Y*){}
template <class T> void g(X<T::N>*){}
template <class T> void h(Z<T::template TT>*){}
struct A {};
struct B { int Y; };
struct C {
    typedef int N;
};
struct D {
    typedef int TT;
};

int main()
{
    // Deduction fails in each of these cases:
    f<A>(0);  // A does not contain a member Y
    f<B>(0);  // The Y member of B is not a type
    g<C>(0);  // The N member of C is not a non-type
    h<D>(0);  // The TT member of D is not a template
}
```

— end example]

— Attempting to create a pointer to reference type.

— Attempting to create a reference to `void`.

— Attempting to create "pointer to member of `T`" when `T` is not a class type. [*Example:*

```
template <class T> int f(int T::*);
int i = f<int>(0);
```

— end example]

— Attempting to give an invalid type to a non-type template parameter. [*Example:*

```
template <class T, T> struct S {};
template <class T> int f(S<T, T()>*);
struct X {};
int i0 = f<X>(0);
```

— end example]

— Attempting to perform an invalid conversion in either a template argument expression, or an expression used in the function declaration. [*Example:*

```
template <class T, T*> int f(int);
int i2 = f<int,1>(0);           // can't conv 1 to int*
```

— end example]

— Attempting to create a function type in which a parameter has a type of `void`.

— The specified template arguments must meet the requirements of the template (14.9.1).

Add the following new section to 14 [temp]:

14.9 Constrained templates [temp.constrained]

1 A template that has a *requires-clause* (or declares any template type parameters using the simple form of requirements (14.1)) but is not preceded by the `late_check` keyword is a *constrained template*. Constrained templates can only be used with template arguments that satisfy the requirements of the constrained template. The template definitions of constrained templates are similarly constrained, requiring all names (including dependent names (14.6.2)) to be declared in either the requirements clause or is found through normal name lookup (**??**). [*Note:* The practical effect of constrained templates is that they provide improved diagnostics at template definition time, such that any use of the constrained template that satisfies the template's requirements is likely to result in a well-formed instantiation. *— end note*]

2 A template that is not a *constrained template* is an *unconstrained template*.

3 The template parameters of a constrained template are not dependent (14.6.2.1). A constrained template contains no dependent types (14.6.2.1), and therefore no type-dependent expressions (14.6.2.2) or dependent names (14.6.2). [*Note:* Instantiation of constrained templates still substitutes types, templates and values for template parameters, but the substitution does not require additional name lookup (3.4).

14.9.1 Template requirements [temp.req]

1 A template has a *requirements clause* if it contains a *requires-clause* or any of its template parameters were specified using the simple form of requirements (14.1). A requirements clause states the conditions under which the template can be used.

requires-clause:
  `requires` *requirement-seq*
  `requires (` *requirement-seq* `)`

requirement-seq:
  requirement `,` *requirement-seq*
  requirement

requirement:
  `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-id*
  `! ::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-id*

2 A *requires-clause* contains a list of requirements, all of which must be satisfied by the template arguments for the template. A *requirement* not preceded by a `!` is a *concept-id requirement*. A *requirement* preceded by a `!` is a *negative requirement*.

3 A *concept-id requirement* requires that a concept map corresponding to its *concept-id* be fully defined. [*Example:*

```
concept A<typename T> { }
auto concept B<typename T> { T operator+(T, T); }

concept_map A<float> { }
concept_map B<float> { }

template<typename T> requires A<T> void f(T);
template<typename T> requires A<T> void g(T);

struct X { };
void h(float x, int y, int X::* p) {
  f(x); // okay: uses concept map A<float>
  f(y); // error: no concept map A<int>; requirement not satisfied
  g(x); // okay: uses concept map B<float>
  g(y); // okay: implicitly defines and uses concept map B<int>
  g(p); // error: no implicit definition of concept map B<int X::*>; requirement not satisfied
}
```

— end example]

4 A *negative requirement* requires that no concept map corresponding to its *concept-id* be defined, implicitly or explicitly. [*Example:*

```
concept A<typename T> { }
auto concept B<typename T> { T operator+(T, T); }

concept_map A<float> { }
concept_map B<float> { }

template<typename T> requires !A<T> void f(T);
template<typename T> requires !A<T> void g(T);

struct X { };
void h(float x, int y, int X::* p) {
  f(x); // error: concept map A<float> has been defined
  f(y); // okay: no concept map A<int>
  g(x); // error: concept map B<float> has been defined
  g(y); // error: implicitly defines concept map B<int>, requirement not satisfied
  g(p); // okay: concept map B<int X::*> cannot be implicitly defined
}
```

— end example]

5 A *concept-id requirement* that refers to the `SameType` concept (**??**) is a *same-type requirement*. A same-type requirement is satisfied when its two concept arguments refer to the same type. In a constrained template (14.9), a same-type

requirement SameType<T1, T2> makes the types T1 and T2 equivalent (14.4). [*Note:* type equivalence is a congruence relation, thus

— SameType<T1, T2> implies SameType<T2, T1>,

— SameType<T1, T2> and SameType<T2, T3> implies SameType<T1, T3,

— SameType<T1, T1> is trivially true,

— SameType<T1*, T2*> implies SameType<T1, T2> and SameType<T1**, T2**>, etc.

— *end note*] [*Example:*

```
concept C<typename T> {
  typename assoc;
  assoc a(T);
}

concept D<typename T> {
  T::T(const T&);
  T operator+(T, T);
}

template<typename T, typename U>
requires C<T>, C<U>, SameType<C<T>::assoc, C<U>::assoc>, D<C<T>::assoc>
C<T>::assoc f(T t, U u) {
  return a(t) + a(u); // okay: C<T>::assoc and D<T>::assoc are the same type
}
```

— *end example*]

14.9.1.1 Requirement propagation [temp.req.prop]

1 In a template with a requirements clause (14.9.1), additional requirements implied by the declaration of the template are implicitly added to the requirement clause. The requirements are implied by the type of a function template (14.5.5), the template arguments of a class template partial specialization (14.5.4), and the concept arguments of a concept map template (14.5.6).

2 For every type T that appears as an argument or return type in a function declarator, the requirement std::MoveConstructible<T> is implicitly added to the requirements clause. [*Example:*

```
template<EqualityComparable T>
bool eq(T x, T y); // implicitly adds requirement MoveConstructible<T>
```

— *end example*]

3 For every *template-id* X<A1, A2, ..., *AN*>, where X is a constrained template, the requirements of X are implicitly added to the requirements clause after substituting the arguments A1, A2, ..., *AN* into the requirements. [*Example:*

```
template<LessThanComparable T> class set { /* ... */ };

template<CopyConstructible T>
```

```
void maybe_add_to_set(std::set<T>& s, const T& value);
// use of std::set<T> implicitly adds requirement LessThanComparable<T>
```

— end example]

4 For every associated type *concept-id*::*name*, the requirement *concept-id* is implicitly added to the requirements clause. [*Example:*

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

template<CopyConstructible T, CopyConstructible U>
Addable<T, U>::result_type // implicitly adds Addable<T, U> to the requirements clause
add(T t, U u) {
  return t + u;
}
```

— end example]

5 For every *concept-id requirement* in the requirements clause (either explicitly, or added implicitly), requirements for the refinements of the associated concept (7.6.3) and associated requirements of the concept (7.6.1.3) are implicitly added to the requirements clause.

6 Two requirements clauses are *identical* if, after the introduction of implicit requirements, they contain the same *concept-id* and *negative* requirements (14.9.1), and their *same-type requirements* (14.9.1) produce the same equivalence classes of types.

14.9.2 Opaque types [temp.opaque]

1 An *opaque type* is a type in a constrained template (14.9) that would be dependent (14.6.2.1) in the equivalent unconstrained template. A type in a constrained template is an opaque type if it is:

— a template type parameter (14.1),

— an associated type (7.6.1.2), or

— a *template-id* whose *template-name* is a template template parameter (14.1) or an associated template parameter (7.6.1.2).

2 An opaque type is a class type (9) whose members are defined by the template requirements (14.9.1) of its constrained template. The opaque type `T` contains a member function corresponding to each associated member function requirement (7.6.1.1) of each concept names by a *concept-id requirement* (14.9.1). [*Example:*

```
concept CopyConstructible<typename T> {
  T::T(const T&);
}

concept MemSwappable<typename T> {
  T::swap(T&);
}
```

```
template<typename T>
requires CopyConstructible<T>, MemSwappable<T>
void foo(T& x) {
  // opaque type T contains a copy constructor T::T(const T&) from CopyConstructible<T>
  // and a member function swap(T&) from MemSwappable<T>
  T y(x);
  y.swap(x);
}
```

— end example]

3 If no requirement specifies a default constructor for an opaque type T, a default constructor is not implicitly declared (12.1).

4 If no requirement specifies a copy constructor for an opaque type T, an inaccessible copy constructor is implicitly declared (12.8). [*Example:*

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

concept MoveConstructible<typename T> {
  T::T(T&&);
}

template<typename T>
requires DefaultConstructible<T>, MoveConstructible<T>
void f(T x) {
  T y = T(); // okay: move-constructs y from default-constructed T
  T z(x); // error: overload resolution selects implicitly-declared
          // copy constructor, which is inaccessible
}
```

— end example]

5 If no requirement specifies a copy assignment operator for an opaque type T, an inaccessible copy assignment operator is implicitly declared (12.8).

6 If no requirement specifies a destructor for an opaque type T, a destructor is not implicitly declared (12.4).

7 If two associated member function requirements for an opaque type T have the same signature, and the return types are equivalent (14.4), the duplicate signature is ignored. If the return types are not equivalent, the program is ill-formed.

14.9.3 Instantiation of constrained templates [temp.constrained.inst]

1 Instantiation of a constrained template replaces each template parameter within the definition of the template with its corresponding template argument, using the same process as for unconstrained templates (14.7).

2 In the instantiation of a constrained template, a call to a function template will undergo a second partial ordering of function templates. The function template selected at the time of the constrained template's definition is called the *seed*

function. At instantiation time, the *candidate set* of functions for the instantiation will contain all functions in the same scope as the seed function that:

— succeed at template argument deduction (14.8.2), and

— have the same name as the seed function, and

— have the same signature as the seed function, modulo the requirements claus (14.9.1), and

— are more specialized (14.5.5.2) than the seed function

Partial ordering of function templates (14.5.5.2) determines which of the function templates in the candidate set will be called in the instantiation of the constrained template. [*Example:*

```
concept InputIterator<typename Iter> { }
concept BidirectionalIterator<typename Iter> : InputIterator<Iter> { }
concept RandomAccessIterator<typename Iter> : BidirectionalIterator<Iter> { }

template<InputIterator Iter> void advance(Iter& i, Iter::difference_type n); // #1
template<BidirectionalIterator Iter> void advance(Iter& i, Iter::difference_type n); // #2
template<RandomAccessIterator Iter> void advance(Iter& i, Iter::difference_type n); // #3

template<BidirectionalIterator Iter> void f(Iter i) {
  advance(i, 1); // seed function is #2
}

concept_map RandomAccessIterator<int*> { }

void g(int* i) {
  f(i); // in call to advance(), #2 and #3 are in the candidate set
        // partial ordering of function templates selects #3
}
```

— end example]

3 In the instantiation of a constrained template, the use of a member of an opaque type (14.9.2) instantiates to a use of the corresponding member in the type that results from substituting the template arguments from the instantiation for the template parameters used in the opaque type. [*Example:*

```
auto concept MemSwappable<typename T> {
  void T::swap(T&);
}

template<MemSwappable T>
void swap(T& x, T& y) {
  x.swap(y); // when instantiated, calls X::swap(X&)
}

struct X {
  void swap(X&);
};
```

```
void f(X& x1, X& x2) {
  swap(x1, x2); // okay
}
```

— end example]

14.9.4 Late-checked templates [temp.late]

1 A *late-checked template* is a template with a requirements clause (**??**) that is preceded by the `late_check` keyword, or a concept map template (14.5.6) that is preceded by the `late_check` keyword. A late-checked template is an unconstrained template (14.9) [*Note:* it therefore follows the normal rules for computing dependent types (14.6.2.1), type-dependent expressions (14.6.2.2), and dependent names (14.6.2), and therefore does not provide the instantiation guarantees provided by constrained templates. *— end note*]

2 A late-checked template may only be used when the requirements in its requirements clause are satisfied by the template arguments (14.9.1). [*Note:* The definition of a late-checked template may still use dependent names that will be looked up at instantiation time, bypassing the declarations in concept maps that would be found if the template were a constrained template. *— end note*] [*Example:*

```
concept Semigroup<typename T> {
  T operator+(T, T);
}

concept_map Semigroup<int> {
  int operator+(int x, int y) { return x * y; }
}

template<Semigroup T>
T add(T x, T y) {
  return x + y;
}

late_check template<Semigroup T>
T late_add(T x, T y) {
  return x + y;
}

void f() {
  add(1, 2); // returns 2, because Semigroup<int>::operator+ is implemented with operator*
  late_add(1, 2); // returns 3, because late-checked template finds built-in operator+ as instantiation time
}
```

— end example]

Chapter 18 Language support library [lib.language.support]

1 This clause describes the function signatures that are called implicitly, and the types of objects generated implicitly, during the execution of some C++ programs. It also describes the headers that declare these function signatures and define any related types.

2 The following subclauses describe common type definitions used throughout the library, characteristics of the predefined types, support for concepts, functions supporting start and termination of a C++ program, support for dynamic memory management, support for dynamic type identification, support for exception processing, and other runtime support, as summarized in Table 4.

Table 4: Language support library summary

Subclause	Header(s)
?? Types	<cstddef>
	<limits>
?? Implementation properties	<climits>
	<cfloat>
	<cstdint>
	<cinttypes>
?? Start and termination	<cstdlib>
?? Dynamic memory management	<new>
?? Type identification	<typeinfo>
?? Exception handling	<exception>
	<cstdarg>
	<csetjmp>
?? Other runtime support	<ctime>
	<csignal>
	<cstdlib>
	<cstbool>
18.9 Concepts	<concepts>

Add the following new section to [lib.language.support]:

18.9 Concepts [support.concepts]

1 The header <concepts> defines several concepts that support certain kinds of template requirements (14.9.1).

Header <concepts> **synopsis**

```
namespace std {
  concept SameType<typename T, typename U> { /* see below */ }
  concept MoveConstructible<typename T> { /* see below */ }
}
```

18.9.1 Concept SameType [concept.sametype]

```
namespace std {
  concept SameType<typename T, typename U> {
    // unspecified
  }
}
```

1 The SameType concept indicates that its two concept arguments have the same type. A template requirement using the SameType concept is a *same-type requirement* (14.9.1).

18.9.2 Concept MoveConstructible [concept.moveconstruct]

```
namespace std {
  concept MoveConstructible<typename T> {
    T::T(T&&);
  }
}
```

1 The MoveConstructible concept indicates that one can move (or copy) an rvalue of type T into a new object of type T. The MoveConstructible concept is implicitly introduced by requirement propagation (14.9.1.1) for types that are argument or return types in a function declarator.

Appendix A (informative) Grammar summary [gram]

A.4 Expressions [gram.expr]

qualified-id:
  `::`$_{opt}$ *nested-name-specifier* `template`$_{opt}$ *unqualified-id*
  `::` *identifier*
  `::` *operator-function-id*
  `::` *template-id*
  `::` *concept-id*

A.6 Declarations [gram.dcl]

declaration:
  block-declaration
  function-definition
  template-declaration
  explicit-instantiation
  explicit-specialization
  linkage-specification
  namespace-definition
  concept-definition
  concept-map-definition

A.7 Declarators [gram.decl]

concept-id:
  concept-name `<` *template-argument-list*$_{opt}$ `>`

concept-name:
  identifier

concept-definition:
  `auto`$_{opt}$ `concept` *identifier* `<` *template-parameter-list* `>` *refinement-clause*$_{opt}$ *concept-body* `;`$_{opt}$

concept-body:
  `{` *concept-member-specification*$_{opt}$ `}`

concept-member-specification:
associated-function concept-member-specification$_{opt}$
associated-parameter concept-member-specification$_{opt}$
associated-requirements concept-member-specification$_{opt}$
axiom-definition concept-member-specification$_{opt}$

associated-function:
simple-declaration
function-definition
template-declaration

associated-parameter:
type-parameter `;`

associated-requirements:
requires-clause `;`

axiom-definition:
`axiom` *identifier* `(` *parameter-declaration-clause* `)` *axiom-body*

axiom-body:
`{` *axiom-seq$_{opt}$* `}`

axiom-seq:
axiom axiom-seq$_{opt}$

axiom:
expression-statement
`if` `(` *condition* `)` *expression-statement*

concept-map-definition:
`concept_map` *concept-id* `{` *concept-map-member-specification$_{opt}$* `}` `;`$_{opt}$

concept-map-member-specification:
simple-declaration concept-map-member-specification$_{opt}$
function-definition concept-map-member-specification$_{opt}$
template-declaration concept-map-member-specification$_{opt}$

A.8 Classes [gram.class]

member-declaration:
decl-specifier-seq$_{opt}$ member-declarator-list$_{opt}$ `;`
function-definition `;`$_{opt}$
`::`$_{opt}$ *nested-name-specifier* `template`$_{opt}$ *unqualified-id* `;`
using-declaration
static_assert-declaration
template-declaration
requires-clause member-declaration

A.12 Templates [gram.temp]

template-declaration:
  `export`$_{opt}$ `late_check`$_{opt}$ `template` < *template-parameter-list* > *requires-clause*$_{opt}$ *declaration*

type-parameter:
  `class` *identifier*$_{opt}$
  `class` *identifier*$_{opt}$ = *type-id*
  `typename` *identifier*$_{opt}$
  `typename` *identifier*$_{opt}$ = *type-id*
  `template` < *template-parameter-list* > `class` *identifier*$_{opt}$
  `template` < *template-parameter-list* > `class` *identifier*$_{opt}$ = *id-expression*
  `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-name identifier*
  `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-name identifier* = *type-id*
  `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-id identifier*
  `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-id identifier* = *type-id*

requires-clause:
  `requires` *requirement-seq*
  `requires` (*requirement-seq*)

requirement-seq:
  requirement , *requirement-seq*
  requirement

requirement:
  `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-id*
  `! ::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-id*

Bibliography

[1] B. Dawes. Proposed C++0x keywords considered. Technical Report N2105=06-0175, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, November 2006.

[2] D. Gregor and A. Lumsdaine. Considering concept constraint combinators. Technical Report N2161=07-0021, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, January 2007.

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

[4] G. D. Reis, B. Stroustrup, and J. Maurer. Generalized constant expressions – revision 4. Technical Report N2116=06-0186, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, November 2006.

[5] B. Stroustrup and G. D. Reis. An analysis of concept intersection. Technical Report N2221=07-0081, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, March 2007.