Proposed Wording for Concepts
(Changes from Revision 3 to Revision 4)

Authors: Douglas Gregor, Indiana University
Bjarne Stroustrup, Texas A&M University
Jeremy Siek, University of Colorado at Boulder
James Widman, Gimpel Software
Document number: N2520=08-0030
Revises document number: N2421=07-0281
Date: 2008-02-03
Project: Programming Language C++, Core Working Group
Reply-to: Douglas Gregor <doug.gregor@gmail.com>

Introduction

This document provides proposed wording for concepts as a set of changes from revision 3 (N2421) to revision 4 (N2501). This document is intended primarily for readers familiar with N2421, who are looking to determine what changes have been made. Readers not familiar with the concepts wording should instead read N2501, the latest concepts wording.

Changes from N2421

The wording in this document reflects several changes to the formulation of concepts presented in N2421 [2]. The non-trivial changes reflected in this wording are:

— The simple form of requirements for multi-parameter concepts has changed to use the `auto` keyword as a placeholder for the first concept argument 14.1, e.g.

```
concept BinaryPredicate<typename F, typename T> {
  bool operator()(F&, const T&, const T&);
}

template<Regular T, BinaryPredicate<auto, T> Pred>
void find(const T* first, const T* last, const T& value, Pred pred);

// equivalent to

template<typename T, typename Pred>
requires Regular<T> && BinaryPredicate<Pred, T>
void find(const T* first, const T* last, const T& value, Pred pred);
```

— Introduced syntax adaptation for member functions. The author's original concerns over this feature appear to be unfounded, and there are significant use cases (particularly in the Standard Library) that motivate its inclusion. However, we still do not permit syntax adaptation for constructors or destructors (for implementation reasons)

or for non-class types (for syntactic and parsing reasons). See the updated wording for associated functions in concept maps (14.9.2.1).

— Removed the automatic translation from by-value parameters to by-reference parameters within concept maps (14.9.2.1), based on recent analyses by Sean Parent and Peter Dimov. One can now describe pass-by-value parameters in concepts, which will be forwarded as rvalues.

— Removed out-of-line definition of concept map functions (14.9.2.1), making all functions in concept maps `inline` (14.9.2.1) with external linkage (3.5). The addresses of concept map functions cannot be taken (5.3.1). Also, removed the ability to specify an explicit instantiation of concept maps, which no longer makes sense (14.7.2).

— Removed late-checked members, which provided very little benefit, since they can be emulated with an auto concept containing only an associated type, and used from late-checked blocks.

— Simplified the rules for partial ordering of constrained function templates (14.5.6.1). This change makes a few more cases ambiguous (where the parameters of one function template are more specialized than the other function template, but it has fewer requirements), but it makes partial ordering more regular.

— Collapsed the section [concept.implicit], which described the implicit definition of concept map members and of concept maps (the latter, for implicit concepts only), into [concept.map.implicit], which is a far better place for this wording. The resulting organization can be a bit clearer.

— Introduced the term *instantiated archetype* to refer to an archetype whose definition comes from an instantiation of a class template. As part of this, specified that a class template specialization like `vector<T>` can be treated like a "normal" archetype if it shows up in one of the template requirements. See the new example in paragraph 17 of [temp.archetype].

— Introduced the notion of a *constrained context* (14.10), which helps describe where template parameters behave as archetypes.

— Clarified when implicitly-declared default constructors, destructors, copy constructors, and copy assignment operators are deleted (12.1, 12.4, 12.8).

— Type-checking for certain C++ constructs in constrained templates depends on specific concepts (e.g., the return type of a function must meet the requirements of the `Returnable` concept). See, for example, 8.3.

— Archetypes (14.10.2) can now have different classifications based on the template requirements (e.g. an archetype will be a scalar type if the `ScalarType` concept is used in the template requirements). Expanded the definition of archetypes to include templates and values, and we now say that a type in a constrained template aliases an archetype, rather than "having" an archetype.

— Requirement implication (14.10.1.1); previously called requirement propagation) has been extended to implicitly introduce compiler-supported concepts where possible. This section has been reworded and greatly improved thanks to feedback from Jens Maurer and John Spicer.

— Added an explicit description of the associated functions for `new`, `new[]`, `delete`, and `delete[]` requirements (14.9.1.1) and their concept matching (14.9.2.1).

— Support `explicit` constructor (14.9.2.1) and conversion requirements (14.9.2.3).

— Don't perform the conversion of arguments to references for associated functions marked `constexpr` (14.9.2.1).

— Support the use of `&` and `&&` on member function requirements, as introduced into the language through N2439, *Extending Move Semantics to* `*this` (14.9.2.1).

— Use the term *concept instance* to refer to a concept as used with specific concept arguments (as the non-syntactic equivalent of a *concept-id*). The old meaning of *concept instance* is now called a *concept map archetype*. Replaced references to *concept-id* with *concept instance* where we want to talk about the semantic entity (rather than the syntax).

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~~. Non-editorial changes from the previous wording are highlighted in green.

Purely editorial comments will be written in a separate, shaded box.

Chapter 1 General [intro]

1.3 Definitions [intro.defs]

1.3.1 [defns.signature]

signature

the name and the parameter-type-list (8.3.5) of a function, as well as the class, concept, concept map, or namespace of which it is a member. If a function or function template is a class member its signature additionally includes the *cv*-qualifiers (if any) and the *ref-qualifier* (if any) on the function or function template itself. The signature of a function template additionally includes its return type ~~and~~, its template parameter list, and its template requirements (if any). The signature of a function template specialization includes the signature of the template of which it is a specialization and its template arguments (whether explicitly specified or deduced). [*Note:* Signatures are used as a basis for name mangling and linking. — *end note*]

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	continue	friend	register	throw
auto	default	goto	reinterpret_cast	true
axiom	delete	if	requires	try
bool	do	inline	return	typedef
break	double	int	short	typeid
case	dynamic_cast	late_check	signed	typename
catch	else	long	sizeof	union
char	enum	mutable	static	unsigned
char16_t	explicit	namespace	static_assert	using
char32_t	export	new	static_cast	virtual
class	extern	operator	struct	void
concept	false	private	switch	volatile
concept_map	float	protected	template	wchar_t
const	for	public	this	while
const_cast				

Chapter 3 Basic concepts [basic]

3 An *entity* is a value, object, subobject, base class subobject, array element, variable, function, instance of a function, enumerator, type, class member, template, namespace, ~~or~~ parameter pack, concept, or concept map.

6 Some names denote types, classes, concepts, concept map names, 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 (14.9), concept map (14.9.2), enumeration type ([dcl.enum]), inline function with external linkage ([dcl.fct.spec]), class template (clause 14), non-static function template (14.5.6), static data member of a class template ([temp.static]), member function of a class template ([temp.mem.func]), or template specialization for which some template parameters are not specified (14.7, 14.5.5) 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 (14.9) is immediately after the identifier in the *concept-definition*. The point of declaration for a concept map (14.9.2) is immediately after the *concept-id* in the *concept-map-definition*.

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

3.3.8 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 bodies in that concept or concept map.

2) A name `N` used in a concept or concept map `S` shall refer to the same declaration in its context and when re-evaluated in the completed scope of `S`. No diagnostic is required for a violation of this rule.

3) 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.

4) 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.

5) ~~The potential scope of a declaration that extends to or past the end of a 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 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 (14.9.3) its concept,

— after the `::` scope resolution operator (5.1) applied to the name of a concept map or template type parameter (14.1).

3.3.9 Requirements scope [basic.scope.req]

1 In a ~~template that contains template requirements (14.10.1)~~constrained template (14.10), the names of all associated functions inside the concepts named ~~or implied~~ by the concept requirements in the template's requirements are visible in the scope of the template declaration. ~~If the name of an associated function is the same as the name of a *template-parameter* in the scope of the template, the program is ill-formed ([temp.local]).~~

[*Example:*

```
concept Integral<typename T> {
  T::(const 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 operator- finds RAIterator<Iter>::operator-
                          // and Integral<RAIterator<Iter>::difference_type>::operator-
                          // overload resolution picks the appropriate operator for both uses of -
}
```

— *end example*]

3.3.10 Name hiding [basic.scope.hiding]

1 A name can be hidden by an explicit declaration of that same name in a nested declarative region, refining concept (14.9.3), or derived class ([class.member.lookup]).

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 3.3.8.

3.4 Name lookup [basic.lookup]

1 The name lookup rules apply uniformly to all names (including *typedef-names* ([dcl.typedef]), *namespace-names* ([basic.namespace]), *concept-names* (14.9), *concept-map-names* (14.9.2), and *class-names* ([class.name]) 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 ([basic.def]) of that name. Name lookup shall find an unambiguous declaration for the name (see [class.member.lookup]). 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 (13.1). Overload resolution ([over.match]) takes place after name lookup has succeeded. The access rules (clause [class.access]) 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.

[*Example:*

```
concept Callable<class F, class T1> {
  result_type operator() (F&, T1)
  typename result_type; // error result_type used before declared
}
```

— *end example*]

17 A name used in the definition of an associated function (14.9.1.1) 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 ([stmt.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 (but not concept), or namespace member or enumerator can be referred to after the `::` scope resolution operator (5.1) applied to a *nested-name-specifier* that nominates its class, concept map, namespace, or

enumeration. During the lookup for a name preceding the :: scope resolution operator, object, function, and enumerator names are ignored. If the name found does not designate a namespace, concept map, or a class, enumeration, or dependent type, the program is ill-formed.

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

6 In a constrained template (14.10), a name prefixed by a *nested-name-specifier* that nominates a template type parameter `T` is looked up in each concept named by a concept requirement (14.10.1) in the template requirements whose template argument list contains `T`. That name shall refer to one or more associated types (names of associated functions are ignored) that are all equivalent (14.4).

[*Example:*

```
concept C<typename T> {
  typename assoc_type;
}

template<typename T, typename U> requires C<T> && C<U>
  T::assoc_type      // okay: refers to C<T>::assoc_type
  f();
```

— *end example*]

If qualified name lookup for associated types does not find any associated type names, qualified name lookup (3.4.3) can still find the name within the archetype (14.10.2) of `T`.

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 map (not a concept), the name specified after the *nested-name-specifier* is looked up in the scope of the concept map (3.3.8) or any of the concept maps for concepts its concept refines (14.9.3.1). 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 (3.3.8). [*Example:*

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

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

— *end example*] — *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.

3.5 Program and linkage [basic.link]

5 In addition, a member function, static data member, a named class or enumeration of class scope, or an unnamed class or enumeration defined in a class-scope typedef declaration such that the class or enumeration has the typedef name for linkage purposes ([dcl.typedef]), has external linkage if the name of the class has external linkage. An ~~explicitly-defined~~ associated function definition (14.9.2.1) has external linkage.

3.9 Types [basic.types]

1 [*Note:* 3.9 and the subclauses thereof impose requirements on implementations regarding the representation of types. There are two kinds of types: fundamental types and compound types. Types describe objects ([intro.object]), references (8.3.2), or functions (8.3.5). In a constrained context (14.10), type archetypes can behave like different kinds of types, e.g., object types, scalar types, literal types, etc. — *end note*]

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 13.5; for *conversion-function-id*s, see 12.3.2; for *template-id*s, see [temp.names] . 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 ([class.mfct.non-static]). — *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*

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

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*. 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`." The address of a member of a concept map (14.9.2) shall not be taken, either implicitly or explicitly. [*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 ([dcl.stc]) does not reflect

the `mutable` specifier associated with the non-static data member. — *end note*]

5.19 Constant expressions [expr.const]

3 A constant expression is an *integral constant expression* if it is of integral or enumeration type~~.~~, or, in a constrained template (14.10), if it is of a type *cv* `T` that is an archetype and if the concept requirement `IntegralConstantExpressionType<T>` ([concept.support]) is part of the template's requirements. [*Note:* such expressions may be used as array bounds (8.3.4, 5.3.4), as case expressions (6.4.2), as bit-field lengths (9.6), as enumerator initializers (7.2), as static member initializers (9.4.2), and as integral or enumeration non-type template arguments (14.3). — *end note*]

Chapter 6 Statements [stmt.stmt]

1 Except as indicated, statements are executed in sequence.

statement:
labeled-statement
expression-statement
compound-statement
selection-statement
iteration-statement
jump-statement
declaration-statement
try-block
late-check-block

6.9 Late-checked block [stmt.late]

1 In a constrained context (14.10), a late-checked block treats the enclosed statements as if they were not in a constrained context. Outside of a constrained context, the late-checked block has no effect. [*Note:* in a late-checked block, template parameters do not behave as if they were replaced with their corresponding archetypes. Thus, template parameters imply the existence of dependent types, type-dependent expressions, and dependent names as in an unconstrained template. *— end note*]

late-check-block:
`late_check` *compound-statement*

2 [*Example:*

```
concept Semigroup<typename T> {
  T::T(const 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) {
  T r = x + y; // uses Semigroup<T>::operator+
  late_check {
    r = x + y; // uses operator+ found at instantiation time (not considering Semigroup<T>::operator+)
  }
```

```
  return r;
}
```

— *end example*]

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
alias-declaration

alias-declaration:
`using` *identifier* `=` *type-id*

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 [dcl.asm] , and *linkage-specification*s are described in [dcl.link] . *Function-definition*s are described in [dcl.fct.def] and *template-declaration*s are described in clause 14. *Namespace-definition*s are described in [namespace.def] , *concept-definition*s are described in 14.9.1, *concept-map-definition*s are described in 14.9.2, *using-declaration*s are described in 7.3.3 and *using-directive*s are described in [namespace.udir] . — *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 [dcl.spec] 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, concept map, 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.

7.3.3 The `using` declaration [namespace.udecl]

1 A *using-declaration* introduces a name into the declarative region in which the *using-declaration* appears. That name is a synonym for the name of some entity declared elsewhere.

using-declaration:
`using` `typename`$_{opt}$ `::`$_{opt}$ *nested-name-specifier unqualified-id* `;`
`using` `::` *unqualified-id* `;`
`using` `::`$_{opt}$ *nested-name-specifier*$_{opt}$ `concept_map` `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-id* `;`
`using` `::`$_{opt}$ *nested-name-specifier*$_{opt}$ `concept_map` `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-name*$_{opt}$ `;`
`using` `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-name* `;`

21 A *using-declaration* for a concept map is an alias to the concept map that matches (14.5.8) the concept instance corresponding to the *concept-id* from the specified namespace. [*Example:*

```
namespace N1 {
  concept C<typename T> { }
}
namespace N2 {
  concept_map N1::C<int> { } // A
  template<typename T> concept_map N1::C<T*> { } // B
}
namespace N3 {
  using N2::concept_map N1::C<int>;   // aliases A
  using N2::concept_map N1::C<int*>; // aliases B, instantiated with T=int
}
```

— end example]

22 A *using-declaration* for a concept map that specifies a *concept-name* (and not a *concept-id*) brings all of the concept maps and concept map templates from the specified namespace for the given concept into the scope in which the *using-declaration* appears. [*Example:*

```
namespace N1 {
  concept C<typename T> { }
  template<C T> void f(T) { }
}
namespace N2 {
  concept_map N1::C<int> { } // A
  template<typename T> concept_map N1::C<T*> { } // B
}
namespace N3 {
  using N2::concept_map N1::C; // aliases A and B
```

```
  void g() {
    f(1); // uses concept map N1::C<int> from A
    f(new int); // uses concept map N1::C<int*> instantiated from B with T=int
  }
}
```

— end example]

23 If no concept is specified in the concept map using declaration, all concept maps from the specified namespace are brought into scope. [*Example:*

```
namespace N1 {
  concept C<typename T> { }
  template<C T> void f(T) { }
}
namespace N2 {
  concept D<typename T> { }
}
namespace N3 {
  concept_map N1::C<int> { } // A
  template<typename T> concept_map N1::C<T*> { } // B
  concept_map N2::D<int> { } // C
}
namespace N4 {
  using N3::concept_map; // aliases A, B, and C
}
```

— end example]

24 If the second *nested-name-specifier* is specified but no concept is specified, then all concept maps in the namespace specified by the first *nested-name-specifier* for all concepts in the namespace specified by the second *nested-name-specifier* are brought into scope.

[*Note:* a *using-directive* for a namespace brings the concept maps of that namespace into scope, just like other entities. *— end note*] [*Example:*

```
namespace N1 {
  concept C<typename T> { }
}
namespace N2 {
  concept_map N1::C<int> { }
}
namespace N3 {
  using namespace N2;

  template<N1::C T> void foo(T) { };

  void bar() {
    foo(17); // ok, finds the concept map from N2
  }
}
```

— end example]

Chapter 8 Declarators [dcl.decl]

8.3 Meaning of declarators [dcl.meaning]

7 In a constrained template (14.10), a type archetype *cv* `T` shall only be used as the type of a variable if the template has a concept requirement `VariableType<T>`.

8.3.1 Pointers [dcl.ptr]

5 In a constrained template (14.10), a type archetype *cv* `T` shall only be used to form a type "pointer to *cv* `T`" if the template has a concept requirement `PointeeType<T>`.

8.3.2 References [dcl.ref]

6 In a constrained template (14.10), a type archetype *cv* `T` shall only be used to form a type "reference to *cv* `T`" if the template has a concept requirement `ReferentType<T>`.

8.3.3 Pointers to members [dcl.mptr]

3 A pointer to member shall not point to a static member of a class ([class.static]), a member with reference type, or "*cv* `void`." In a constrained template (14.10), a pointer to member shall only point to a type archetype *cv* `T` if the template has a concept requirement `ReferentType<T>`. [*Note:* see also 5.3 and [expr.mptr.oper] . The type "pointer to member" is distinct from the type "pointer", that is, a pointer to member is declared only by the pointer to member declarator syntax, and never by the pointer declarator syntax. There is no "reference-to-member" type in C++. — *end note*]

8.3.4 Arrays [dcl.array]

2 An array can be constructed from one of the fundamental types (except `void`), from a pointer, from a pointer to member, from a class, from an enumeration type, or from another array. In a constrained template (14.10), an array shall only be constructed from a type archetype *cv* `T` if the template has a concept requirement `ObjectType<T>`.

8.3.5 Functions [dcl.fct]

6 If the type of a parameter includes a type of the form "pointer to array of unknown bound of `T`" or "reference to array of unknown bound of `T`," the program is ill-formed.[1)] Functions shall not have a return type of type array or function, although they may have a return type of type pointer or reference to such things. There shall be no arrays of functions, although there can be arrays of pointers to functions. In a constrained template (14.10), a type archetype *cv* `T` shall only be used as the return type of a function type if the template has a concept requirement `Returnable<T>`. Types shall not be defined in return or parameter types. The type of a parameter or the return type for a function definition shall not be

[1)] This excludes parameters of type "*ptr-arr-seq* `T2`" where `T2` is "pointer to array of unknown bound of `T`" and where *ptr-arr-seq* means any sequence of "pointer to" and "array of" derived declarator types. This exclusion applies to the parameters of the function, and if a parameter is a pointer to function or pointer to member function then to its parameters also, etc.

an incomplete class type (possibly cv-qualified) unless the function definition is nested within the *member-specification* for that class (including definitions in nested classes defined within the class).

Chapter 9 Classes [class]

9.2 Class members [class.mem]

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

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

member-requirement:
requires-clause

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 9 [class]

19 A non-template *member-declaration* that has a *member-requirement* (14.10.1) is a *constrained member* and shall only occur in a class template (14.5.1) or nested class thereof. A constrained member shall be a member function. A constrained member is treated as a constrained template (14.10).

20 ~~A *member-declaration* that contains a `late_check` is a *late-checked member*. In the declaration of a late-checked member, archetypes are not substituted for their corresponding (dependent) types. [*Note:* the effect is that late-checked members are processed as if they occurred in an unconstrained template. — *end note*] A late-checked member shall only occur in a constrained class template (14.10) or nested class thereof. A late-checked member shall declare a data~~

~~member. Late-checked members shall not be named from a constrained template, except within a late-checked block (6.9) or in the declaration of a late-checked member. [*Example:* — *end example*]~~

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`, 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 ([class.virtual]) and no virtual base classes ([class.mi]), 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.

An implicitly-declared default constructor for class `X` is deleted if:

- any non-static data member is of reference type,
- any non-static data member of const-qualified type (or array thereof) does not have a user-provided default constructor; or
- any non-static data member or direct or virtual base class has class type `M` (or array thereof) and `M` has no default constructor, or if overload resolution ([over.match]) as applied to `M`'s default constructor, results in an ambiguity or a function that is deleted or inaccessible from the implicitly-declared default constructor.

7 A non-user-provided default constructor for a class is *implicitly defined* when it is used (3.2) to create an object of its class type ([intro.object]). The implicitly-defined or explicitly-defaulted default constructor performs the set of initializations of the class that would be performed by a user-written default constructor for that class with an empty *mem-initializer-list* ([class.base.init]) and an empty function body. ~~If that user-written default constructor would be ill-formed, the program is ill-formed.~~If the implicitly-defined copy constructor is explicitly defaulted, but the corresponding implicit declaration would have been deleted, the program is ill-formed. If that user-written default constructor would satisfy the requirements of a constexpr constructor ([decl.constexpr]), the implicitly-defined default constructor is constexpr. Before the non-user-provided default constructor for a class is implicitly defined, all the non-user-provided default constructors for its base classes and its non-static data members shall have been implicitly defined. [*Note:* an implicitly-declared default constructor has an *exception-specification* ([except.spec]). An explicitly-defaulted definition has no implicit *exception-specification*. — *end note*]

12.3 Conversions [class.conv]

12.3.2 Conversion functions [class.conv.fct]

1 A member function of a class `X` having no parameters or an associated function of a concept whose sole parameter is of type `X`, and with a name of the form

conversion-function-id:
`operator` *conversion-type-id*

conversion-type-id:
type-specifier-seq conversion-declarator$_{opt}$

conversion-declarator:
ptr-operator conversion-declarator$_{opt}$

specifies a conversion from `X` to the type specified by the *conversion-type-id*. Such ~~member~~ functions are called conversion functions. Classes, enumerations, and *typedef-name*s shall not be declared in the *type-specifier-seq*. ~~Neither parameter types nor~~No return type can be specified. The type of a conversion function (8.3.5) is "function taking no parameter (if the conversion function is a member function) or a parameter of type `X` (if the conversion function is an associated function) returning *conversion-type-id*." A conversion function is never used to convert a (possibly cv-qualified) object to the (possibly cv-qualified) same object type (or a reference to it), to a (possibly cv-qualified) base class of that type (or a reference to it), or to (possibly cv-qualified) void.[2)]

[*Example:*

```
class X {
    // ...
public:
    operator int();
};

void f(X a)
{
    int i = int(a);
    i = (int)a;
    i = a;
}
```

In all three cases the value assigned will be converted by `X::operator int()`. — *end example*]

12.4 Destructors [class.dtor]

3 If a class has no user-declared 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.

[2)] Even though never directly called to perform a conversion, such conversion functions can be declared and can potentially be reached through a call to a virtual conversion function in a base class

An implicitly-declared destructor for a class `X` is deleted if:

— any of the non-static data members has class type `M` (or array thereof) and `M` has an deleted destructor or a destructor that is inaccessible from the implicitly-declared destructor, or

— any direct or virtual base class has a deleted destructor or a destructor that is inaccessible from the implicitly-declared 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&`.[3)]

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. An implicitly-declared copy constructor for a class `X` is deleted if `X` has:

— a non-static data member of class type `M` (or array thereof) that cannot be copied because overload resolution ([over.match]), as applied to `M`'s copy constructor, results in an ambiguity or a function that is deleted or inaccessible from the implicitly-declared copy constructor, or

— a direct or virtual base class `B` that cannot be copied because overload resolution ([over.match]), as applied to `B`'s copy constructor, results in an ambiguity or a function that is deleted or inaccessible from the implicitly-declared copy constructor.

7 A non-user-provided copy constructor is *implicitly defined* if it is used to initialize an object of its class type from a copy of an object of its class type or of a class type derived from its class type[4)]. [*Note:* the copy constructor is implicitly defined even if the implementation elided its use ([class.temporary]). — *end note*] A program is ill-formed ~~if the class for which a copy constructor is implicitly defined or explicitly defaulted has:~~if the implicitly-defined copy constructor is explicitly defaulted, but the corresponding implicit declaration would have been deleted.

— ~~a non-static data member of class type (or array thereof) with an inaccessible or ambiguous copy constructor, or~~

— ~~a base class with an inaccessible or ambiguous copy constructor.~~

Before the non-user-provided copy constructor for a class is implicitly defined, all non-user-provided copy constructors for its direct and virtual base classes and its non-static data members shall have been implicitly defined. [*Note:* an

[3)] This implies that the reference parameter of the implicitly-declared copy constructor cannot bind to a `volatile` lvalue; see [diff.special] .

[4)] See [dcl.init] for more details on direct and copy initialization.

implicitly-declared copy constructor has an *exception-specification* ([except.spec]). An explicitly-defaulted definitions has no implicit *exception-specifion*. — *end note*]

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`.[5)]

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. An implicitly-declared copy assignment operator for class `X` is deleted if `X` has:

— a non-static data member of `const` non-class type (or array thereof), or

— a non-static data member of reference type, or

— a non-static data member of class type `M` (or array thereof) that cannot be copied because overload resolution ([over.match]), as applied to `M`'s copy assignment operator, results in an ambiguity or a function that is deleted or inaccessible from the implicitly-declared copy assignment operator, or

— a direct or virtual base class `B` that cannot be copied because overload resolution ([over.match]), as applied to `B`'s copy assignment operator, results in an ambiguity or a function that is deleted or inaccessible from the implicitly-declared copy assignment operator.

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 ([over.ass]). A *using-declaration* (7.3.3) 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.

12 A non-user-provided copy assignment operator is *implicitly defined* when an object of its class type is assigned a value of its class type or a value of a class type derived from its class type. A program is ill-formed ~~if the class for which a copy assignment operator is implicitly defined has:~~if the implicitly-defined copy constructor is explicitly defaulted, but the corresponding implicit declaration would have been deleted.

— ~~a non-static data member of `const` type, or~~

— ~~a non-static data member of reference type, or~~

[5)] This implies that the reference parameter of the implicitly-declared copy assignment operator cannot bind to a `volatile` lvalue; see [diff.special] .

— ~~a non-static data member of class type (or array thereof) with an inaccessible copy assignment operator, or~~

— ~~a base class with an inaccessible copy assignment operator.~~

Before the non-user-provided copy assignment operator for a class is implicitly defined, all non-user-provided copy assignment operators for its direct base classes and its non-static data members shall have been implicitly defined. [*Note:* an implicitly-declared copy assignment operator has an *exception-specification* ([except.spec]). An explicitly-defaulted definition has no implicit *exception-specification*. — *end note*]

Chapter 13 Overloading [over]

13.1 Overloadable declarations [over.load]

2 Certain function declarations cannot be overloaded:

— Function declarations that differ only in the return type cannot be overloaded.

— Member function declarations with the same name ~~and~~, the same parameter-type-list and the same template requirements (if any), the same cannot be overloaded if any of them is a `static` member function declaration ([class.static]). Likewise, member function template declarations with the same name, the same parameter-type-list, ~~and~~ the same template parameter lists, and the same template requirements (if any) cannot be overloaded if any of them is a `static` member function template declaration. The types of the implicit object parameters constructed for the member functions for the purpose of overload resolution ([over.match.funcs]) are not considered when comparing parameter-type-lists for enforcement of this rule. In contrast, if there is no `static` member function declaration among a set of member function declarations with the same name and the same parameter-type-list, then these member function declarations can be overloaded if they differ in the type of their implicit object parameter. [*Example:* the following illustrates this distinction:

```
class X {
    static void f();
    void f();                   // ill-formed
    void f() const;             // ill-formed
    void f() const volatile;    // ill-formed
    void g();
    void g() const;             // OK: no static g
    void g() const volatile;    // OK: no static g
};
```

— end example]

— Member function declarations with the same name and the same *parameter-type-list* as well as member function template declarations with the same name, the same *parameter-type-list*, ~~and~~ the same template parameter lists, and the same template requirements, cannot be overloaded if any of them, but not all, have a *ref-qualifier* (8.3.5). [*Example:*

```
class Y {
    void h() &;
    void h() const &; // OK
    void h() &&; // OK, all declarations have a ref-qualifier
    void i() &;
    void f() const; // ill-formed, prior declaration of i
```

```
                                // has a ref-qualifier
};
```

— *end example*]

13.5 Overloaded operators [over.oper]

13.5.4 Function call [over.call]

1 If declared in a class type, `operator()` shall be a non-static member function with an arbitrary number of parameters. It can have default arguments. It implements the function call syntax

postfix-expression `(` *expression-list*$_{opt}$ `)`

where the *postfix-expression* evaluates to a class object and the possibly empty *expression-list* matches the parameter list of an `operator()` member function of the class. Thus, a call `x(arg1,...)` is interpreted as `x.operator()(arg1,...)` for a class object `x` of type `T` if `T::operator()(T1, T2, T3)` exists and if the operator is selected as the best match function by the overload resolution mechanism ([over.match.best]).

2 If declared in a concept, `operator()` shall be a non-member associated function with one or more parameters. ~~It can have default arguments.~~ It implements the function call syntax

postfix-expression `(` *expression-list*$_{opt}$ `)`

where the *postfix-expression* evaluates to an object and the possibly empty *expression-list* matches the parameter list of the `operator()` associated function after the first parameter of the parameter list has been removed. Thus, a call `x(arg1,...)` is interpreted as `operator()(x, arg1, ...)` for an object `x` of type `T` if `operator()(T, T1, T2, T3)` exists and if the operator is selected as the best match function by the overload resolution mechanism ([over.match.best]).

13.5.5 Subscripting [over.sub]

1 If declared in a class type, `operator[]` shall be a non-static member function with exactly one parameter. It implements the subscripting syntax

postfix-expression `[` *expression* `]`

Thus, a subscripting expression `x[y]` is interpreted as `x.operator[](y)` for a class object `x` of type `T` if `T::operator[](T1)` exists and if the operator is selected as the best match function by the overload resolution mechanism ([over.match.best]).

2 If declared in a concept, `operator[]` shall be a non-member associated function with exactly two parameters. It implements the subscripting syntax

postfix-expression `[` *expression* `]`

Thus, a subscripting expression `x[y]` is interpreted as `operator[](x, y)` for an object `x` of type `T` if `operator[](T, T1)` exists and if the operator is selected as the best match function by the overload resolution mechanism ([over.match.best]).

13.5.6 Class member access [over.ref]

1 If declared in a class type, `operator->` shall be a non-static member function taking no parameters. It implements class member access using `->`

postfix-expression `->` *id-expression*

An expression `x->m` is interpreted as `(x.operator->())->m` for a class object `x` of type `T` if `T::operator->()` exists and if the operator is selected as the best match function by the overload resolution mechanism ([over.match]).

2 If declared in a concept, `operator->` shall be a non member associated function taking exactly one parameter. It implements class member access using `->`

postfix-expression `->` *id-expression*

An expression `x->m` is interpreted as `(operator->(x))->m` for an object `x` of type `T` if `operator->(T)` exists and if the operator is selected as the best match function by the overload resolution mechanism ([over.match]).

Chapter 14 Templates [temp]

1 A *template* defines a family of classes~~or functions~~, functions, or concept maps, or an alias for a family of types.

template-declaration:
`export`$_{opt}$ `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, or

— be an *alias-declaration*~~.~~, or

— define a concept map.

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 (14.5.5). 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 ([temp.over]), a template name declared in namespace scope or in class scope shall be unique in that scope.

Add the following new paragraphs to [temp]:

12 A *template-declaration* with a `requires` keyword is a constrained template (14.10). The *requires-clause* specifies template requirements (14.10.1).

14.1 Template parameters [temp.param]

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

template-parameter:
type-parameter
parameter-declaration

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

simple-requirement-argument-list:
`auto`
`auto` , *template-argument-list*

4 A non-type *template-parameter* shall have one of the following (optionally *cv-qualified*) types:

— integral or enumeration type,

— pointer to object or pointer to function,

— reference to object or reference to function,

— pointer to member~~.~~, or

— in a constrained template (14.10), a type archetype `T` for which the concept requirement `NonTypeTemplateParameterType<T>` ([concept.support]) is part of the template's requirements.

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

18 A *type-parameter* declared with a *concept-name* is a template type parameter or parameter pack that specifies a template requirement (14.10.1) using the *simple form* of template requirements. A template type parameter or parameter pack written ::$_{opt}$ *nested-name-specifier*$_{opt}$ *C* ...$_{opt}$ *T*, where `C` is a *concept-name*, is equivalent to a template type parameter or parameter pack written as `typename T` or `typename... T`, respectively, with the template requirement or pack expansion ::$_{opt}$ *nested-name-specifier*$_{opt}$ *C<T>* ...$_{opt}$ added to the template requirements. A template type parameter or parameter pack written ::$_{opt}$ *nested-name-specifier*$_{opt}$ *C* <*auto, T2, T3, ..., TN*>...$_{opt}$ *T*, is equivalent to a template type parameter or parameter pack written as `typename T` or `typename... T`, respectively, with the template requirement ::$_{opt}$ *nested-name-specifier*$_{opt}$ *C<T, T2, T3, ..., TN>*...$_{opt}$ added to the template requirements. The first concept parameter of concept `C` shall be a type parameter, and all concept parameters not otherwise specified shall have default values. [*Example:*

```
concept C<typename T> { ... }
concept D<typename T, typename U> { ... }
concept E<typename T, typename U, typename V = U> { ... }

template<C T, D<auto, T> P> void f(T, P);
// equivalent to
template<class T, class P> requires C<T> && D<P, T> void f(T, P);

template<C T, E<auto, T> P> void f(T, P);
```

```
// equivalent to
template<class T, class P> requires C<T> && E<P, T, T> void f(T, P);
```

— *end example*]

When the *type-parameter* is a template type parameter pack, the equivalent requirement is a pack expansion (14.5.3). [*Example:*

```
concept C<typename T> { }

template<C... Args> void g(Args const&...);
// equivalent to
template<typename... Args> requires C<Args>... void g(Args const&...);
```

— *end example*]

14.4 Type equivalence [temp.type]

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

2 In a constrained template (14.10), two types are the same type if some same-type requirement makes them equivalent (14.10.1).

14.5 Template declarations [temp.decls]

14.5.1 Class templates [temp.class]

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

5 A constrained member (9.2) in a class template is declared only in class template specializations in which its template requirements (14.10.1) are satisfied. If there exist multiple overloads of the constrained member with identical signatures, ignoring the template requirements, only the most specialized overload, as determined by partial ordering of the template requirements (14.5.6.1), will be declared in the instantiation. If partial ordering ~~does not produce a single best candidate~~results in an ambiguity, ~~a deleted function with the given signature will be declared in the instantiation~~use of the function results in an ambiguity. [*Example:*

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

concept Radix<T> : LessThanComparable<T> { /* ... */ }

template<typename T>
class list {
  requires LessThanComparable<T> void sort(); //#1
  requires Radix<T> void sort(); //#2
};

struct X { };
concept_map Radix<int> { /* ... */ }

void f(list<float> lf, list<int> li, list<X> lX)
```

```
{
  lf.sort(); // okay: LessThanComparable<float> implicitly defined, calls #1
  li.sort(); // okay: calls #2, which is more specialized than #1
  lX.sort(); // error: no 'sort' member in list<X>
}
```

— end example]

14.5.3 Variadic templates [temp.variadic]

1 A *template parameter pack* is a template parameter that accepts zero or more template arguments. [*Example:*

```
template<class ... Types> struct Tuple { };

Tuple<> t0;             // Types contains no arguments
Tuple<int> t1;          // Types contains one argument: int
Tuple<int, float> t2;   // Types contains two arguments: int and float
Tuple<0> eror;          // error: 0 is not a type
```

— end example]

[*Note:* a template parameter pack can also occur in a concept parameter list (14.9.1). [*Example:*

```
auto concept Callable<typename F, typename... Args> {
  typename result_type;
  result_type operator()(F&, Args...);
}
```

— end example] *— end note*]

4 A *pack expansion* is a sequence of tokens that names one or more parameter packs, followed by an ellipsis. The sequence of tokens is called the *pattern of the expansion*; its syntax depends on the context in which the expansion occurs. Pack expansions can occur in the following contexts:

- In an *expression-list* ([expr.post]); the pattern is an *assignment-expression*.
- In an *initializer-list* ([dcl.init]); the pattern is an *initializer-clause*.
- In a *base-specifier-list* ([class.derived]); the pattern is a *base-specifier*.
- In a *mem-initializer-list* ([class.base.init]); the pattern is a *mem-initializer*.
- In a *template-argument-list* ([temp.arg]); the pattern is a *template-argument*.
- In an *exception-specification* ([except.spec]); the pattern is a *type-id*.
- In a *requirement-list* (14.10.1); the pattern is a *requirement*.

6 The instantiation of an expansion produces a ~~comma-separated~~ list $\mathtt{E}_1$~~,~~$\oplus$ $\mathtt{E}_2$~~,~~$\oplus$ ...~~,~~$\oplus$ $\mathtt{E}_N$, where N is the number of elements in the pack expansion parameters and $\oplus$ is the syntactically-appropriate separator for the list. Each $\mathtt{E}_i$ is generated by instantiating the pattern and replacing each pack expansion parameter with its ith element. All of the $\mathtt{E}_i$ become elements in the enclosing list. [*Note:* The variety of list varies with the context: *expression-list*, *base-specifier-list*, *template-argument-list*, *requirement-list*, etc. *— end note*]

14.5.5 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 template requirements that are more specific (14.5.6.1) than the primary template's requirements. [*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.[6)]

— An argument shall not contain an unexpanded parameter pack. If an argument is a pack expansion (14.5.3), it shall be the last argument in the template argument list.

10 The template requirements of a primary class template are implied (14.10.1.1) in its class template partial specializations. [*Example:*

```
concept LessThanComparable<typename T> { /* ... */ }
concept Hashable<typename T> { /* ... */ }

template<typename T> requires LessThanComparable<T> class Y { /* ... */ };
template<typename T>
```

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

```
requires Hashable<T> // same as requires LessThanComparable<T> && Hashable<T>
class Y<T> { /* ... */ };
```

— end example]

14.5.5.1 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 satisfy the partial specialization's template requirements (if any). [*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

concept_map Hashable<int> { /* ... */ }
struct Y { };

X<int> x1;                      // uses #7
X<Y> x2;                        // uses #6
```

— 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 primary template has template requirements, the arguments shall satisfy those requirements. The template arguments of a specialization are deduced from the arguments of the primary template.

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

2 [*Example:*

```
concept Con1<typename T> { }
concept Con2<typename T> :  Con1<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 Con1<T> class X<I, J, T> { }; // #3
template<int I, int J, class T> requires Con2<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 Con1<T> void f(X<I, J, T>); // C
template<int I, int J, class T> requires Con2<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 partial ordering rules for function templates. — *end example*]

14.5.6 Function templates [temp.fct]

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 template requirements, 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.

14.5.6.1 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. [*Note:* if template argument deduction succeeds, the deduced arguments were used to determine if the requirements of the template are satisfied. — *end note*] The deduction process 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 (including template parameter packs thereof) synthesize a unique type, value, or class template respectively and substitute it for each occurrence of that parameter in the function type of the template. When the template is a constrained template, the unique type is an archetype and concept maps for each of the requirements stated in or implied by its template requirements are also synthesized; see 14.10. [*Note:* because the unique types are archetypes, two template type parameters may share the same archetype due to same-type constraints. — *end note*]

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 [temp.deduct.partial] .

[*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 partial ordering of function templates has not determined which template is more specialized, and if at least one of the function templates has template requirements, partial ordering of function templates compares the template requirements. If one of the function templates has template requirements and the other does not, the function template with the template requirements is more specialized. If both templates have template requirements, partial ordering determines whether the transformed function type (with its synthesized concept maps, 14.10.2) satisfies the requirements of the other template.~~ [*Note:* when two constrained templates have identical signatures (ignoring template requirements), the partial ordering is based on those template requirements. Similarly, a constrained template is more specialized than an unconstrained template because it has more strict requirements. *— end note*] [*Example:*

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

template<CopyConstructible T> struct A { A(); };

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

template<class T> requires C<T> void f(T&) { } //#1
template<class T> requires D<T> void f(T&) { } //#2
template<class T> requires C<A<T>> void f(A<T>&) { } //#3
template<class T> void f(T&); //#4

void m() {
  int *p;
  f(p);           // calls #1: template argument deductions fails #2 and #3, and #1 is more specialized than #4
  float x;
  f(x);           // #2 is called because #3 is not callable and #2 is more specialized than #1 and #4
  A<int> z;
  f(z);           // ambiguous: no partial ordering between #2 and #3
}
```

— end example]

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

14.5.8 Concept map templates [temp.concept.map]

1 A *concept map template* defines an unbounded set of concept maps with a common set of associated function, associated

type, and associated template definitions. [*Example:*

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

template<typename T>
concept_map F<T*> {
  typedef T& type;
  T& f(T*);
}
```

— *end example*]

2 A concept map template is a constrained template (14.10) [*Note:* a concept map template may be a constrained template even if it does not have template requirements. — *end note*]

3 Within the *template-argument-list* of the *concept-id* in a concept map template (including nested template argument lists), the following restrictions apply:

— A non-type argument expression shall not involve a template parameter of the concept map except when the argument expression is a simple *identifier*.

— The type of a template parameter corresponding to a non-type argument shall not be dependent on a parameter of the concept map.

— The template parameter list of a concept map template shall not contain default template argument values.[7)]

4 When a particular concept map is required, concept map matching determines whether a particular concept map template can be used. Concept map matching matches the concept arguments in the ~~*concept-id*~~*concept instance* to the concept arguments in the concept map template, using matching of class template partial specializations (14.5.5.1).

5 If more than one concept map template matches a specific ~~*concept-id*~~*concept instance*, partial ordering of concept map templates proceeds as partial ordering of class template specializations (14.5.5.2).

6 A concept map template shall satisfy the requirements of its corresponding concept (14.9.2) at the time of definition of the concept map template. [*Example:*

```
concept C<typename T> { }

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

template<C T> struct X;

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

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

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

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

— end example]

7 If the definition of a concept map template ~~instantiates a primary class template or a class template partial specialization (14.5.5) with template arguments that contain one or more archetypes~~uses an instantiated archetype (14.10.2), and instantiation of the concept map template results in a different specialization of that class template with an incompatible definition, the program is ill-formed. The specialization is considered to have an incompatible definition if the specialization's definition causes a different definition of any associated type or associated template in the concept map, if its definition causes any of the associated function definitions to be ill-formed, or if the resulting concept map fails to satisfy the axioms of the corresponding concept. [*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]

8 A concept map template shall be declared before the first use of a concept map that would make use of the concept map template as the result of an implicit or explicit instantiation in every translation unit in which such a use occurs; no

diagnostic is required.

14.6 Name resolution [temp.res]

14.6.3 Non-dependent names [temp.nondep]

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

2 [*Note:* if a template contains template requirements, name lookup of non-dependent names in the template definition can find the names of associated functions in the requirements scope (3.3.9). — *end note*]

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, concept map template, a member class or a static data member of a class template that does not require instantiation. 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 If no concept map exists for a given ~~*concept-id*~~concept instance , and there exists a concept map template that matches the ~~*concept-id*~~concept instance, 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 (14.10.1) or when name lookup finds a concept map member.

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.

2 The syntax for explicit instantiation is:

explicit-instantiation:
`extern`$_{opt}$ `template` *declaration*
~~`extern`$_{opt}$ `template concept_map` *concept-id* `;`~~

14.7.3 Explicit specialization [temp.expl.spec]

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

23 [*Note:* The template arguments provided for an explicit specialization shall satisfy the template requirements of the primary template (14.5.5.1). — *end note*] [*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 satisfies the requirement for 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). There must not be more arguments than there are parameters, unless at least one parameter is a template parameter pack. 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 [temp.arg.nontype] , otherwise type deduction fails.

— All references in the function type and template requirements of the function template to the corresponding template parameters are replaced by the specified template argument values. If a substitution in a template parameter, the template requirements (if any), 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.

[*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`.

— Attempting to use a type in a *nested-name-specifier* of a *qualified-id* that refers to a member in a concept instance, for which there is no concept map corresponding to that concept instance (14.9.2).

— Attempting to instantiate a pack expansion containing multiple parameters packs whose lengths are different.

— Attempting to use a class or function template with template arguments that do not satisfy the template requirements. [*Example:*

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

template<typename T> requires C<T> class X { /* ... */ };

template<typename T> int f(X<T>*);
int i0 = f<int>(0);
```

— *end example*]

— If the specified template arguments do not satisfy the requirements of the template, type deduction fails.

Add the following new sections to 14 [temp]:

14.9 Concepts [concept]

1 Concepts describe an abstract interface that can be used to constrain templates (14.10). Concepts state certain syntactic and semantic requirements (14.9.1) 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 concept map (14.9.2)~~names a specific use of a concept by its *concept-name* and a set of concept arguments. The concept and its concept arguments, together, are referred to as a *concept instance*. [*Example:* `CopyConstructible<int>` is a *concept-id* if name lookup (3.4) determines that the identifier `CopyConstructible` refers to a *concept-name*; then, `CopyConstructible<int>` is a concept instance that refers to the `CopyConstructible` concept used with the type `int`. — *end example*]

14.9.1 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 declare *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*. A *full concept name* is an identifier that is treated as if it were composed of the concept name and the sequence of its enclosing namespaces.

3 Concepts shall only be defined at namespace scope.

4 A concept that starts with `auto` defines an *implicit concept* (14.9.2.3), otherwise it defines an *explicit concept*.

5 The *template-parameter-list* of a *concept-definition* shall not contain any requirements specified in the simple form (14.10.1).

6

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

concept-member-specification:
concept-member-specifier concept-member-specification$_{opt}$

concept-member-specifier:
associated-function
type-parameter `;`
associated-requirements
axiom-definition

The body of a concept contains associated functions (14.9.1.1), associated types (14.9.1.2), associated templates, associated requirements (14.9.1.3), and axioms (14.9.1.3). A name `x` declared in the body of a concept shall refer to only one of: an associated type, an associated template, an axiom, the name of the concept, or one or more associated functions that have been overloaded (13).

14.9.1.1 Associated functions [concept.fct]

1 Associated functions describe functions, member functions, or operators that specify the functional behavior of the concept arguments and associated types and templates (14.9.1.2). A concept map (14.9.2) for a given concept must provide, either implicitly (14.9.2.3) or explicitly (14.9.2.1), definitions for each associated function in the concept.

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

2 An *associated-function* shall declare ~~one or more~~a function or function template. If the *declarator-id* of the declaration is a *qualified-id*, its *nested-name-specifier* shall name a template parameter of the enclosing concept; the declaration declares a member function or member function template. An associated function shall not be `extern` or `static` ([dcl.stc]), inline ([dcl.fct.spec]), explicitly-defaulted or deleted ([dcl.fct.def]), or a friend function. An associated function shall not contain an *exception-specification* ([except.spec]).

3 Associated functions may specify requirements for non-member functions and operators. [*Example:*

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

— end example]

4 With the exception of the assignment operator ([over.ass]) and operators `new`, `new[]`, `delete`, and `delete[]`, associated functions shall specify requirements for operators as non-member functions. [*Note:* This restriction applies even to the operators `()`, `[]`, and `->`, which can otherwise only be ~~overloaded via~~declared as non-static member functions (13.5): [*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] *— end note*]

5 Associated functions may specify requirements for static or non-static member functions, constructors, and destructors. [*Example:*

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

— end example]

6 Associated functions may specify requirements for `new` and `delete`. [*Example:*

```
concept HeapAllocatable<typename T> {
  void* T::operator new(std::size_t);
  void* T::operator new[](std::size_t);
  void T::operator delete(void*);
  void T::operator delete[](void*);
}
```

— end example]

7 Associated functions may 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]

8 Concepts may contain overloaded associated functions (clause 13). [*Example:*

```
concept C<typename X> {
  void f(X);
  void f(X, X); // okay
```

```
  int f(X, X); // error: differs only by return type
};
```

— *end example*]

9 Associated member functions with the same name and the same *parameter-type-list*, as well as associated member function templates with the same name, the same *parameter-type-list*, the same template parameter lists, and the same template requirements (if any), cannot be overloaded if any of them, but not all, have a *ref-qualifier* (8.3.5).

10 Associated ~~non-member~~ functions may have a default implementation. This implementation will be instantiated when implicit definition of an implementation (14.9.2.3) for the associated function (14.9.2.1) fails. A default implementation of an associated function is a constrained template (14.10). [*Example:*

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

14.9.1.2 Associated types and templates [concept.assoc]

1 Associated types and associated templates are types and templates, respectively, defined in the concept body and used in the description of the concept.

2 An associated type 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 types and templates may be provided with a default value. The default value will be used to define the associated type or template when no corresponding definition is provided in a concept map (14.9.2.2). [*Example:*

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

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

— *end example*]

4 Associated types ~~and templates~~ may use the simple form to specify requirements (14.10.1) on the associated type ~~or template~~. The simple form is equivalent to a declaration of the associated type ~~or template~~ followed by an associated requirement (14.9.1.3) stated using the general form (14.10.1). [*Example:*

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

concept Container<typename X> {
  InputIterator iterator; // same as typename iterator; requires InputIterator<iterator>;
}
```

— end example]

14.9.1.3 Associated requirements [concept.req]

1 Associated requirements place additional requirements on concept parameters, associated types, and associated templates. Associated requirements have the same form and behavior as template requirements in a constrained template (14.10).

associated-requirements:
requires-clause ;

[*Example:*

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

— end example]

14.9.1.4 Axioms [concept.axiom]

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

axiom-definition:
requires-clause$_{opt}$ `axiom` *identifier* `(` *parameter-declaration-clause* `)` *axiom-body*

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

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

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

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

```
concept Semigroup<typename Op, typename T> : CopyConstructible<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 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 (14.9.1.1) in the concept, one of the concepts it refines (14.9.3), or in an associated requirement (14.9.1.3). [*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*]

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

```
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. The *expression* is contextually converted to bool (clause [conv]). 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 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; }
  axiom Totality(Op op, T x, T y) { (op(x, y) || op(y, x)) == true; }
}
```

— *end example*]

6 An axiom containing a *requires-clause* only applies when the specified template requirements are satisfied. [*Example:*

```
concept EqualityComparable2<typename T, typename U = T> {
  bool operator==(T, U);
  bool operator!=(T, U);

  requires SameType<T, U> axiom Reflexivity(T x) {
      x == x; // okay: T and U have the same type
    }
}
```

— *end example*]

7 Whether an implementation replaces any expression according to an axiom is implementation-defined. With the exception of such substitutions, the presence of an axiom shall have no effect on the observable behavior of the program. [*Note:* the intent of axioms is to provide a mechanism to express the semantics of concepts. Such semantic information can be used for optimization, software verification, software testing, and other program analyses and transformations, all of which are outside the scope of this International Standard. — *end note*]

14.9.2 Concept maps [concept.map]

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

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

concept-map-member-specification:
concept-map-member concept-map-member-specification$_{opt}$

concept-map-member:
simple-declaration
function-definition
template-declaration

2 Concept maps describe how a set of template arguments satisfy the requirements stated in the body of a concept definition (14.9.1). Whenever a constrained template specialization (14.10) is named, there shall be a concept map corresponding to each concept requirement in the template requirements. This concept map may be written explicitly (14.9.2), instantiated from a concept map template (14.5.8), or generated implicitly (14.9.2.3). The concept map (14.9.1) is inserted into the scope in which the concept map or concept map template (14.5.8) is defined immediately after the *concept-id* is seen. The name of the concept map is the full concept name of the concept in the ~~*concept-id*~~corresponding concept instance. [*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 A concept map for an implicit concept is implicitly defined when it is needed. The implicitly-defined concept map is defined in the namespace of the concept. [*Example:*

```
auto concept Addable<typename T> {
  T::T(const 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*]

4 Concept maps shall provide~~, either implicitly or explicitly,~~ a definition for every associated function (14.9.1.1), associated type (14.9.1.2), and associated template of the concept named by its ~~*concept-id*~~concept instance and ~~any of its refined concepts~~all of the requirements inherited from its refined concepts. (14.9.3). [*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*]

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

6 Concept maps shall not contain declarations that do not satisfy 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*]

7 At the point of definition of a concept map, all associated requirements (14.9.1.3) of the corresponding concept and its refined concepts (14.9.3) 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>

class file_iterator { ... };

concept_map ForwardIterator<file_iterator> {
  typedef long difference_type;
} // error: no concept_map SignedIntegral<long> if ptrdiff_t is not long
```

— *end example*]

8 If a concept map is provided for a particular ~~*concept-id*~~concept instance, then that concept map shall be defined before a constrained template referring to that ~~*concept-id*~~concept instance is instantiated. If the introduction of a concept map changes a previous result (e.g., in template argument deduction (14.8.2)), the program is ill-formed, no diagnostic

required. Concept map templates must be instantiated if doing so would affect the semantics of the program. A concept map for a particular concept instance shall not be defined both implicitly and explicitly in the same namespace in a program. If one translation unit of a program contains an explicitly-defined concept map for that concept instance, and a different translation contains an implicitly-defined concept map for that concept instance, then the program is ill-formed, no diagnostic required.

9 The implicit or explicit definition of a concept map asserts that the axioms (14.9.1.4) stated in its corresponding concept (and the refinements of that concept) hold, permitting an implementation to perform the transformations described in [concept.axiom]. If an axiom is violated, the behavior of the program is undefined. ~~[*Note:* axioms may be used for transformation and optimization of programs without verifying their correctness. — *end note*]~~

14.9.2.1 Associated function definitions [concept.map.fct]

1 Associated ~~non-member~~ function requirements (14.9.1.1) 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 const&);
  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> const& vec) { return vec.empty(); }
  void push(std::vector<T>& vec, value_type const& 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 with the same name and signature, after substitution of the concept arguments in the concept map's ~~*concept-id*~~concept instance into the associated function ~~and, in some cases, transformation of parameter types to references. Let `P` be the type produced by substituting the concept arguments into the declared type of a parameter in an associated function of the concept corresponding to the concept map. The actual type `Q` of the corresponding parameter in the concept map is `P`, if `P` is a reference type or if the associated function is `constexpr`, or `P const&`, if `P` is not a reference type.~~ [*Example:*

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

struct Y {};
```

```
concept_map C<Y> {
  void f(Y const&); // okay: matches required signature void f(Y const&)
}

concept_map C<int> {
  void f(int) { } // error: does not match required signature void f(int const&)
}
```

— end example]

3 ~~Functions declared within a concept map may be defined outside the concept map.~~ Functions defined in a concept map are *inline*.

4 Function templates declared within a concept map match an associated function template with the same signature. [*Example:*

```
concept InputIterator<typename Iter> {
  typename value_type;
  // ...
}

concept C<typename X> {
  typename value_type;

  template<InputIterator Iter>
    requires Convertible<Iter::value_type, value_type>
    void assign(X&, Iter first, Iter last); //#1
}

concept_map C<MyContainer> {
  typedef int value_type;

  template<InputIterator Iter>
    requires Convertible<Iter::value_type, int>
    void assign(MyContainer&, Iter first, Iter last)
    { ... } // matches #1
}
```

— end example]

5 Member functions and member function templates (not including constructors or destructors) declared within a concept map match an associated member function or member function template, respectively, with the same signature. Such member functions and member function templates can only be declared for a class type `X`; for non-class types, member functions and member function templates in the concept map are declared implicitly (14.9.2.3). [*Example:*

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

struct X { };
```

```
concept_map Swappable<X> {
  void X::swap(X& other) { /* ... */ }
}
```

— end example]

6 Associated function and function template requirements that name a constructor (14.9.1.1) are satisfied by constructors in the corresponding concept map argument (call it `X`). Let `parm1`, `parm2`, ..., `parm`*N* be the constructor parameters and `parm1′`, `parm2′`, ..., `parm`*N*′ be expressions, where each `parm`*i*′ is an *id-expression* naming `parm`*i*. If the declared type of `parm`*i* is an lvalue reference type, then `parm`*i*′ is treated as an lvalue, otherwise, `parm`*i*′ is treated as an rvalue; then

— if the constructor requirement has $N \neq 1$ parameters, it is satisfied if an object of type `X` can be direct-initialized with arguments `parm1′`, `parm2′`, ..., `parm`*N*′, [*Example:*

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

struct X { X(long, int); };
concept_map TwoIntConstructible<X> { } // okay: X has a constructor that can accept two ints
                                       // (the first is converted to a long)
```

— end example]

— if the constructor requirement has one parameter, the requirement is satisfied if an object of type `X` can be initialized with the argument `parm1′`. The initialization is direct-initialization if the constructor requirement is `explicit`, or copy-initialization if the constructor requirement is not `explicit`, [*Example:*

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

concept EC<typename T> {
  explicit T::T(int);
}

struct X {
  X(int);
};

struct Y {
  explicit Y(int);
};

concept_map IC<X> { } // okay
concept_map EC<X> { } // okay
concept_map IC<Y> { } // error: cannot copy-initialize Y from an int
concept_map EC<Y> { } // okay
```

— end example]

7 An associated function requirement for a destructor is satisfied if the corresponding concept map argument (call it `X`) is a non-class type or has a public, non-deleted destructor. [*Example:*

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

concept_map Destructible<int> { } // okay: int is not a class 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]

14.9.2.2 Associated type and template definitions [concept.map.assoc]

1 Definitions in the concept map provide types and templates that satisfy requirements for associated types and templates (14.9.1.2), respectively.

2 Associated type parameter requirements are satisfied by type definitions in the 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 requirements are satisfied by class template definitions or template aliases ([temp.alias]) 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]

14.9.2.3 Implicit definitions [concept.map.implicit]

1 Any of the requirements of a concept and its refined concepts (14.9.3) that are not satisfied by the definitions in the body of a concept map (14.9.2.1, 14.9.2.2) are *unsatisfied requirements*.

2 Definitions for unsatisfied requirements in a concept map are implicitly generated from the requirements and their default values as follows. If any unsatisfied requirement is not matched by this process, the concept map is ill-formed.

3 The implicit definition of a concept map involves the implicit definition of concept map members for each associated non-member function (14.9.1.1) and associated type or template (14.9.1.2) 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 not be satisfied by the implicitly-defined concept map (14.9.2), 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]

4 The implicit concept map member defined for an associated ~~non-member~~ function or function template (14.9.1.1), including member functions, has the same signature as the associated function or function template, after the concept map parameters have been substituted into the associated function or function template [*Note:* the implicitly-defined function satisfies the associated function or function template (14.9.2.1) *— end note*]. Let `parm1`, `parm2`, ..., `parmN` be the parameters of the associated function and `parm1′`, `parm2′`, ..., `parmN′` be expressions, where each `parmi′` is an *id-expression* naming `parmi`. If the declared type of `parmi` is an ~~r~~lvalue reference type, then `parmi′` is treated as an ~~r~~lvalue, otherwise, `parmi′` is treated as an ~~r~~value.

For an associated member function (or member function template) in a type `X` (after substitution the concept map's template arguments into the associated member function or member function template), let `x` be an object of type *cv* `X`, where *cv* are the *cv-qualifiers* on the associated member function (or member function template). If the requirement has no *ref-qualifier* or if its *ref-qualifier* is `&`, `x` is an lvalue; otherwise, `x` is an rvalue.

If the return type of the associated 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 or function template is a prefix unary operator `Op`, the *expression* is `Op parm1′`,

— if the associated function or function template is a postfix unary operator `Op`, the *expression* is `parm1′ Op`,

— if the associated function or function template is a binary operator `Op`, the *expression* is `parm1′ Op parm2′`,

— if the associated function or function template is the function call operator, the *expression* is `parm1′(parm2′, parm3′, ..., parmN′)`,

— if the associated function is a conversion operator, the *expression* is `parm1′` if the conversion operator requirement is not `explicit` and `(U)parm1′` if the conversion operator requirement is `explicit`, where `U` is the return type of the conversion operator,

— ~~otherwise,~~if the associated function or function template is a non-member function or function template (call it `f`). The *expression* is an unqualified call ([expr.call]) to `f` whose arguments are the parameters `parm1′`, `parm2′`, ..., `parmN′`,

— if the associated member function requirement is a requirement for an operator `new` or `new[]`, the *expression* is `X::operator new(parm1′, parm2′, ..., parmN′)` or `X::operator new[](parm1′)`, respectively, or, if that *expression* is ill-formed, `::operator new(parm1′, parm2′, ..., parmN′)` or `::operator new[](parm1′)`, respectively,

— if the associated member function requirement is a requirement for an operator `delete` or `delete[]`, the *expression* is `delete x` or `delete[] x`, respectively,

— otherwise, ~~the associated member function or member function template requirement requires a member function or member function template (call it `f`), respectively. 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~~the *expression* is `x.f(parm1′, parm2′, ..., parmN′)` ~~is well-formed and its type is implicitly convertible to the return type of the associated member function or member function template requirement~~.

If the *expression* is ill-formed, and the associated ~~non-member~~ function has a default implementation (14.9.1.1), the implicit concept map member is defined by substituting the concept map arguments into the default implementation.

5 ~~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 implicitly-defined associated function definitions, so that the use of constrained templates does not incur any overhead relative to unconstrained templates. — *end note*]

6 The implicit concept map member defined for an associated type or template can have its value deduced from the return type of an associated function definition in the concept map or from the member functions used to satisfy the requirements of the corresponding concept (14.9.2.3, 14.9.2.1), using template argument deduction ([temp.deduct.type]). Let `P` be the return type of the associated function after substitution of the concept parameters specified by the concept map with their concept arguments, and where each undefined associated type and associated template has been replaced with a newly invented type or template template parameter, respectively. Let `A` be defined by the following:

— ~~if the associated function is a member function requirement, `A` is `decltype(e)`, where `e` is the *expression* used to determine that the associated function requirement is satisfied (14.9.2.1), or~~

— if the function definition that satisfies the associated function requirement is implicitly defined, `A` is `decltype(e)`, where `e` is the *expression* synthesized for the body of that function definition, otherwise,

— if the function definition that satisfies the associated function requirement was defined explicitly, `A` is the return type of that function definition in the concept map.

The definitions of the associated parameters are determined using the rules of template argument deduction from a function call ([temp.deduct.call]), where `P` is a function template parameter type and `A` the corresponding argument type. If the deduction fails, no concept map members are implicitly defined by that associated function. If the results of deduction produced by different associated functions yield more than one possible `A` for a particular `P`-`A` combination, that associated type or template is not implicitly defined. [*Example:*

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

template<typename T> requires Dereferenceable<T> void f(T&);

void g(int* x) {
  f(x); // okay: Dereferenceable<int*> implicitly defined
                    // implicitly-defined Dereferenceable<int*>::operator* calls built-in * for integer pointers
                    // implicitly-defined Dereferenceable<int*>::value_type is int
}
```

— *end example*]

7 If an associated type or template (14.9.1.2) has a default argument, a concept map member satisfying the associated type or template 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*]

14.9.3 Concept refinement [concept.refine]

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

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

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

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

2 Refinements specify an inheritance relationship among concepts. A concept `B` named in a *refinement-specifier* of concept `D` is a *refined concept* of `D` and `D` is a *refining concept* of `B`. A concept refinement inherits all requirements in the body of a concept (14.9.1), such that the requirements of the refining concept are a superset of the requirements of the refined concept. [*Note:* when a concept `D` refines a concept `B`, every set of concept arguments that satisfies the requirements of `D` also satisfies the requirements of `B`. The refinement relationship is transitive. — *end note*] [*Example:* In the following example, `EquilateralPolygon` refines `Polygon`. Thus, every `EquilateralPolygon` is a `Polygon`, and constrained templates (14.10) that are well-formed with a `Polygon` constraint are well-formed when given an `EquilateralPolygon`.

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

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

— *end example*]

3 A *requirement-specifier* shall refer to a defined concept. [*Example:*

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

— *end example*]

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

5 The *template-argument-list* of a *refinement-specifier*'s *concept-id* 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*]

6 Within the definition of a concept, a ~~concept instance~~concept map archetype (14.10.2) is synthesized for each *refinement-specifier* in the concept's *refinement-clause* (if any).

14.9.3.1 Concept member lookup [concept.member.lookup]

1 Concept member lookup determines the meaning of a name (*id-expression*) in concept scope (3.3.8). The following steps

define the result of name lookup for a member name `f` in concept scope `C`. 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 type or template (14.9.1.2), then the result of name lookup is the associated type or template.

3 If the name `f` is declared in concept scope `C`, and `f` refers to one or more associated functions (14.9.1.1), then the result of name lookup is ~~an overload set~~the set consisting of the associated functions in `C` in addition to the ~~overload sets~~associated functions in each concept scope in C_R for which name lookup of `f` results in ~~an overload set~~a set of associated functions. [*Example:*

```
concept C1<typename T> : CopyConstructible<T> {
  T f(T); //#1
}

concept C2<typename T> {
  typename f;
}

concept D<typename T> : C1<T>, C2<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 (C_R). If name lookup of `f` is ambiguous in any concept scope C_R, name lookup of `f` in `C` is ambiguous. Otherwise, the set of concept scopes $C_{R'}$ is a subset of 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 $C_{R'}$ is empty, name lookup of `f` in `C` returns no result, or

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

— if `f` refers to ~~an overload set~~one or more functions in all of the concept scopes in $C_{R'}$, then `f` refers to ~~an overload set~~the set consisting of all associated functions from all of ~~these overload sets~~the concept scopes in $C_{R'}$, or

— if `f` refers to an associated type in all concept scopes in $C_{R'}$, and all of the associated types are equivalent (14.10.1), the result is the associated type `f` found first by a depth-first traversal of the refinement clauses,

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

[*Example:*

```
concept A<typename T> { typename t; }

concept B<typename T> { typename t; }

concept C<typename T> :  A<T>, B<T> {
  f(t); // error: ambiguous, the two t's are not equivalent
  f(A<T>::t); // okay
}
```

— *end example*]

5 When name lookup in a concept scope `C` results in ~~an overload set~~a set of associated functions, 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 refinement clauses 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*]

14.9.3.2 Implicit concept maps for refined concepts [concept.refine.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 in the namespace of which the concept map is a member unless already defined ~~(either implicitly or explicitly)~~. [*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 refinement, definitions in the concept map for the refining concept can also be used to satisfy the requirements of the refined concept (14.9.2). [*Note:* a single function definition in a concept map can be used to satisfy multiple requirements. — *end note*] [*Example:* in this example, the concept map `D<int>` implicitly defines the concept map `C<int>`.

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

concept D<typename T> : C<T> {
  void g(T);
}

concept_map D<int> {
  int f(int x) { return -x; } // satisfies requirement for C<int>::f
  void g(int x) { } // satisfies requirement for C<int>::g and D<int>::g
}
```

— *end example*]

3 Concept map templates (14.5.8) are implicitly defined only for certain refinements of the concept corresponding to the concept map template. A concept map template for a particular refined concept is defined if all of the template parameters of the refining concept map template can be deduced from the *template-argument-list* of the *refinement-specifier*'s ~~*concept-id*~~corresponding concept instance ([temp.deduct.type]). If template argument deduction fails, then a concept map template corresponding to the refined concept shall have been defined~~, either implicitly or explicitly~~. [*Example:*

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

template<typename T> struct A { };

template<typename T> concept_map D<A<T>, T> { }
// implicitly defines:
template<typename T> concept_map C<A<T>> { }

template<typename T, typename U>
  concept_map D<T, A<U>> { } // ill-formed: cannot deduce template parameter U from C<T>
                             // and there is no concept map template C<T>
```

— *end example*]

4 Each concept map or concept map template shall provide only definitions corresponding to requirements of the refinements of its concept that are compatible with the definitions provided by the concept map or concept map template named by the refinement. A definition in the refining concept map or concept map template is compatible with its corresponding definition in the refined concept map or concept map template if

— the definition in the refined concept map or concept map template was implicitly defined from the refining concept map or concept map template,

— the definition was explicitly provided in the refined concept map or concept map template and implicitly defined in the refining concept map or concept map template, or

— the definitions satisfy an associated type or template requirement (14.9.1.2) and both definitions name the same type or template, respectively.

If a program contains definitions in a concept map or concept map template that are not compatible with their corresponding definitions in a refined concept map or concept map template, the program ~~violates the one definition rule (3.2)~~is ill-formed. [*Note:* if the concept maps or concept map templates with definitions that are not compatible occur in different translation units, no diagnostic is required. — *end note*] [*Example:*

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

concept D<typename T> : C<T> {
  int g(T);
}

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

concept_map C<int> {
  typedef int assoc;
  int f(int x) { return x; }
}

concept_map D<int> {
  typedef int assoc; // okay: same type as C<int>::assoc
  // okay: f is not defined in D<int>
  int g(int x) { return -x; } // okay: satisfies D<int>::g
}

concept_map E<int> {
  typedef float assoc; // error: E<int>::assoc and D<int>::assoc are not the same type
  // okay: f is not defined in D<int>
  int g(int x) { return x; } // error: D<int>::g already defined in concept map D<int>
}
```

— *end example*]

14.10 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)) is a *constrained template*. A constrained template can only be instantiated with template arguments that satisfy its template requirements. The template definitions of constrained templates are similarly constrained, requiring all names to be ~~declared in either the template requirements or~~ found through name lookup at template definition time (3.4). [*Note:* names can be found in the template requirements of a constrained template (3.3.9). 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 A *constrained context* is a part of a constrained template in which all name lookup is resolved at template definition time. Names that would be dependent outside of a constrained context are found in the current scope, which includes the template requirements of the constrained template (3.3.9). [*Note:* Within a constrained context, template parameters behave as if aliased their corresponding archetypes (14.10.2) so there are no dependent types ([temp.dep.type]), and no type-dependent values ([temp.dep.expr]) or dependent names ([temp.dep]). Instantiation in constrained contexts (14.10.3) still substitutes types, templates and values for template parameters, but the substitution does not require additional name lookup (3.4). — *end note*] A constrained context is:

- the body of a constrained function template,
- the *expression* in a `decltype` type or `sizeof` expression that occurs within the signature of a constrained function template,
- the *base-clause* (if any) of a constrained class template,
- a member of a constrained class template,
- the body of a concept map template,
- a member of a concept map template,
- the body of a concept,
- a member of a concept.

4 Any context that is not a constrained context is an *unconstrained context*. Within a constrained context, several constructs provide unconstrained contexts:

- a late-checked block (6.9) indicates a compound statement that is an unconstrained context,
- a default template argument in a *template-parameter* is an unconstrained context, and
- a default argument in a *parameter-declaration*.

14.10.1 Template requirements [temp.req]

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

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

requirement-list:
requirement `...`$_{opt}$ `&&` *requirement-list*
requirement `...`$_{opt}$

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 containing a `!` is a *concept requirement*. A *requirement* containing a `!` is a *negative requirement*.

3 A concept requirement requires that there be a most specific concept map according to concept map matching and partial ordering of concept map templates (14.5.8). [*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 B<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*]

A concept requirement is satisfied in the following way. First, unqualified name lookup (3.4) searches for all concept maps for the full concept name of the concept in the concept instance corresponding to the *concept-id*. Then concept map matching (14.5.8) selects the most specific concept map ~~or concept instance~~ that matches the arguments in the ~~*concept-id*~~concept instance. [*Example:* In the following, to satisfy the template requirement for `f`, the implementation performs name loookup to find concept maps for the concept `N1::C`. Lookup finds three concept maps: `::concept_map N1::C<bool>`, `N3::concept_map N1::C<int>`, and `N3::concept_map ' N1::C<float>`. Concept map matching then selects the second concept map because that matches the requirement `C<T>` with `T` replaced by the deduced type argument `int`.

```
namespace N1 {
  concept C<typename T> { };
  template<C T> void f(T x);
```

```
}
namespace N2 {
  concept_map N1::C<int> { };
  concept C<typename T> { };
}
concept_map N1::C<bool> { };
namespace N3 {
  concept_map N1::C<int> { };
  concept_map N1::C<float> { };
  concept_map N2::C<char> { };

  void g() { f(1); }
}
```

— *end example*]

4 If there are no matching concept maps found by unqualified name lookup, then the namespace of which the concept is a member is searched for concept maps. This search finds only those concept maps defined in the same namespace as the concept, ignoring any *using-declaration*s that apply to concept maps (7.3.3). [*Example:*

```
namespace N1 {
  concept C<typename T> { };
  concept_map C<int> { };
}
template<N1::C T> void f(T x);
void g() { f(1); } // Ok, finds N1::concept_map C<int> because it is in the same namespace as concept N1::C.
```

— *end example*]

5 A negative requirement requires that no concept map corresponding to its ~~*concept-id*~~concept instance 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 !B<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*]

6 A concept requirement that refers to the `SameType` concept ([concept.support]) is a *same-type requirement*. A same-type requirement is satisfied when its two concept arguments refer to the same type (including the same *cv* qualifiers). In a constrained template (14.10), a same-type requirement `SameType<T1, T2>` makes the types `T1` and `T2` equivalent. ~~If `T1` and `T2` cannot be made equivalent, the program is ill-formed.~~ [*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 C<U>::assoc are the same type
}
```

— *end example*]

7 ~~A concept requirement that refers to the `DerivedFrom` concept ([concept.support]) is a *derivation requirement*. A derivation requirement is satisfied when both concept arguments are class types and the first concept argument is either equal to or publicly and unambiguously derived from the second concept argument. [*Example:* — *end example*]~~

8 A *requirement* followed by an ellipsis is a pack expansion (14.5.3). Requirement pack expansions place requirements on all of the arguments in one or more template parameter packs. [*Example:*

```
auto concept OutputStreamable<typename T> {
  std::ostream& operator<<(std::ostream&, const T&);
}

template<typename T, typename... Rest>
requires OutputStreamable<T> && OutputStreamable<Rest>...
void print(const T& t, const Rest&... rest) {
  std::cout << t;
  print(rest);
}
```

```
template<typename T>
requires OutputStreamable<T>
void print(const T& t) {
  std::cout << t;
}

void f(int x, float y) {
  print(17, ", ", 3.14159); // okay: implicitly-generated OutputStreamable<int>, OutputStreamable<const char[3]>,
                            // and OutputStreamable<double>
  print(17, " ", std::cout); // error: no concept map OutputStreamable<std::ostream>
}
```

— end example]

9 If the requirements of a template are inconsistent, such that no set of template arguments can satisfy all of the requirements, the program is ill-formed, no diagnostic required. [*Example:*

```
concept C<typename T> { }

template<typename T>
requires C<T> && !C<T>
void f(const T&); // error: no type can satisfy both C<T> && !C<T>, no diagnostic required
```

— end example]

14.10.1.1 Requirement implication [temp.req.impl]

1 The declaration of a constrained template implies additional template requirements that are available within the body of the template. A requirement is *implied* if the absence of that requirement would render the constrained template declaration ill-formed. Template requirements are implied from:

- the type of a constrained function template,
- the types named by an *exception-specification* (if any) of a constrained function template,
- the template arguments of a constrained class template partial specialization,
- the template arguments of a concept map template,
- the template parameters of a constrained template,
- the requirements of a constrained template (including implied requirements),
- the associated requirements of a concept, and
- the type of an associated function requirement.

2 For every concept requirement in a template's requirements (including implied requirements), requirements for the refinements and associated requirements of the concept named by the ~~*concept-id*~~concept instance (14.9.3, 14.9.1.3) are implied.

3 The formation of types within the declaration of a constrained template implies the template requirements needed to ensure that the types themselves are well-formed within any instantiation. The following type constructions imply template requirements:

— For every *template-id* `X<A1, A2, ..., AN>`, where `X` is a constrained template, the requirements of `X` (after substitution of the arguments `A1`, `A2`, ..., `AN`) are implied. [*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]

— If the formation of a type containing an archetype `T` will be ill-formed due to a missing requirement `C<T>`, where `C` is a compiler-supported concept ([concept.support]), the requirement `C<T>` is implied. [*Example:*

```
concept C<typename T> { typename assoc; }

template<typename T>
requires C<T>
C<T>::assoc  // implies Returnable<C<T>::assoc>
f(T*, T&); // implies PointeeType<T> and ReferentType<T>
```

— end example]

— For every *qualified-id* that names an associated type or template, a concept requirement for the concept instance containing that associated type or template is implied. [*Example:*

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

template<CopyConstructible T, CopyConstructible U>
Addable<T, U>::result_type // implies Addable<T, U>
add(T t, U u) {
  return t + u;
}
```

— For every type archetype `T` that is the type of a parameter in a function type, the requirement `MoveConstructible<T>` is implied. *— end example*]

4 In the definition of a class template partial specialization, the requirements of its primary class template (14.5.5), after substitution of the template arguments of the class template partial specialization, are implied. [*Note:* this rule ensures that a class template partial specialization of a constrained template is a constrained template, even if does not have template requirements explicitly specified. *— end note*] If this substitution results in a requirement that does not depend on any template parameter, then the requirement must be satisfied (14.10.1); otherwise, the program is ill-formed. [*Example:*

```
template<typename T>
requires EqualityComparable<T>
class simple_set { };

template<typename T>
class simple_set<T*> // implies EqualityComparable<T*>
{
};
```

— *end example*]

5 The template requirements for two templates are *identical* if they contain the same concept, negative, and same-type~~, and derivation~~ requirements in arbitrary order. Two requirements are the same if they have the same kind, name the same concept, and have the same template argument lists.

14.10.2 Archetypes [temp.archetype]

1 An *archetype* is a ~~class~~non-dependent type, template, or value whose behavior is defined by the template requirements (14.10.1) of its constrained template. Within a constrained context (14.10), a template parameter behaves as if it were its archetype. [*Note:* this substitution of archetypes (which are not dependent) for their corresponding types, templates, or values (which would be dependent in an unconstrained template) effectively treats all types and templates (and therefore both expressions and names) in a constrained ~~template~~context as "non-dependent". — *end note*]

2 The archetype of a type is a type, the archetype of a template is a class template, and the archetype of a value is a value.

3 A type in a constrained template ~~has~~aliases an archetype if it is:

- a template type parameter (14.1),
- an associated type (14.9.1.2), or
- ~~a class template specialization whose *template-name* is a template template parameter (14.1) or an associated template (14.9.1.2), or~~
- a class template specialization involving one or more archetypes

4 A template in a constrained template aliases an archetype if it is:

- a template template parameter (14.1) or
- an associated template (14.9.1.2).

5 A value in a constrained template aliases an archetype if it is a *constant-expression* (5.19) whose value depends on a template parameter.

6 If two types, `T1` and `T2`, both ~~have~~alias archetypes and are ~~equivalent~~the same (e.g., due to one or more same-type requirements (14.10.1)), then `T1` and `T2` ~~have~~alias the same archetype `T′`. [*Note:* there is no mechanism to specify the relationships between different value archetypes, because such a mechanism would introduce the need for equational reasoning within the translation process. — *end note*]

7 An archetype `T′` for a type `T` is

- an object type ([intro.object]), if the template contains the requirement `ObjectType<T>`,

— a class type (clause 9), if the template contains the requirement `ClassType<T>`,

— a class (clause 9), if the template contains the requirement `Class<T>`,

— a union ([class.union]), if the template contains the requirement `Union<T>`,

— a trivial type (3.9), if the template contains the requirement `TrivialType<T>`,

— a standard layout type (3.9), if the template contains the requirement `StandardLayoutType<T>`,

— a literal type (3.9), if the template contains the requirement `LiteralType<T>`,

— a scalar type (3.9), if the template contains the requirement `ScalarType<T>`,

— an integral type ([basic.fundamental]), if the template contains the requirement `IntegralType<T>`, and

— an enumeration type ([dcl.enum]), if the template contains the requirement `EnumerationType<T>`.

8 The archetype `T′` of `T` contains a public member function or member function template corresponding to each member function or member function template of each ~~concept instance~~concept map archetype corresponding to a concept requirement that names `T` (14.10.1). [*Example:*

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

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

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

— *end example*]

9 ~~If no requirement specifies a default constructor for a type `T`, a default constructor is not implicitly declared (12.1) for the archetype of `T`.~~

10 If no requirement specifies a copy constructor for a type `T`, a copy constructor is implicitly declared (12.8) in the archetype of `T` with the following signature:

```
T(const T&) = delete;
```

[*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 deleted
}
```

— end example]

11 If no requirement specifies a copy assignment operator for a type `T`, a copy assignment operator is implicitly declared (12.8) in the archetype of `T` with the following signature:

```
T& T::operator=(const T&) = delete;
```

12 If no requirement specifies a destructor for a type `T`, a destructor is implicitly declared (12.4) in the archetype of `T` with the following signature:

```
~T() = delete;
```

13 If no requirement specifies a unary `&` operator for a type `T`, a unary member operator `&` is implicitly declared in the archetype of `T` for each *cv* that is a valid *cv-qualifier-seq*:

```
cv T* operator&() cv = delete;
```

14 For each of the allocation functions `new`, `new[]`, `delete`, and `delete[]` ([class.free]), if no requirement specifies the corresponding operator with a signature below, that allocation function is implicitly declared as a member function in the archetype T' of `T` with the corresponding signature from the following list:

```
static void* T′::operator new(std::size_t) = delete;
static void* T′::operator new(std::size_t, void*) = delete;
static void* T′::operator new(std::size_t, const std::nothrow_t&) throw() = delete;
static void* T′::operator new[](std::size_t) = delete;
static void* T′::operator new[](std::size_t, void*) = delete;
static void* T′::operator new[](std::size_t, const std::nothrow_t&) throw() = delete;
static void T′::operator delete(void*) = delete;
static void T′::operator delete(void*, void*) = delete;
static void T′::operator delete(void*, const std::nothrow_t&) throw() = delete;
static void T′::operator delete[](void*) = delete;
static void T′::operator delete[](void*, void*) = delete;
static void T′::operator delete[](void*, const std::nothrow_t&) throw() = delete;
```

15 If the template requirements contain a ~~derivation~~ requirement `DerivedFrom<T, Base>`, then the archetype of `T` is publicly derived from the archetype of `Base`. If the same `DerivedFrom<T, Base>` requirement occurs more than once within the template requirements, the repeated `DerivedFrom<T, Base>` requirements are ignored.

16 If two associated member function or member function template requirements that name a constructor or destructor for a type `T` have the same signature~~, and the return types are equivalent~~, the duplicate signature is ignored. ~~If the return types are not the same, the program is ill-formed.~~

17 ~~If the processing of a constrained template definition requires the instantiation of a template whose template argument list contains a type `T` with an archetype `T′` or whose template `U` has an archetype `U′`, the template is instantiated (14.7) with the archetype `T′` substituted for each occurrence of `T`.~~If a class template specialization is an archetype that does not appear as a template argument of any explicitly-specified requirement in the template requirements and whose template is not itself an archetype, then the archetype is an instantiated archetype. An *instantiated archetype* is an archetype whose definition is provided by the instantiation of its template with its template arguments (which involve archetypes). The template shall not be an unconstrained template. [*Note:* partial ordering of class template partial specializations (14.5.5.2) will depend on the properties of the archetypes, as defined by the requirements of the constrained template. When the constrained template is instantiated (14.10.3), partial ordering of class template partial specializations will occur a second time based on the actual template arguments. — *end note*] [*Example:*

```
template<EqualityComparable T>
struct simple_multiset {
  bool includes(const T&);
  void insert(const T&);
  // ...
};

template<LessThanComparable T>
struct simple_multiset<T> { // A
  bool includes(const T&);
  void insert(const T&);
  // ...
};

template<LessThanComparable T>
bool first_access(const T& x) {
  static simple_multiset<T> set; // instantiates simple_multiset<T′>, where T′ is the archetype of T ,
                                 // from the partial specialization of simple_multiset marked 'A'
  return set.includes(x)? false : (set.insert(x), true);
}
```

— *end example*]

[*Note:* class template specializations for which template requirements are specified behave as normal archetypes. — *end note*] [*Example:*

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

template<CopyConstructible T> struct vector;

auto concept VectorLike<typename X> {
  typename value_type = typename X::value_type;
  X::X();
```

```
  void X::push_back(const value_type&);
  value_type& X::front();
}

template<CopyConstructible T>
requires VectorLike<vector<T>> // vector<T> is an archetype (but not an instantiated archetype)
void f(const T& value) {
  vector<T> x; // okay: default constructor in VectorLike<vector<T> >
  x.push_back(value); // okay: push_back in VectorLike<vector<T> >
  VectorLike<vector<T>::value_type& val = x.front(); // okay: front in VectorLike<vector<T> >
}
```

— end example]

18 [*Note:* constrained class templates involving recursive definitions are ill-formed if the recursive class template specialization is an instantiated archetype. Constrained class templates involving recursive definitions can be specified by adding template requirements on the recursive class template specializations, making them archetypes that are not instantiated archetypes. [*Example:*

```
template<CopyConstructible... T> class tuple;

template<CopyConstructible Head, CopyConstructible... Tail>
class tuple<Head, Tail...> : tuple<Tail...> // ill-formed: tuple<Tail...> is an instantiated archetype,
                                            // but it is an incomplete type
{
  Head head;
  // ...
};

template<> class tuple<> { /* ... */ };
```

— end example] *— end note*]

19 In a constrained template, for each concept requirement that is stated in or implied by the template requirements, a ~~*concept instance*~~*concept map archetype* for that requirement is synthesized by substituting the archetype of `T` for each occurrence of `T` within the concept arguments of the requirement. The ~~concept instance~~concept map archetype acts as a concept map, and is used to resolve name lookup into requirements scope (3.3.9) and satisfy the requirements of templates used inside the definition of the constrained template. ~~Concept instances act as concept maps~~ [*Example:*

```
concept SignedIntegral<typename T> {
  T::T(const T&);
  T operator-(T);
}
concept RandomAccessIterator<typename T> {
  SignedIntegral difference_type;
  difference_type operator-(T, T);
}
template<SignedIntegral T> T negate(const T& t) { return -t; }

template<RandomAccessIterator Iter>
```

```
RandomAccessIterator<Iter>::difference_type distance(Iter f, Iter l) {
  typedef RandomAccessIterator<Iter>::difference_type D;
  D dist = f - l;   // okay: - operator resolves to synthesized operator- in
                    // the concept map archetype RandomAccessIterator<Iter'>,
                    // where Iter' is the archetype of Iter
  return negate(dist); // okay, concept map archetype RandomAccessIterator<Iter'>
                       // implies the concept map archetype SignedIntegral<D'>,
                       // where D' is the archetype of D
```

— *end example*]

14.10.3 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 Instantiation of a constrained template also replaces each ~~concept instance associated with the template requirements~~concept map archetype with the concept map that satisfied the corresponding template requirement. [*Note:* Concept members that had resolved to members of the ~~concept instance~~concept map archetype now refer to members of the corresponding concept maps. — *end note*]

3 If a concept requirement appears (directly or indirectly) multiple times in the requirements of a constrained template, the program is ill-formed if the concept maps used to satisfy the multiple occurrences of the concept requirement are not the same concept map. [*Example:*

```
concept A<typename T> { }
concept B<typename T> {
  typename X;
  requires A<X>;
}
concept C<typename T> {
  typename X;
  requires A<X>;
}
namespace N1 {
  concept_map A<int> { }  // #1
  concept_map B<int> { }  // uses #1 to satisfy the requirement for A<int>
}
namespace N2 {
  concept_map A<int> { }  // #2
  concept_map C<int> { }  // uses #2 to satisfy the requirement for A<int>
}
tempate<typename T> requires B<T> && C<T>
struct S { };
using N1::concept_map B<int>;
using N2::concept_map C<int>;
S<int> s; // ill-formed, two different concept maps for A<int>, #1 and #2
```

— *end example*]

4 In the instantiation of a constrained template, a call to a function or a use of an operator that resolves to an associated function in a ~~concept instance~~concept map archetype (14.10.2) will be instantiated with a call to the corresponding associated function definition in the concept map that satisfies the concept requirement (14.10.1). [*Example:*

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

template<LessThanComparable T>
const T& min(const T& x, const T& y) {
  return x < y? x : y; // < resolves to LessThanComparable<T'>::operator<
                       // uses LessThanComparable<X>::operator< when instantiated with T=X
}

struct X { int member; };

concept_map LessThanComparable<X> {
  bool operator<(X x1, X x2) {
    return x1.member < x2.member;
  }
}

void f(X x1, X x2) {
  min(x, y); // okay: concept map LessThanComparable<X> satisfies requirement
             // for LessThanComparable<T> (with T = X).
}
```

— *end example*]

5 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 the seed function and all functions in the same scope as the seed function that

— succeed at template argument deduction (14.8.2),

— have the same name as the seed function,

— have the same signature and return type as the seed function after substitution of the template arguments (ignoring the template requirements),

— are more specialized (14.5.6.1) than the seed function.

Partial ordering of function templates (14.5.6.1) determines which of the function templates in the candidate set will be called in the instantiation of the constrained template. If partial ordering does not provide a unique answer, the program is ill-formed. [*Example:*

```
concept InputIterator<typename Iter> {
  typename difference_type;
}
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*> {
  typedef std::ptrdiff_t difference_type;
}

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]

The concept maps that are used in this partial ordering include the concept maps that 1) have replaced the ~~concept instances~~concept map archetypes used in the first partial ordering, 2) are in the same namespace as the concept maps from 1), 3) are in the associated namespaces ([basic.lookup.argdep]) of the concept map arguments for the concept maps from 1), or 4) are in the namespace of which the concept is a member. [*Example:*

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

namespace N1 {
  concept_map C<int> { }
  concept_map D<int> { }
}
namespace N2 {
  template<C T> void f(T);                 //#1
  template<C T> requires D<T> void f(T);   //#2
  template<C T> void g(T x) {
    f(x);
  }
  using N1::concept_map C<int>;
  void h() {
    g(1); // inside, g, the call to f goes to #2
  }
}
```

— end example]

6 In the instantiation of a constrained template, a template specialization whose template arguments involve archetypes (14.10.2) will be replaced by the template specialization that results from replacing each occurrence of an archetype with its corresponding type. The resulting template specialization (call it `A<X>`) shall be compatible with the template

specialization involving archetypes (call it `A<T′>`) that it replaced, otherwise the program is ill-formed. The template specializations are compatible if all of the following conditions hold:

— for each function, function template, or data member `m` of `A<T′>` referenced by the constrained template, there exists a member named `m` in `A<X>` that is accessible from the constrained template and whose type, storage specifiers, template parameters (if any), and template requirements (if any) are the same as the those of `A<T′>::m` after replacing the archetypes with their actual template argument types.

— for each member type `t` of `A<T′>` referenced by the constrained template, there exists a member type `t` in `A<X>` that is accessible from the constrained template and is compatible with the member type `A<T′>::t` as specified herein.

— for each base class `B′` of `A<T′>` referenced by a derived-to-base conversion ([conv.ptr]) in the constrained template, there exists an unambiguous base class `B` of `A<X>` that is accessible from the constrained template, where `B` is the type produced by replacing the archetypes in `B′` with their template argument types.

[*Example:*

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

template<CopyConstructible T>
struct vector { // A
  vector(int, T const &);
  T& front();
};

template<typename T>
struct vector<T*> { // B
  vector(int, T* const &);
  T*& front();
};

template<>
struct vector<bool> { // C
  vector(int, bool);
  bool front();
};

template<CopyConstructible T>
void f(const T& x) {
  vector<T> vec(1, x);
  T& ref = vec.front();
}

void g(int i, int* ip, bool b) {
  f(i); // okay: instantiation of f<int> uses vector<int>, instantiated from A
  f(ip); // okay: instantiation of f<int*> uses vector<int*>, instantiated from B
  f(b); // ill-formed, detected in the instantiation of f<bool>, which uses the vector<bool> specialization C:
        // vector<bool>::front is not compatible with vector<T>::front (where T=bool)
```

```
}
```

— end example]

7 ~~In the instantiation of a constrained template, the use of a member of an archetype (14.10.2) instantiates to a use of the corresponding member in the type that results from substituting the template arguments from the instantiation into the type corresponding to the archetype.~~

Appendix B
(informative)
Implementation quantities [implimits]

Add the following bullet to paragraph 2 :

— Recursively nested implicit concept map definitions [1024]

Acknowledgments

The effort to introduce concepts into C++ has been shaped by many. The authors of the "Indiana" and "Texas" concepts proposals have had the most direct impact on concepts: Gabriel Dos Reis, Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. Other major contributors to the introduction of concepts in C++ include David Abrahams, Matthew Austern, Mat Marcus, David Musser, Sean Parent, Sibylle Schupp, Alexander Stepanov, and Marcin Zalewski. Howard Hinnant helped introduce support for rvalue references. Stephen Adamczyk, Daniel Krügler, Jens Maurer, John Spicer, and James Widman provided extremely detailed feedback on various drafts and prior revisions of this wording, and the wording itself has benefited greatly from their efforts and the efforts of the C++ committee's Core Working Group.

Bibliography

[1] 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.

[2] D. Gregor, B. Stroustrup, J. Siek, and J. Widman. Proposed wording for concepts (revision 3). Technical Report N2421=07-0281, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, October 2007.