Document Number:	N4333
Date:	2014-11-21
Revises:	N4205
Reply to:	Andrew Sutton
	University of Akron
	asutton@uakron.edu

Working Draft, C++ extensions for Concepts

Note: this is an early draft. It's known to be incomplet and incorrekt, and it has lots of bad formatting.

Contents

List of Tables

List of Figures

1 General [intro]

1.1 Scope [intro.scope]

1 This Technical Specification describes extensions to the C++ Programming Language 1.2 that enable the specification and checking of constraints on template arguments, and the ability to overload functions and specialize class templates based on those constraints. These extensions include new syntactic forms and modifications to existing language semantics.

2 The International Standard, ISO/IEC 14882, provides important context and specification for this Technical Specification. This document is written as a set of changes against that specification. Instructions to modify or add paragraphs are written as explicit instructions. Modifications made directly to existing text from the International Standard use underlining to represent added text and ~~strikethrough~~ to represent deleted text.

3 WG21 paper N4919 defines "fold expressions", which are used to define constraint expressions resulting from the use of *constrained-type-specifier*s in parameter packs (14.10.6). However, this feature is not present in ISO/IEC 14882. That paper is currently under consideration for inclusion in the next version of the C++ Programming Language. After reivew, the relevant wording will be incorporated into this Technical Specification.

1.2 Normative references [intro.refs]

1 The following referenced document is indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

(1.1) — ISO/IEC 14882:2014, *Programming Languages – C++*

ISO/IEC 14882:2014 is herein after called the *C++ Standard*.

1.3 Terms and definitions [intro.defs]

Modify the definitions of "signature" to include associated constraints (Clause 14).

1.3.1 [defns.signature]

signature

<function> name, parameter type list (8.4.1), ~~and~~ enclosing namespace (if any), and any associated constraints (Clause 14)

[*Note:* Signatures are used as a basis for name mangling and linking. — *end note*]

1.3.2 [defns.signature.templ]

signature

<function template> name, parameter type list (8.4.1), enclosing namespace (if any), return type, ~~and~~ template parameter list, and any associated constraints (Clause 14)

1.3.3 [defns.signature.member]

signature

<class member function> name, parameter type list (8.4.1), class of which the function is a member, *cv*-qualifiers (if any), ~~and~~ *ref-qualifier* (if any), and any associated constraints (Clause 14)

1.3.4 **[defns.signature.member.templl]**
signature
<class member function template> name, parameter type list (8.4.1), class of which the function is a member, *cv*-qualifiers (if any), *ref-qualifier* (if any), return type, ~~and~~ template parameter list, and any associated constraints (Clause 14)

1.4 Implementation compliance [intro.compliance]

1 Conformance requirements for this specification are the same as those defined in 1.4 in the C++ Standard. [*Note:* Conformance is defined in terms of the behavior of programs. — *end note*]

1.5 Acknowledgments [intro.ack]

1 The design of this specification is based, in part, on a concept specification of the algorithms part of the C++ standard library, known as "The Palo Alto" report (WG21 N3351), which was developed by a large group of experts as a test of the expressive power of the idea of concepts. Despite syntactic differences between the notation of the Palo Alto report and this Technical Specification, the report can be seen as a large-scale test of the expressiveness of this Technical Specification.

2 This work was funded by NSF grant ACI-1148461.

2 Lexical conventions [lex]

2.1 Keywords [lex.key]

In 2.1, add the keywords `concept` and `requires` to Table 4.

5 Expressions [expr]

Modify paragraph 8 to include a reference to *requires-expression*s.

1 In some contexts, unevaluated operands appear (5.2.8, 5.3.3, 5.3.7, 5.1.3).

This is a test 5.2.8

5.1 Primary expressions [expr.prim]

5.1.1 General [expr.prim.general]

In this section, add the *requires-expression* to the rule for *primary-expression*.

primary-expression:
requires-expression

In paragraph 8, add `auto` to *nested-name-specifier*:

8 *nested-name-specifier:*
`auto ::`

Add a new paragraph after paragraph 11:

12 In a *nested-name-specifier* of the form `auto::`, the `auto` specifier is a placeholder for a type to be deduced (7.1.6.4).

Add a new paragraph after paragraph 13:

14 If an *id-expression* denotes a non-overloaded function declaration that was declared with a *requires-clause* (8.4.1), its associated constraints shall be satisfied (14.10). [*Example:*

```
void f(int) requires false;

f(0);                 // error: constraints not satisfied
void (*p)(int) = f;   // error: constraints not satisfied
```

— *end example*]

5.1.2 Lambda expressions [expr.prim.lambda]

Insert the following paragraph after paragraph 4 to define the term "generic lambda".

5 A *generic lambda* is a *lambda-expression* where either the `auto` *type-specifier* (7.1.6.4) or a *constrained-type-specifier* (7.1.6.5) appears in a parameter type of the *lambda-declarator*.

Modify paragraph 5 so that the meaning of a generic lambda is defined in terms of its abbreviated member function template call operator.

The closure type for a non-generic *lambda-expression* has a public inline function call operator (13.5.4) whose parameters and return type are described by the *lambda-expression*'s *parameter-declaration-clause* and *trailing-return-type*, respectively. ~~For a generic lambda, the closure type has a public inline function call operator member template (14.5.2) whose *template-parameter-list* consists of one invented type *template-parameter* for each occurrence of `auto` in the lambda's~~

~~*parameter-declaration-clause*, in order of appearance. The invented type *template-parameter* is a parameter pack if the corresponding *parameter-declaration declares* a function parameter pack (8.4.1). The return type and function parameters of the function call operator template are derived from the *lambda-expression*'s *trailing-return-type* and *parameter-declaration-clause* by replacing each occurrence of `auto` in the *decl-specifiers* of the *parameter-declaration-clause* with the name of the corresponding invented template-parameter.~~ For a generic lambda, the function call operator (13.5.4) is an abbreviated function template (8.4.1).

Add the following example after those in paragraph 5 in the original document.

[*Example:*

```
template<typename T> concept bool C = true;

auto gl = [](C& a, C* b) { a = *b }; // OK: denotes a generic lambda

struct Fun {
  auto operator()(C& a, C* b) const { a = *b; }
} fun;
```

`C` is a *constrained-type-specifier*, signifying that the lambda is generic. The generic lambda `gl` and the function object `fun` have equivalent behavior when called with the same arguments. — *end example*]

5.1.3 Requires expressions [expr.prim.req]

Add this section to 5.1.

1 A *requires-expression* provides a concise way to express syntactic requirements on template arguments. A syntactic requirement is one that can be checked by name lookup (3.4) or by checking properties of types and expressions.

requires-expression:
  `requires` *requirement-parameter-list requirement-body*

requirement-parameter-list:
  `(` *parameter-declaration-clause$_{opt}$* `)`

requirement-body:
  `{` *requirement-list* `}`

requirement-list:
  requirement
  requirement-list requirement

requirement:
  simple-requirement
  type-requirement
  compound-requirement
  nested-requirement

2 A *requires-expression* defines a conjunction of the constraints (14.10) introduced by its *requirements*.

3 A *requires-expression* has type `bool` and is an unevaluated operand (5).

4 A *requires-expression* shall appear only within a concept definition (7.1.6.5), or a *requires-clause* following a *template-parameter-list* (14) or function declaration (8.4.1).

[*Example:* A common use of *requires-expressions* is to define syntactic requirements in concepts such as the one below:

```
template<typename T>
  concept bool R() {
    return requires (T i) {
      typename A<T>;
      {*i} -> const A<T>&;
    };
  }
```

A *requires-expression* can also be used in a *requires-clause* templates as a way of writing ad hoc constraints on template arguments such as the one below:

```
template<typename T>
  requires requires (T x) { x + x; }
    T add(T a, T b) { return a + b; }
```

The first `requires` introduces the *requires-clause*, and the second introduces the *requires-expression*. — *end example*] [*Note:* Such requirements can also be written using by defining them within a concept.

```
template<typename T>
  concept bool C = requires (T x) { x + x; };

template<typename T> requires C<T>
  T add(T a, T b) { return a + b; }
```

— *end note*]

5 A *requires-expression* may introduce local parameters using a *parameter-declaration-clause* (8.4.1). A local parameter of a *requires-expression* shall not have a default argument. Each name introduced by a local parameter is in scope from the point of its declaration until the closing brace of the *requirement-body*. These parameters have no linkage, storage, or lifetime; they are only used as notation for the purpose of defining *requirement*s. The *requirement-parameter-list* shall not include an ellipsis.

6 The *requirement-body* is comprised of a sequence of *requirements*. These *requirements* may refer to local parameters, template parameters, and any other declarations visible from the enclosing context. Each *requirement* appends one or more atomic constraints (14.10) to the conjunction of constraints defined by the *requires-expression*s.

7 The substitution of template arguments into a *requirement* may result in an invalid type or expression, in which case the constraint is not satisfied; it does not cause the program to be ill-formed. [*Note:* But if the substitution of template arguments into a *requirement* would always result in a substitution failure, the program is ill-formed; no diagnostic required (14.7). — *end note*] [*Example:*

```
template<typename T> concept bool C =
  requires () {
    new int[(int)-sizeof(T)]; // ill-formed, no diagnostic required
  };
```

— *end example*]

5.1.3.1 Simple requirements [expr.prim.req.simple]

simple-requirement: *expression* ;

1 A *simple-requirement* introduces an expression constraint (14.10.2.2) for its *expression*.

[*Example:*

```
template<typename T> concept bool C =
  requires (T a, T b) {
    a + b;  // a simple requirement
  };
```

— end example]

5.1.3.2 Type requirements [expr.prim.req.type]

type-requirement:
typename-specifier ;

1 A *type-requirement* introduces type constraint (14.10.2.3) for the type named by its *typename-specifier*. [*Note:* A type requirement asserts the validity of an associated type, either as a member type, a class template specialization, or an alias template. It is not used to specify requirements for arbitrary *type-specifiers*. *— end note*] [*Example:*

```
template<typename T> struct S { };
template<typename T> using Ref = T&;

template<typename T> concept bool C =
  requires () {
    typename T::inner; // required nested type name
    typename S<T>;     // required class template specialization
    typename Ref<T>;   // required alias template substitution
  };
```

— end example]

5.1.3.3 Compound requirements [expr.prim.req.compound]

compound-requirement:
{ *expression* } noexcept$_{opt}$ *trailing-return-type*$_{opt}$;

1 A *compound-requirement* introduces a conjunction of one or more atomic constraints for the *expression* E:

(1.1) — the *compound-requirement* introduces an expression constraint for E (14.10.2.2);

(1.2) — if the the noexcept specifier is present, the *compound-requirement* appends an exception constraint for E (14.10.2.6);

(1.3) — if the *trailing-return-type* is given and its type T contains no placeholders (7.1.6.4, 7.1.6.5), the requirement appends two constraints to the conjunction of constraints: a type constraint on the formation of T (14.10.2.3) and an implicit conversion constraint from E to T (14.10.2.4). Otherwise, if T contains placeholders, an argument deduction constraint (14.10.2.5) of E against the type T is appended to the conjunction of constraints.

[*Example:*

```
template<typename T> concept bool C1 =
  requires(T x) {
    {x++};
  };
```

The *compound-requirement* in C1 introduces an expression constraint on x++. It is equivalent to a *simple-requirement* with the same *expression*.

```
template<typename T> concept bool C2 =
  requires(T x) {
    {*x} -> typename T::inner;
  };
```

The *compound-requirement* in `C2` introduces three constraints: an expression constraint for `*x`, a type constraint for `typename T::inner`, and a conversion constraint requiring `*x` to be implicitly convertible to `typename T::inner`.

```
template<typename T> concept bool C3 =
  requires(T x) {
    {g(x)} noexcept;
  };
```

The *compound-requirement* in `C3` introduces two constraints: an expression constraint for `g(x)` and an exception constraint on `g(x)`.

```
template<typename T> concept bool C() { return true; }

template<typename T> concept bool C5 =
  requires(T x) {
    {f(x)} -> const C&;
  };
```

The *compound-requirement* in `C5` introduces two constraints: expression constraint for `f(x)`, and a deduction constraint requiring that overload resolution succeeds for the call `g(f(x))` where `g` is the following invented abbreviated function template.

```
void g(const C&);
```

— *end example*]

5.1.3.4 Nested requirements [expr.prim.req.nested]

nested-requirement:
  requires-clause ;

1 A *nested-requirement* can be used to specify additional constraints in terms of local parameters. A *nested-requirement* appends the normalized form (14.10.4) of its *constraint-expression* to the conjunction of constraints introduced by its enclosing *requires-expression*.

[*Example:*

```
template<typename T> concept bool C() { return sizeof(T) == 1; }

template<typename T> concept bool D =
  requires (T t) {
    requires C<decltype (+t)>();
  };
```

The *nested-requirement* appends the predicate constraint `sizeof(decltype (+t)) == 1` (14.10.2.1). — *end example*]

7 Declarations [dcl.dcl]

7.1 Specifiers [dcl.spec]

Extend the *decl-specifier* production to include the `concept` specifier.

decl-specifier:
`concept`

7.1.6 Type specifiers [dcl.type]

7.1.6.2 Simple type specifiers [dcl.type.simple]

Add *constrained-type-specifier* to the grammar for *simple-type-specifier*s.

simple-type-specifier:
constrained-type-specifier

7.1.6.4 `auto` specifier [dcl.spec.auto]

Modify paragraph 1 to extend the use of `auto` to designate abbreviated function templates.

1 The `auto` and `decltype(auto)` *type-specifier*s (as well as *constrained-type-specifier*s, 7.1.6.5) designate ~~a~~ placeholder ~~type~~types that will be replaced later, either by deduction from an initializer or by explicit specification with a *trailing-return-type*. The `auto` *type-specifier* is also used to signify that a lambda is a generic lambda (5.1.2) or that a function declaration is an abbreviated function template (8.4.1).

[*Example:*

```
struct N {
  template<typename T> struct Wrap;
  template<typename T> static Wrap<T> make_wrap(T);
};
template<typename T, typename U> struct Pair;
template<typename T, typename U> Pair<T, U> make_pair(T, U);

template<int N> struct Size { void f(int) { }  };
Size<0> s;
bool g(char, double);

void (auto::*)(auto) p = &Size<0>::f;    // OK
N::Wrap<auto> a = N::make_wrap(0.0);     // OK
Pair<auto, auto> p = make_pair(0, 'a'); // OK
auto::Wrap<int> x = N::make_wrap(0);     // error: failed to deduce value for auto
Size<sizeof(auto)> y = s;                // error: failed to deduce value for auto
```

— *end example*]

Modify paragraph 2, allowing multiple occurrences of `auto` in those contexts where it is valid.

2 ~~The~~A placeholder type can appear with a function declarator in the *decl-specifier-seq*, *type-specifier-seq*, *conversion-function-id*, or *trailing-return-type*, in any context where such a declarator is valid. If the function declarator includes a *trailing-return-type* (8.4.1), that specifies the declared return type of the function. If the declared return type of the function contains a placeholder type, the return type of the function is deduced from return statements in the body of the function, if any.

Modify paragraph 3 to allow the use of `auto` within the parameter type of a lambda or function.

3 If the `auto` *type-specifier* appears ~~as one of the *decl-specifiers* in the *decl-specifier-seq* of a *parameter-declaration*~~ in a parameter type of a *lambda-expression*, the lambda is a generic lambda (5.1.2). [*Example:*

```
auto glambda = [](int i, auto a) { return i; }; // OK: a generic lambda
```

— *end example*] Similarly, if the `auto` *type-specifier* appears in a parameter type of a function declaration, the function declaration declares an abbreviated function template (8.4.1). [*Example:*

```
void f(const auto&, int); // OK: an abbreviated function template
```

— *end example*]

Add the following after paragraph 3 to allow the use of `auto` in the *trailing-return-type* of a *compound-requirement*.

4 If the `auto` *type-specifier* appears in the *trailing-return-type* of a *compound-requirement* in a *requires-expression* (5.1.3.3), that return type introduces an argument deduction constraint (14.10.2.5). [*Example:*

```
template<typename T> concept bool C() {
  return requires (T i) {
    {*i} -> const auto&; // OK: introduces an argument deduction constraint
  };
}
```

— *end example*]

Modify paragraph 4 to allow multiple uses of `auto` within certain declarations. Note that the examples in the original text are unchanged and therefore omitted.

5 The type of a variable declared using `auto` or `decltype(auto)` is deduced from its initializer. This use is allowed when declaring variables in a block (6.3), in namespace scope (3.3.6), and in a *for-init-statement* (6.5.3). ~~`auto` or `decltype(auto)` shall appear as one of the *decl-specifiers* in the *decl-specifier-seq*~~ `auto` can appear anywhere in the declared type of the variable, but `decltype(auto)` shall appear only as one of the *decl-specifiers* of the *decl-specifier-seq*. ~~and the~~The *decl-specifier-seq* shall be followed by one or more *init-declarators*, each of which shall have a non-empty initializer. In an initializer of the form

```
( expression-list )
```

the *expression-list* shall be a single *assignment-expression*.

Update the rules in paragraph 7 to allow deduction of multiple occurrences of `auto` in a declaration.

7 When a variable declared using a placeholder type is initialized, or a `return` statement occurs in a function declared with a return type that contains a placeholder type, the deduced return type or variable type is determined from the type of its initializer. In the case of a return

with no operand, the initializer is considered to be `void()`. Let `T` be the declared type of the variable or return type of the function. ~~If the placeholder is the `auto` *type-specifier*,~~ If `T` contains any occurrences of the `auto` *type-specifier*, the deduced type is determined using the rules for template argument deduction. If the deduction is for a return statement and the initializer is a *braced-init-list* (8.5.4), the program is ill-formed. ~~Otherwise, obtain `P` from `T` by replacing the occurrences of `auto` with either a new invented type template parameter `U` or, if the initializer is a *braced-init-list*, with `std::initializer_list<U>`.~~

Otherwise, obtain `P` from `T` as follows:

(7.1) — replace each occurrence of `auto` in the variable type with a new invented type template parameter, or

(7.2) — when the initializer is a *braced-init-list* and `auto` is a *decl-specifier* of the *decl-specifier-seq* of the variable declaration, replace that occurrence of `auto` with `std::initializer_list<U>` where `U` is an invented template type parameter.

Deduce a value for ~~`U`~~ each invented template type parameter in `P` using the rules of template argument deduction from a function call (14.8.2.1), where `P` is a function template parameter type and the initializer is the corresponding argument. If the deduction fails, the declaration is ill-formed. Otherwise, the type deduced for the variable or return type is obtained by substituting the deduced ~~`U`~~ values for each invented template parameter into `P`. [*Example:*

```
template<typename T> struct Vec { };
template<typename T> Vec<T> make_vec(std::initializer_list<T>) { return Vec<T>{}; } }

auto x1 = { 1, 2 };            // OK: decltype(x1) is std::initializer_list<int>
auto x2 = { 1, 2.0 };          // error: cannot deduce element type
```

— *end example*]

Add the following to the first example in paragraph 7.

[*Example:*

```
auto& x3 = *x1.begin();             // OK: decltype(x3) is int&
const auto* p = &x3;                // OK: decltype(p) is const int*
Vec<auto> v1 = make_vec({1, 2, 3}); // OK: decltype(v1) is Vec<int>
Vec<auto> v2 = {1, 2, 3};           // error: cannot deduce element type
```

— *end example*]

Add the following after the second example in paragraph 7.

[*Example:* In the following program

```
template<typename F, typename S> struct Pair;
template<typename T, typename U> Pair<T, U> make_pair(T, U);

struct S { void mfn(bool); } s;
int fn(char, double);

Pair<auto (*)(auto, auto), auto (auto::*)(auto)> p = make_pair(fn, &S::mfn);
```

the declared type of `p` is the deduced type of the parameter `x` in the call of `g(make_pair(fn, &S::mfn))` of the following invented function template:

```
template<class T1, class T2, class T2, class T3, class T4, class T5, class T6>
void g(Pair< T1(*)(T2, T3), T4 (T5::*)(T6)> x);
```

— *end example*]

7.1.6.5 Constrained type specifiers [dcl.spec.constr]

Add this section to 7.1.6.

1 Like the `auto` *type-specifier* (7.1.6.4), a *constrained-type-specifier* designates a placeholder that will be replaced later by deduction from the *expression* in a *compound-requirement* or a function argument. *constrained-type-specifier*s have the form

constrained-type-specifier:
  nested-name-specifier$_{opt}$ constrained-type-name

constrained-type-name:
  concept-name
  partial-concept-id

concept-name:
  identifier

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

A *constrained-type-specifier* may also designate placeholders for deduced non-type and template template arguments. [*Example:*

```
template<typename T> concept bool C1 = false;
template<int N> concept bool C2 = false;
template<template<typename> class X> C3 = false;

template<typename T, int N> class Array { };
template<typename T, template<typename> class A> class Stack { };
template<typename T> class Alloc { };

void f1(C1 c);             // C1 designates a placeholder type
void f2(Array<auto, C2>); // C2 designates a placeholder for an integer value
void f3(Stack<auto, C3>); // C3 designates a placeholder for a class template
```

— *end example*]

A *constrained-type-specifier* can appear in many of the same places as the `auto` *type-specifier*, is subject to the same rules, and has equivalent meaning in those contexts. In particular, a *constrained-type-specifier* can appear in the following contexts with the given meaning:

(1.1) — a parameter type of a function declaration, signifying an abbreviated function template (8.4.1);

(1.2) — a parameter of a lambda, signifying a generic lambda (5.1.2);

(1.3) — the parameter type of a template parameter, signifying a constrained template parameter (14.1);

(1.4) — the *trailing-return-type* of a *compound-requirement* (5.1.3.3), signifying an argument deduction constraint (14.10.2.5).

A program that includes a *constrained-type-specifier* in any other contexts is ill-formed. [*Example:*

```
template<typename T> concept bool C1 = true;
template<typename T, int N> concept bool C2 = true;
template<bool (*)(int)> concept bool C3 = true;

template<typename T> class Vec;
```

```
struct N {
  template<typename T> struct Wrap;
};
template<typename T, typename U> struct Pair;
template<bool (*)(int)> struct Pred;

auto gl = [](C1& a, C1* b) { a = *b; }; // OK: a generic lambda
void af(const Vec<C1>& x);              // OK: an abbreviated function template

void f1(N::Wrap<C1>);      // OK
void f2(Pair<C1, C2<0>>); // OK
void f3(Pred<C3>);         // OK

template<typename T> concept bool Iter() {
  return requires(T i) {
    {*i} -> const C1&; // OK: an argument deduction constraint
  };
}
```

The declaration of `f4` is valid, but a value can never be deduced for the placeholder designated by `C1` since it appears in a non-deduced context (14.8.2.5). However, a value may be explicitly given as a template argument in a *template-id*. — *end example*]

2 [*Note:* Unlike `auto`, a *constrained-type-specifier* cannot be used in the type of a variable declaration or the return type of a function. [*Example:*

```
template<typename T> concept bool C = true;
template<typename T, int N> concept bool D = true;

const C* x = 0;  // error: C used in a variable type
D<0> fn(int x);  // error: D<0> used as a return type
```

— *end example*] — *end note*]

3 An identifier is a *concept-name* if it refers to a set of concept definitions (7.1.7). [*Note:* The set of concepts has multiple members only when referring to a set of overloaded function concepts. There is at most one member of this set when a *concept-name* refers to a variable concept. — *end note*] [*Example:*

```
template<typename T> concept bool C() { return true; }             // #1
template<typename T, typename U> concept bool C() { return true; } // #2
template<typename T> concept bool D = true;                        // #3

void f(C); // OK: the concept-name C refers to both #1 and #2
void g(D); // OK: the concept-name D refers only to #3
```

— *end example*]

4 A *partial-concept-id* is a *concept-name* followed by a sequence of *template-arguments*. [*Example:*

```
template<typename T, int N = 0> concept bool Seq = true;

void f1(Seq<3>);
void f2(Seq<>);
```

— *end example*]

5 The concept designated by a *constrained-type-specifier* is the one selected by concept resolution (14.10.5).

7.1.7 concept specifier [dcl.spec.concept]

Add ths section to 7.1.

1 The `concept` specifier shall be applied only to the definition of a function template or variable template, declared in namespace scope (3.3.6). A function template definition having the `concept` specifier is called a *function concept*. A function concept is a non-throwing function (15.4). When a function is declared to be a concept, it shall be the only declaration of that function. A variable template definition having the `concept` specifier is called a *variable concept*. A *concept definition* refers to either a function concept and its definition or a variable concept and its initializer. [*Example:*

```
template<typename T>
  concept bool F1() { return true; } // OK: declares a function concept
template<typename T>
  concept bool F2();                 // error: function concept is not a definition
template<typename T> constexpr bool F3();
template<typename T>
  concept bool F3() { return true; } // error: redeclaration of a function as a concept
template<typename T>
  concept bool V1 = true;            // OK: declares a variable concept
template<typename T>
  concept bool V2;                   // error: variable concept with no initializer
struct S {
  template<typename T>
    static concept bool C = true;    // error: concept declared in class scope
};
```

— *end example*]

2 Every concept definition is implicitly defined to be a `constexpr` declaration (7.1.5).

3 A concept definition shall not be declared with the `friend` or `constexpr` specifiers.

4 Additionally, a concept definition shall not have associated constraints (Clause 14).

5 The definition of a function concept or the initializer of a variable concept shall not include a reference to the concept being declared. [*Example:*

```
template<typename T>
  concept bool F() { return F<typename T::type>(); } // error
template<typename T>
  concept bool V = V<T*>;                           // error
```

— *end example*]

6 The first declared template parameter of a concept definition is its *prototype parameter*. A *variadic concept* is a concept whose prototype parameter is a template parameter pack.

7 A function concept has the following restrictions:

(7.1) — No *function-specifier*s shall appear in its declaration (7.1.2).

(7.2) — The declared return type shall be the non-deduced type `bool`.

(7.3) — The declaration shall have a *parameter-declaration-clause* equivalent to `()`.

(7.4) — The function body shall consist of a single `return` statement whose *expression* shall be a *constraint-expression* (14.10.4).

[*Note:* Return type deduction requires the instantiation of the function definition, but concept definitions are not instantiated; they are normalized (14.10.4). — *end note*] [*Example:*

```
template<typename T>
  concept int F1() { return 0; }        // error: return type is not bool
template<typename T>
  concept auto F3(T) { return true; } // error: return type is deduced
template<typename T>
  concept bool F2(T) { return true; } // error: must have no parameters
```

— *end example*]

8 A variable concept has the following restrictions:

(8.1) — The declared type shall be `bool`.

(8.2) — The declaration shall have an initializer.

(8.3) — The initializer shall be a *constraint-expression*.

[*Example:*

```
template<typename T>
  concept bool V1 = 3 + 4; // error: initializer is not a constraint-expression
concept bool V3 = 0;       // error: not a template

template<typename T> concept bool C = true;

template<C T>
  concept bool V2 = true;  // error: constrained template declared as a concept
```

— *end example*]

9 A program shall not declare an explicit instantiation (14.8.2), an explicit specialization (14.8.3), or a partial specialization of a concept definition. [*Note:* This prevents users from subverting the constraint system by providing a meaning for a concept that differs from its original definition. — *end note*]

8 Declarators [dcl.decl]

Factor the grammar of *declarators* to allow the specification of constraints on function declarations.

declarator:
  ~~*ptr-declarator*~~
  ~~*noptr-declarator parameters-and-qualifiers trailing-return-type*~~
  basic-declarator requires-clause$_{opt}$

basic-declarator:
  ptr-declarator
  noptr-declarator parameters-and-qualifiers trailing-return-type

Add the following paragraph at the end of this section.

The optional *requires-clause* (Clause 14) in a *declarator* shall be present only when the declarator declares a function (8.4.1).

[*Example:*

```
void f1(int a) requires true;          // OK
auto f2(int a) -> bool requires true; // OK
void (*pf)() requires true;            // error: constraint on a variable
void g(int (*)() requires true);       // error: constraint on a parameter

auto* p = new void(*[3])(char) requires true; // error: not a function declaration
```

— *end example*]

8.3 Meaning of declarators [dcl.meaning]

8.4.1 Functions [dcl.fct]

Modify the matching condition in paragraph 1 to accept a *requires-clause*.

1 `D1 (` *parameter-declaration-clause* `)` *cv-qualifier-seq*$_{opt}$ *ref-qualifier*$_{opt}$ *exception-specification*$_{opt}$ *attribute-specifier-seq*$_{opt}$ *requires-clause*$_{opt}$

Modify the matching condition in paragraph 2 to accept a *requires-clause*.

2 `D1 (` *parameter-declaration-clause* `)` *cv-qualifier-seq*$_{opt}$ *ref-qualifier*$_{opt}$ *exception-specification*$_{opt}$ *attribute-specifier-seq*$_{opt}$ *trailing-return-type* *requires-clause*$_{opt}$

Modify the second sentence of paragraph 5. The remainder of this paragraph has been omitted.

5 A single name can be used for several different functions in a single scope; this is function overloading (Clause 13). All declarations for a function shall agree exactly in ~~both~~ the return type, ~~and~~ the parameter-type-list, and associated constraints, if any (Clause 14).

Modify paragraph 6 to exclude constraints from the type of a function. Note that the change occurs in the sentence following the example in the original document.

6 The return type, the parameter-type-list, the *ref-qualifier*, and the *cv-qualifier-seq*, but not the default arguments (8.3.6), constraints associated by an optional *requires-clause* (Clause 14), or the exception specification (15.4), are part of the function type.

Modify paragraph 15. Note that the footnote reference has been omitted.

15 There is a syntactic ambiguity when an ellipsis occurs at the end of a *parameter-declaration-clause* without a preceding comma. In this case, the ellipsis is parsed as part of the *abstract-declarator* if the type of the parameter either names a template parameter pack that has not been expanded or contains either `auto` or a *constrained-type-specifier* ; otherwise, it is parsed as part of the *parameter-declaration-clause*.

Add the following paragraphs after paragraph 15.

16 An *abbreviated function template* is a function declaration whose parameter-type-list includes one or more placeholders (7.1.6.4, 7.1.6.5). An abbreviated function template is equivalent to a function template (14.6.6) whose *template-parameter-list* includes one invented *template-parameter* for each occurrence of a placeholder in the *parameter-declaration-clause*, in order of appearance. If the placeholder is designated by the `auto` *type-specifier*, then the corresponding invented template parameter is a type *template-parameter*. Otherwise, the placeholder is designated by a *constrained-type-specifier*, and the corresponding invented parameter matches the type and form of the prototype parameter (7.1.7) of the concept designated by the *constrained-type-specifier* (14.10.5). The invented *template-parameter* is a parameter pack if the corresponding *parameter-declaration* declares a function parameter pack and the type of the parameter contains only one placeholder. If the prototype parameter of the designated concept declares a template parameter pack, the corresponding *parameter-declaration* shall declare a function parameter pack. The adjusted function parameters of an abbreviated function template are derived from the *parameter-declaration-clause* by replacing each occurrence of a placeholder with the name of the corresponding invented *template-parameter*. If the replacement of a placeholder with the name of a template parameter results in an invalid parameter declaration, the program is ill-formed.

[*Example:*

```
template<typename T> class Vec { };
template<typename T, typename U> class Pair { };

void f1(const auto&, auto);
void f2(Vec<auto*>...);
void f3(auto (auto::*)(auto));

template<typename T, typename U>
  void f1(const T&, U);          // redeclaration of f1(const auto&, auto)
template<typename... T>
  void f2(Vec<T*>...);           // redeclaration of f2(Vec<auto*>...)
template<typename T, typename U, typename V>
  void f3(T (U::*)(V));          // redeclaration of f3(auto (auto::*)(auto))

template<typename T> concept bool C1 = true;
template<typename T> concept bool C2 = true;
template<typename T, typename U> concept bool D = true;
template<<typename... Ts> concept bool E = true;

void g1(const C1*, C2&);
void g2(Vec<C1>&);
void g3(C1&...);
void g4(Vec<D<int>>);
void g5(E...); // OK
void g6(E);    // error: E does not declare a function parameter pack

template<C1 T, C2 U> void g1(const T*, U&); // redeclaration of g1(const C1*, C2&)
```

```
template<C1 T> void g2(Vec<T>&);            // redeclaration of g2(Vec<C1>&)
template<C1... Ts> void g3(Ts&...);         // redeclaration of g3(C1&...)
template<D<int> T> void g4(Vec<T>);         // redeclaration of g4(Vec<D<int>>)
```

— *end example*]

[*Example:*

```
template<int N> concept bool Num = true;

void h(Num*); // error: invalid type in parameter declaration
```

The equivalent declaration would have this form:

```
template<int N> void h(N*); // error: invalid type
```

— *end example*]

17 A function template can be an abbreviated function template. The invented *template-parameter*s are appended to the *template-parameter-list* after the explicitly declared *template-parameter*s.

[*Example:*

```
template<typename T, int N> class Array { };

template<int N> void f(Array<auto, N>*);
template<int N, typename T> void f(Array<T, N>*); // OK: equivalent to f(Array<auto, N>*)
```

— *end example*]

18 The use of a *constrained-type-specifier* in a *parameter-declaration* associates a *constraint-expression* with the abbreviated function template. This *constraint-expression* is formed according to the rules in 14.10.6. Two *constrained-type-specifier*s are said to be *equivalent* if their associated *constraint-expression*s are equivalent according to the rules in 14.6.6.1.

19 All placeholders introduced by equivalent *constrained-type-specifier*s have the same invented template parameter. [*Example:*

```
namespace N {
  template<typename T> concept bool C = true;
}
template<typename T> concept bool C = true;
template<typename T, int> concept bool D = true;
template<typename, int = 0> concept bool E = true;

void f0(C a, C b);
```

The types of `a` and `b` are the same invented template type parameter.

```
void f1(C& a, C* b);
```

The type of `a` is a reference to an invented template type parameter (call it `T`), and the type of `b` is a pointer to `T`.

```
void f2(N::C a, C b);
void f3(D<0> a, D<1> b);
```

In both functions, the parameters `a` and `b` have different invented template type parameters.

```
void f4(E a, E<> b, E<0> c);
```

The types of `a`, `b`, and `c` are the same because the *constrained-type-specifier*s `E`, `E<>`, and `E<0>` all associate the *constraint-expression* `E<T, 0>`, where `T` is an invented template type parameter.

10 Derived classes [class.derived]

10.3 Virtual functions [class.virtual]

Insert the following paragraph after paragraph 5 in order to prohibit the declaration of constrained virtual functions and the overriding of a virtual function by a constrained member function.

6 If a virtual function has associated constraints (Clause 14), the program is ill-formed. [*Example:*

```
struct A {
  virtual void f() requires true; // error: constrained virtual function
};

struct B {
  virtual void f();
};

struct D : B {
  void f() requires true; // error: constrained override
};
```

— end example]

13 Overloading [over]

Modify paragraph 1 to allow overloading based on constraints.

1 When two or more different declarations are specified for a single name in the same scope, that name is said to be overloaded. By extension, two declarations in the same scope that declare the same name but with different types or different associated constraints (Clause 14) are called *overloaded declarations*. Only function and function template declarations can be overloaded; variable and type declarations cannot be overloaded.

13.1 Overloadable declarations [over.load]

Update paragraph 3 to mention a function's overloaded constraints. Note that the itemized list in the original text is omitted in this document.

3 [*Note:* As specified in 8.4.1, function declarations that have equivalent parameter declarations and associated constraints, if any (Clause 14), declare the same function and therefore cannot be overloaded: ... — *end note*]

13.1.1 Declaration matching [over.dcl]

Modify paragraph 1 to extend the notion of declaration matching to also include a function's associated constrains. Note that the example in the original text is omitted in this document.

Two function declarations of the same name refer to the same function if they are in the same scope and have equivalent parameter declarations (13.1) and equivalent associated constraints, if any (Clause 14).

13.3 Overload resolution [over.match]

13.3.2 Viable functions [over.match.viable]

Update paragraph 1 to require the checking of a candidate's associated constraints when determining if that candidate is viable.

1 From the set of candidate functions constructed for a given context (13.3.1), a set of viable functions is chosen, from which the best function will be selected by comparing argument conversion sequences and associated constraints for the best fit (13.3.3). The selection of viable functions considers associated constraints, if any (Clause 14), and relationships between arguments and function parameters other than the ranking of conversion sequences.

Insert a new paragraph after paragraph 2; this introduces new a criterion for determining if a candidate is viable. Also, update the beginning of the subsequent paragraph to account for the insertion.

3 Second, for a function to be viable, if it has associated constraints (Clause 14), those constraints shall be satisfied (14.10).

4 ~~Second~~Third, for `F` to be a viable function...

13.3.3 Best viable function [over.match.best]

Modify the last item in the list in paragraph 1 and extend it with a final comparison based on the associated constraints of those functions. Note that the preceding (unmodified) items in the original document are elided in this document.

— ...

— `F1` and `F2` are function template specializations, and the function template for `F1` is more specialized than the template for `F2` according to the partial ordering rules described in 14.6.6.2~~.~~, or, if not that,

— `F1` and `F2` are non-template functions with the same parameter-type-lists, and `F1` is more constrained than `F2` according to the partial ordering of constraints described in 14.10.3.

13.4 Address of overloaded function [over.over]

Modify paragraph 4 (paragraph 5 in this document) to incorporate constraints in the selection of an overloaded function when its address is taken.

4 Eliminate from the set of selected functions all those whose constraints are not satisfied (14.10). ~~If more than one function is selected~~ If more than one function in the set remains, any function template specializations in the set are eliminated if the set also contains a function that is not a function template specialization~~, and~~. Any given non-template function `F0` is eliminated if the set contains a second non-template function that is more constrained than `F0` according to the partial ordering rules of 14.10.3. Additionally, any given function template specialization `F1` is eliminated if the set contains a second function template specialization whose function template is more specialized than the function template of `F1` according to the partial ordering rules of 14.6.6.2. After such eliminations, if any, there shall remain exactly one selected function.

Add the following example at the end of paragraph 5.

[*Example:*

```
void f();                  // #1
void f() requires true ; // #2
void g() requires false;
void g() requires false and true;

void (*pf)() = &amp;f;         // selects #2
void (*pg)() = &amp;g;         // error: no matching function
```

— end example]

14 Templates [temp]

Modify the *template-declaration* grammar in paragraph 1 to allow a template declaration introduced by a concept.

1

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

requires-clause:
  `requires` *constraint-expression*

Add the following paragraphs after paragraph 6.

7 A *template-declaration* is written in terms of its template parameters. These parameters are declared explicitly in a *template-parameter-list* (14.1), or they are introduced by a *template-introduction* (14.2).

8 The *associated constraints* of a *template-declaration* are defined to be a *constraint-expression* formed from the conjunction of *constraint-expressions* associated by:

(8.1) — a constraint associated by a template introduction (14.2), and

(8.2) — any constrained template parameters (14.1) in the declaration's *template-parameter-list*, and

(8.3) — a *requires-clause* following a *template-parameter-list*, and

(8.4) — any *constrained-type-specifiers* in the type of a *parameter-declaration* in a function declaration (7.1.6.5), and

(8.5) — a *requires-clause* appearing in the *declarator* of a function declaration (8.4.1).

The formation of the associated constraints for a template declaration gives a definite ordering on subexpressions for the purpose of determining when one template redeclares another.

[*Example:*

```
template<typename T> concept bool C = true;

// all of the following declare the same function:
void g(C);
template<C T> void g(T);
C{T} void g(T);
template<typename T> requires C<T> void g(T);

// these also declare the same function:
template<C T> void f();
template<typename T> void f() requires C<T>;
```

— *end example*]

9 The order in which the subexpressions of the associated constraints are composed is the left-to-right order in which *template-introductions*, *constrained-type-specifiers*, and *requires-clauses* occur in the declaration. [*Note:* A program containing two declarations whose associated constraints are functionally equivalent but not equivalent (14.6.6.1) is ill-formed, no diagnostic required. — *end note*] [*Example:*

```
template<typename T> concept bool C1 = true;
template<typename T> concept bool C2 = true;

C1{T} void f1(C2);                                                       // #1
template<typename T, typename U> requires C1<T> && C2<U> void f1(T, U); // #2
```

In the associated constraints of #1, the constraint associated by the *template-introduction* `C1T` occurs before the constraint associated by the *constrained-type-specifier* in the *parameter-declaration* `C2`. The resulting *constraint-expression* is equivalent to the *requires-clause* in #2.

```
C1{T} void f2(T) requires C2<T>;                        // #1
template<typename T> requires C1<T> && C2<T> void f2(); // #2
```

The associated constraints of #1 and #2 are equivalent.

```
template<C1 T> requires C2<T> void f3(T);                // #1
template<C1 T> void f3(T) requires C2<T>;                // #2
template<typename T> requires C1<T> && C2<T> void f3(T); // #3
template<typename T> void f3(T) requires C1<T> && C2<T>; // #4
```

The associated constraints of #1, #2, #3, and #4 are equivalent. The *constraint-expression* associated by `C1` occurs before the constraint associated by `C2` in each declaration.

```
template<C1 T> requires C2<T> void f5();
template<C2 T> requires C1<T> void f5(); // error: constraints are functionally equivalent but
not equivalent
```

The associated constraints of the first declaration are `C1<T> && C2<T>`, and those of the second are `C2<T> && C1<T>`. — *end example*]

14.1 Template parameters [temp.param]

In paragraph 1, extend the grammar for template parameters to constrained template parameters.

1 *template-parameter:* *constrained-parameter*

constrained-parameter:
constrained-type-specifier ...$_{opt}$ *identifier*$_{opt}$
constrained-type-specifier ...$_{opt}$ *identifier*$_{opt}$= *type-id*
constrained-type-specifier ...$_{opt}$ *identifier*$_{opt}$= *id-expression*
constrained-type-specifier ...$_{opt}$ *identifier*$_{opt}$= *initializer-clause*

Insert a new paragraph after paragraph 1.

2 There is an ambiguity in the syntax of a template parameter between the declaration of a *constrained-parameter* and *parameter-declaration*. If the *type-specifier-seq* of a *parameter-declaration* is a *constrained-type-specifier*, then the *template-parameter* is a *constrained-parameter*.

Insert the following paragraphs after paragraph 8. These paragraphs define the meaning of a constrained template parameter.

9 A *constrained-parameter* declares a template parameter whose kind (type, non-type, template) and type match that of the prototype parameter of the concept designated by its *constrained-type-specifier*. The designated concept is selected by concept resolution (14.10.5). Let `P` be the prototype parameter of the designated concept. The declared template parameter is determined by the type and form of `P` and the optional ellipsis in the *template-parameter*.

(9.1) — If `P` is a type *template-parameter* the declared parameter is a type *template-parameter*.

(9.2) — If P is a non-type *template-parameter*, the declared parameter is a non-type *template-parameter* having the same type as P.

(9.3) — If P is a template *template-parameter*, the declared parameter is a template *template-parameter* having the same *template-parameter-list* as P, excluding default template arguments.

(9.4) — If P declares a template parameter pack, the *constrained-type-specifier* shall be followed by an ellipsis, and the declared parameter is a template parameter pack.

[*Example:*

```
template<typename T> concept bool C1 = true;
template<template<typename> class X> concept bool C2 = true;
template<int N> concept bool C3 = true;
template<typename... Ts> concept bool C4 = true;
template<char... Cs> concept bool C5 = true;

template<C1 T> void f1();      // OK: T is a type template-parameter
template<C2 X> void f2();      // OK: X is a template with one type-parameter
template<C3 N> void f3();      // OK: N has type int
template<C4... Ts> void f4(); // OK: Ts is a template parameter pack of types
template<C4 Ts> void f5();     // error: Ts must be preceded by an ellipsis
template<C5... Cs> f6();       // OK: Cs is a template parameter pack of chars
```

— *end example*]

10 A *constrained-parameter* associates a *constraint-expression* with its *template-declaration*. This *constraint-expression* is formed according to the rules in 14.10.6.

Insert the following paragraph after paragraph 9 to restrict the forms of default template argument for *constrained-parameter*.

12 The default *template-argument* of a *constrained-parameter* shall match the kind (type, non-type, template) of the declared parameter.

[*Example:*

```
template<typename T> concept bool C1 = true;
template<int N> concept bool C2 = true;
template<template<typename> class X> concept bool C3 = true;

template<typename T> struct S0;

template<C1 T = int> struct S1; // OK
template<C2 N = 0> struct S2;   // OK
template<C3 X = S0> struct S3;  // OK
template<C1 T = 0> struct S4;   // error: default argument is not a type
```

— *end example*]

14.2 Introduction of template parameters [temp.intro]

Add this section after 14.1.

1 A *template introduction* provides a more concise way of declaring different templates that have the same template parameters and constraints.

template-introduction:
  nested-name-specifier$_{opt}$ *concept-name* `{` *introduction-list* `}`

introduction-list:
  introduced-parameter introduction-list `,` *introduced-parameter*

introduced-parameter:
  `...`$_{opt}$ *identifier*

A template introduction declares a sequence of *template-parameter*s, which are derived from a *concept-name* and the sequence of *identifier*s in its *introduction-list*.

2 The concept designated by the *concept-name* is selected by concept resolution (14.10.5).

3 For each *introduced-parameter* `I` in an *introduction-list*, and for its corresponding selected template parameter `P` from the designated concept, declare a new template parameter using the rules for declaring a constrained parameter in 14.1 by using `I` as a *declarator-id* and `P` as the prototype parameter. If `I` contains an ellipsis, `P` declares a template parameter pack. [*Example:*

```
template<typename T, int N, typename... Xs> concept bool C1 = true;
template<template<typename> class X> concept bool C2 = true;

C1{A, B, ...C} // OK: A is declared as typename A,
struct s;      // B is declared as int B, and
               // C is declared as typename... C

C1{X, Y, Z} // error: Z must be preceded by an ellipsis
struct t;

C2{T} // OK: T is declared as template<typename> class T
void foo();

C2{...X}    // error: the corresponding parameter is not a
void bar(); // template parameter pack
```

— *end example*]

4 [*Note:* A concept referred to by a *concept-name* may have template parameters with default template arguments. An *introduction-list* may omit *identifier*s for a corresponding template parameter if it has a default argument. However, only the *introduced-parameter*s are declared as template parameters. [*Example:*

```
template<typename A, typename B = bool> concept bool C() { return true; }

C{T} void f(T); // OK: f(T) is a function template with
                // a single template type parameter T
```

There is no *introduced-parameter* that corresponds to the template parameter `B` in the `C` concept, so `f(T)` is declared with only one template parameter. — *end example*] — *end note*]

5 An introduced template parameter does not have a default template argument even if its corresponding template parameter does. [*Example:*

```
template<typename T, int N = -1> concept bool P() { return true; }

P{T, N} struct Array { };

Array<double, 0> s1; // OK
Array<double> s2;    // error: Array takes two template arguments
```

— *end example*]

6 A *template-introduction* associates a *constraint-expression* with its *template-declaration*.

The constraint is formed from the concept `C`, designated by the *constrained-type-specifier* (including its *nested-name-specifier*), and the sequence of introduced parameters.

Form a sequence of *template-argument*s `Args` from the template parameters declared by the *introduced-parameter*s as follows. If an introduced template parameter `T` is declared as a template parameter pack, its corresponding *template-argument* is a pack expansion of `T`. Otherwise, its corresponding *template-argument* matches the type and form of `T`. Let `TT` be a *template-id* formed as `C<Args>`. The associated *constraint-expression* is the *id-expression* `TT` if `C` refers to a variable concept. The associated *constraint-expression* is the function call `TT()` if `C` refers to a function concept. [*Example:*

```
template<typename T, typename U> concept bool C1 = true;
template<typename T, typename U> concept bool C2() { return true; }
template<typename T, typename U = char> concept bool C3 = true;
template<typename... Ts> concept bool C4 = true;

C1{A, B} struct X;
C2{A, B} struct Y;
C3{P} void f(P);
C4{...Qs} void g(Qs&&...);

template<typename A, typename B>
  requires C1<A, B> // constraint associated by C1A, B
    struct X;       // OK: redeclaration of X

template<typename A, typename B>
  requires C2<A, B>()  // constraint associated by C2A, B
    struct Y;          // OK: redeclaration of Y

template<class P>
  requires C3<P> // constraint associated by C3P
    void f(P);   // OK: redeclaration of f(P)

template<typename... Qs>
  requires C4<Qs...> // constraint associated by C4...Qs
    void g(Qs&&...); // OK: redeclaration of g(Qs&&...)
```

— *end example*]

7 A template declared by a *template-introduction* can also be an abbreviated function template (8.4.1). The invented template parameters introduced by the placeholders in the abbreviated function template are appended to the list of template parameters declared by the *template-introduction*. [*Example:*

```
template<typename T> concept bool C1 = true;
template<typename T> concept bool C2 = true;

C1{T} void f(T, auto);
template<C1 T, typename U> void f(T, U); // OK: redeclaration of f(T, auto)
```

[*Note:* The second declaration of `f` is a redeclaration of the first because the associated constraints of each are equivalent (14.6.6.1). — *end note*] — *end example*]

14.3 Names of template specializations [temp.names]

Add this paragraph at the end of the section to require the satisfaction of associated constraints on the formation of the *simple-template-id*.

8 When a *simple-template-id* names a constrained non-function template or a template template parameter, but not a member template that is a member of an unknown specialization 14.7, and all *template-arguments* in the *template-id* are non-dependent 14.6.2.4, the template arguments are substituted into the associated constraints (Clause 14). If, as a result of substitution, the associated constraints are not satisfied (14.10), the *template-id* is ill-formed. [*Example:*

```
template<typename T> concept bool C = false;

template<C T> struct S1 { };
template<C T> using Ptr = T*;

S1<int>* p; // error: constraints not satisfied
Ptr<int> p; // error: constraints not satisfied

template<typename T>
  struct S2 { Ptr<int> x; }; // error: constraints not satisfied

template<typename T>
  struct S3 { Ptr<T> x; };    // OK: satisfaction is not required

S3<int> x;                    // error: constraints not satisfied

template<template<C T> class X>
  struct S4 {
    X<int> x; // error: constraints not satisfied

    using Type = T::template MT<char>;
  };
```

The error in the instantiation of `S4<int>` is caused by the substitution into the type of the member `x`.

Because there is no declaration for the template named by `T::template MT<char>`, it cannot have associated constraints. — *end example*]

14.4 Template arguments [temp.arg]

14.4.1 Template template arguments [temp.arg.template]

Modify paragraph 3 to include rules for matching constrained template *template parameters*. Note that the examples following this paragraph in the original document are omitted.

3 A *template-argument* matches a template *template-parameter* (call it `P`) when each of the template parameters in the *template-parameter-list* of the *template-argument*'s corresponding class template or alias template (call it `A`) matches the corresponding template parameter in the *template-parameter-list* of `P`, and `P` is at least as constrained as `A` according to the rules in 14.10.3. Two template parameters match if they are of the same kind (type, non-type, template), for non-type *template-parameters*, their types are equivalent (14.6.6.1), and for template *template-parameters*, each of their corresponding *template-parameters* matches, recursively. When `P`âĂŹs *template-parameter-list* contains a template parameter pack (14.5.3), the template parameter pack will

match zero or more template parameters or template parameter packs in the *template-parameter-list* of `A` with the same kind (type, non-type, template) and type as the template parameter pack in `P` (ignoring whether those template parameters are template parameter packs).

Add the following example to the end of paragraph 3, after the examples given in the original document.

[*Example:*

```
template<typename T> concept bool C = requires (T t) { t.f(); };
template<typename T> concept bool D = C<T> && requires (T t) { t.g(); };

template<template<C> class P>
  struct S { };

template<C> struct X { };
template<D> struct Y { };
template<typename T> struct Z { };

S<X> s1; // OK: X has the same constraints as P
S<Y> s2; // error: the constraints of P do not subsume those of Y
S<Z> s3; // OK: the constraints of P subsume those of Z
```

— end example]

14.6 Template declarations [temp.decls]

Modify paragraph 2 to indicate that associated constraints are instantiated separately from the template they are associated with.

For purposes of name lookup and instantiation, default arguments, associated constraints (Clause 14), and *exception-specifications* of function templates and default arguments, associated constraints, and *exception-specifications* of member functions of class templates are considered definitions; each default argument, associated constraint, or *exception-specification* is a separate definition which is unrelated to the function template definition or to any other default arguments, associated constraints, or *exception-specifications*.

14.6.1 Class templates [temp.class]

Modify paragraph 3 to require template constraints for out-of-class definitions of members of constrained templates.

3 When a member function, a member class, a member enumeration, a static data member or a member template of a class template is defined outside of the class template definition, the member definition is defined as a template definition in which the *template-parameters* and associated constraints are those of the class template. The names of the template parameters used in the definition of the member may be different from the template parameter names used in the class template definition. The template argument list following the class template name in the member definition shall name the parameters in the same order as the one used in the template parameter list of the member. Each template parameter pack shall be expanded with an ellipsis in the template argument list.

Add the following example at the end of paragraph 3.

[*Example:*

```
template<typename T> concept bool C = true;
template<typename T> concept bool D = true;

template<C T> struct S {
    void f();
    void g();
    template<D U> struct Inner;
  }

template<typename T> requires C<T> void S<T>::f() { } // OK: parameters and constraints match
template<typename T> void S<T>::g() { } // error: no matching declaration for S<T>

template<C T> D{U} struct S<T>::Inner { }; // OK
```

— end example]

14.6.1.1 Member functions of class templates [temp.mem.func]

Add the following example to the end of paragraph 1.

[*Example:*

```
template<typename T> struct S {
  void f() requires true;
  void g() requires true;
};

template<typename T>
  void S<T>::f() requires true { } // OK
template<typename T>
  void S<T>::g() { }               // error: no matching function in S<T>
```

— end example]

14.6.2 Member templates [temp.mem]

Modify paragraph 1 in order to account for constrained member templates of (possibly) constrained class templates.

> A template can be declared within a class or class template; such a template is called a member template.
>
> A member template can be defined within or outside its class definition or class template definition.
>
> A member template of a class template that is defined outside of its class template definition shall be specified with the *template-parameters* and associated constraints of the class template followed by the *template-parameters* and associated constraints (Clause 14) of the member template.

Add the following example at the end of paragraph 1.

[*Example:*

```
template<typename T> concept bool C1 = true;
template<typename T> concept bool C2 = sizeof(T) <= 4;

template<C1 T>
```

```
  struct S {
    template<C2 U> void f(U);
    template<C2 U> void g(U);
  };

template<C1 T> template<typename U>
  void S<T>::f(U) requires C2<U> { } // OK
template<C1 T> template<typename U>
  void S<T>::g(U) { }               // error: no matching function in S<T>
```

— *end example*]

14.6.4 Friends [temp.friend]

Modify paragraph 9 to restrict constrained friend declarations.

9 When a friend declaration refers to a specialization of a function template, the function parameter declarations shall not include default arguments, the declaration shall not be constrained, nor shall the inline specifier be used in such a declaration.

Add examples following that paragraph.

10 [*Note:* Other friend declarations can be constrained. In a constrained friend declaration that is not a definition, the constraints are used for declaration — *end note*] [*Example:*

```
template<typename T> concept bool C1 = true;
template<typename T> concept bool C2 = false;

template<C1 T> g0(T);
template<C1 T> g1(T);
template<C2 T> g2(T);

template<typename T>
  struct S {
    friend void f1() requires true;                    // OK
    friend void f2() requires C1<T>;             // OK
    friend void g0<T>(T) requires C1<T<; // error: constrained friend specialization
    friend void g1<T>(T);                        // OK
    friend void g2<T>(T);                        // error: constraint can never be satisfied, no
diagnostic required
  };

void f1() requires true;     // friend of all S<T>
void f2() requires C1<int>; // friend of S<int>
```

The friend declaration of `g2` is ill-formed, no diagnostic required, because no valid specialization of `S` can be generated: the constraint on `g2` can never be satisfied, so template argument deduction (14.9.5.1) will always fail. — *end example*]

11 [*Note:* Within a class template, a friend may define a non-template function whose constraints specify requirements on template arguments. [*Example:*

```
template<typename T> concept bool Eq = requires (T t) { t == t; };

template<typename T>
  struct S {
    friend bool operator==(S a, S b) requires Eq<T> { return a == b; } // OK
  };
```

— *end example*] In the instantiation of such a class template, the template arguments are substituted into the constraints but not evaluated. Constraints are checked (14.10) only when that function is considered as a viable candidate for overload resolution (13.3.2). — *end note*]

14.6.5 Class template partial specialization [temp.class.spec]

After paragraph 3, insert the following, which explains constrained partial specializations.

4 A class template partial specialization may be constrained (Clause 14). [*Example:*

```
template<typename T> concept bool C = requires (T t) { t.f(); };
template<int I> concept bool N = I > 0;

template<C T1, C T2, N I> class A<T1, T2, I>;  // #6
template<C T, N I>        class A<int, T*, I>; // #7
```

— *end example*]

Remove the 3rd item in the list of paragraph 8 to allow constrained class template partial specializations like #6, and because it is redundant with the 4th item. Note that all other items in that list are elided.

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

(7.1) — ...

(7.2) — ~~The argument list of the specialization shall not be identical to the implicit argument list of the primary template.~~

(7.3) — The specialization shall be more specialized than the primary template (14.6.5.2).

(7.4) — ...

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

Modify paragraph 2; constraints must be satisfied in order to match a partial specialization.

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 (14.9.1) , and the deduced template arguments satisfy the constraints of the partial specialization, if any (14.10).

Add the following example to the end of paragraph 2.

[*Example:*

```
struct S { void f(); };

A<S, S, 1>    a6; // uses #6
A<S, int, 2>  a7; // error: constraints not satisfied
A<int, S*, 3> a8; // uses #7
```

— *end example*]

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

Modify paragraph 1 so that constraints are considered in the partial ordering of class template specializations.

1 For two class template partial specializations, the first is at least as specialized as the second if, given the following rewrite to two function templates, the first function template is at least as specialized as the second according to the ordering rules for function templates (14.6.6.2):

(1.1) — the first function template has the same template parameters and associated constraints as the first partial specialization, and has a single function parameter whose type is a class template specialization with the template arguments of the first partial specialization, and

(1.2) — the second function template has the same template parameters and associated constraints as the second partial specialization, and has a single function parameter whose type is a class template specialization with the template arguments of the second partial specialization.

Add the following example to the end of paragraph 1.

[*Example:*

```
template<typename T> concept bool C = requires (T t) { t.f(); };
template<typename T> concept bool D = C<T> && requires (T t) { t.f(); };


template<typename T> class S { };
template<C T> class S<T> { }; // #1
template<D T> class S<T> { }; // #2

template<C T> void f(S<T>); // A
template<D T> void f(S<T>); // B
```

The partial specialization #2 is more specialized than #1 for template arguments that satisfy both constraints because `B` is more specialized than `A`. — *end example*]

14.6.6 Function templates [temp.fct]

14.6.6.1 Function template overloading [temp.over.link]

Modify paragraph 6 to account for constraints on function templates.

6

~~Two function templates are *equivalent* if they are declared in the same scope, have the same name, have identical template parameter lists, 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 *equivalent* if they:

(6.1) — are declared in the same scope,

(6.2) — have the same name,

(6.3) — have identical template parameter lists,

(6.4) — have return types and parameter lists that are equivalent using the rules described above to compare expressions involving template parameters, and

(6.5) — have associated constraints (Clause 14) that are equivalent using the rules 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, or the associated constraints (Clause 14) 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.

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

Modify paragraph 2 to include constraints in the partial ordering of function templates.

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 type. 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. If both deductions succeed, the partial ordering selects the more constrained template as described by the rules in 14.10.3.

14.7 Name resolution [temp.res]

Modify paragraph 8.

8 Knowing which names are type names allows the syntax of every template to be checked. No diagnostic shall be issued for a template for which a valid specialization can be generated. If no valid specialization can be generated for a template, and that template is not instantiated, the template is ill-formed, no diagnostic required. If every valid specialization of a variadic template requires an empty template parameter pack, the template is ill-formed, no diagnostic required. If no instantiation of the associated constraints of a template would result in a valid expression, the template is ill-formed, no diagnostic required. If a hypothetical instantiation of a template immediately following its definition would be ill-formed due to a construct that does not depend on a template parameter, the program is ill-formed; no diagnostic is required. If the interpretation of such a construct in the hypothetical instantiation is different from the interpretation of the corresponding construct in any actual instantiation of the template, the program is ill-formed; no diagnostic is required.

14.7.4 Dependent name resolution [temp.dep.res]

14.7.4.1 Point of instantiation [temp.point]

Add a new paragraph after paragraph 4.

5 The point of instantiation of a *constraint-expression* of a specialization immediately precedes the point of instantiation of the specialization.

14.8 Template instantiation and specialization [temp.spec]

14.8.1 Implicit Instantiation [temp.inst]

Change paragraph 1 to include associated constraints.

Unless a class template specialization has been explicitly instantiated 14.8.2 or explicitly specialized 14.8.3, the class template specialization is implicitly instantiated when the specialization is referenced in a context that requires a completely-defined object type or when the completeness of the class type affects the semantics of the program. [*Note:* Within a template declaration, a local class or enumeration and the members of a local class are never considered to be entities that can be separately instantiated (this includes their default arguments, exception-specifications, and non-static data member initializers, if any). As a result, the dependent names are looked up, the semantic constraints are checked, and any templates used are instantiated as part of the instantiation of the entity within which the local class or enumeration is declared. — *end note*] The implicit instantiation of a class template specialization causes the implicit instantiation of the declarations, but not of the definitions, default arguments, associated constraints (Clause 14), or *exception-specification*s of the class member functions, member classes, scoped member enumerations, static data members and member templates; and it causes the implicit instantiation of the definitions of unscoped member enumerations and member anonymous unions.

Add a new paragraph after paragraph 15 to describe how associated constraints are instantiated.

16 The associated constraints of a template specialization are not instantiated along with the specialization itself; they are instantiated only to determine if they are satisfied (14.10). [*Note:* The satisfaction of constraints is determined during lookup or overload resolution (13.3). Preserving the spelling of the substituted constraint also allows constrained member function to be partially ordered by those constraints according to the rules in 14.10.3. — *end note*] [*Example:*

```
template<typename T> concept bool C = sizeof(T) > 2;
template<typename T> concept bool D = C<T> && sizeof(T) > 4;

template<typename T> struct S {
  S() requires C<T> { } // #1
  S() requires D<T> { } // #2
};

S<char> s1;    // error: no matching constructor
S<char[8]> s2; // OK: calls #2
```

Even though neither constructor for `S<char>` will be selected by overload resolution, they remain a part of the class template specialization. This also has the effect of suppressing the implicit generation of a default constructor (12.1). — *end example*]

[*Example:*

```
template<typename T> struct S1 {
  template<typename U> requires false struct Inner1; // OK
};

template<typename T> struct S2 {
  template<typename U>
    requires sizeof(T[(int)-sizeof(T)]) > 1 // error: ill-formed, no diagnostic required
      struct Inner2;
};
```

— *end example*] Every instantiation of `S1` results in a valid type, although any use of its nested `Inner1` template is invalid. `S2` is ill-formed, no diagnostic required, since no substitution into the constraints of its `Inner2` template would result in a valid expression.

14.8.2 Explicit instantiation [temp.explicit]

Modify paragraph 8 to ensure that only members whose constraints are satisfied are explicitly instantiated during class template specialization. The note in the original document is omitted.

8 An explicit instantiation that names a class template specialization is also an explicit instantiation of the same kind (declaration or definition) of each of its members (not including members inherited from base classes and members that are templates) that has not been previously explicitly specialized in the translation unit containing the explicit instantiation, and provided that the associated constraints (14), if any, of that member are satisfied (14.10) by the template arguments of the explicit instantiation, except as described below.

Add the following paragraphs to this section. These require an explicit instantiation of a constrained template to satisfy the template's associated constraints.

14 If the explicit instantiation names a class template specialization or variable template specialization of a constrained template, then the *template-arguments* in the *template-id* of the explicit instantiation shall satisfy the template's associated constraints (14.10). [*Example:*

```
template<typename T> concept bool C = sizeof(T) == 1;

template<C T> struct S { };

template struct S<char>;    // OK
template struct S<char[2]>; // error: constraints not satisfied
```

— end example]

15 When an explicit instantiation refers to a specialization of a function template (14.9.5.1), that template's associated constraints shall be satisfied by the template arguments of the explicit instantiation.

[*Example:*

```
template<typename T> concept bool C = requires (T t) { -t; };

template<C T>        void f(T) { } // #1
template<typename T> void g(T) { } // #2
template<C T>        void g(T) { } // #3

template void f(int);   // OK: refers to #1
template void f(void*); // error: no matching template
template void g(int);   // OK: refers to #3
template void g(void*); // OK: refers to #2
```

— end example]

14.8.3 Explicit specialization [temp.expl.spec]

Insert the following paragraphs after paragraph 12. These require an explicit specialization to satisfy the constraints of the primary template.

13 The *template-arguments* in the *template-id* of an explicit specialization of a constrained non-function template shall satisfy the associated constraints of that template, if any (14.10). [*Example:*

```
template<typename T> concept bool C = sizeof(T) == 1;

template<C T> struct S { };

template<> struct S<char> { };    // OK
template<> struct S<char[2]> { }; // error: constraints not satisfied
```

— *end example*]

14 When determining the function template referred to by an explicit specialization of a function template (14.9.5.1), the associated constraints of that template (if any) shall be satisfied (14.10) by the template arguments of the explicit specialization.

[*Example:*

```
template<typename T> concept bool C = requires (T t) { -t; };

template<C T>        void f(T); // #1
template<typename T> void g(T); // #2
template<C T>        void g(T); // #3

template<> void f(int);   // OK: refers to #1
template<> void f(void*); // error: no matching template
template<> void g(int);   // OK: refers to #3
template<> void g(void*); // OK: refers to #2
```

— *end example*]

14.9 Function template specializations [temp.fct.spec]

14.9.1 Template argument deduction [temp.deduct]

Add the following sentences to the end of paragraph 5. This defines the substitution of template arguments into a function template's associated constraints. Note that the last part of paragraph 5 has been duplicated in order to provide context for the addition.

5 When all template arguments have been deduced or obtained from default template arguments, all uses of template parameters in the template parameter list of the template and the function type are replaced with the corresponding deduced or default argument values.

If the substitution results in an invalid type, as described above, type deduction fails.

If the function template has associated constraints (Clause 14), the template arguments are substituted into the associated constraints without evaluating the resulting expression. If this substitution results in an invalid type or expression, type deduction fails. [*Note:* The satisfaction of constraints (14.10) associated with the function template specialization is determined during overload resolution (13.3), and not at the point of substitution. — *end note*]

14.9.5.1 Deducing template arguments from a function declaration [temp.deduct.decl]

Add the following after paragraph 1 in order to require the satisfaction of constraints when matching a specialization to a template.

3 Remove from the set of function templates considered all those whose associated constraints (if any) are not satisfied by the deduced template arguments (14.10).

Update paragraph 2 (now paragraph 3) to accommodate the new wording.

4 If, ~~for the set of function templates so considered~~ for the remaining function templates, there is either no match or more than one match after partial ordering has been considered (14.6.6.2), deduction fails and, in the declaration cases, the program is ill-formed.

14.10 Template constraints [temp.constr]

Add this section after 14.8.

1 [*Note:* This section defines the meaning of constraints on template arguments, including the translation of *constraint-expressions* into constraints (14.10.4), and also the abstract syntax, satisfaction, and subsumption of those constraints (14.10.1, 14.10.2). — *end note*]

2 A *constraint* is a sequence of logical operations and operands that specifies requirements on template arguments. [*Note:* The operands of a logical operation are constraints. — *end note*]

3 After substitution, a constraint is *satisfied* if and only if it and all of its operands are satisfied according to the evaluation rules described in 14.10.1 and 14.10.2. If the substitution of template arguments into a constraint fails, that constraint is not satisfied. [*Note:* Substitution into a constraint may yield a well-formed constraint that contains ill-formed expressions or types. This may happen, for example, in the substitution into expression constraints (14.10.2.2) and type constraints (14.10.2.3). — *end note*]

4 A constraint `P` is said to *subsume* another constraint `Q` if, informally, it can be determined that `P` implies `Q`, up to the equivalence of types and expressions in atomic constraints. 14.10.2. [*Note:* Subsumption does not determine, for example, if the predicate constraint (14.10.2.1) `N % 2 == 1` subsumes `N & 1` for some integral template argument, `N`. — *end note*] The rules determining when one constraint subsumes another is given in 14.10.3, and subsumption rules for each kind of atomic constraint are given in 14.10.2.

14.10.1 Logical operations [temp.constr.op]

1 There are two logical operations on constraints: conjunction and disjunction. [*Note:* These logical operations have no corresponding C++ syntax. For the purpose of exposition, conjunction is spelled using the symbol $\wedge$ and disjunction is spelled using the symbol $\vee$. Grouping of constraints is shown using parentheses. — *end note*]

2 A *conjunction* is a logical operation taking two operands. A conjunction of constraints is satisfied if and only if both operands are satisfied.

3 A *disjunction* is a logical operation taking two operands. A disjunction of constraints is satisfied if and only if either operand is satisfied or both operands are satisfied.

14.10.2 Atomic constraints [temp.constr.atom]

Any constraint that is not a conjunction or disjunction is an *atomic constraint*.

14.10.2.1 Predicate constraints [temp.constr.atom.pred]

1 A *predicate constraint* is an atomic constraint that evaluates a prvalue constant expression of type `bool` (5.19). The constraint is satisfied if and only if the expression evaluates to `true`. [*Note:* Predicate constraints allow the definition of template requirements in terms of constant expressions. This allows the specification constraints on non-type template arguments and template template arguments. — *end note*] [*Example:*

```
template<typename T> concept bool C = sizeof(T) == 4 && !true;
```

Here, `sizeof(T) == 4` and `!true` are predicate constraints required by the concept, `C`. — *end example*]

2 A predicate constraint `P` subsumes another predicate constraint `Q` if and only if `P` and `Q` are equivalent *constraint-expressions* (14.10.4). [*Example:* The predicate `M >= 0` does not subsume the predicate `M > 0` because they are not equivalent *constraint-expressions*. — *end example*]

14.10.2.2 Expression constraints [temp.constr.atom.expr]

1 An *expression constraint* is an atomic constraint that specifies a requirement on the formation of an *expression* `E` through substitution of template arguments. An expression constraint is satisfied if substitution yielding `E` did not fail. Within an expression constraint, `E` is an unevaluated operand (Clause 5). [*Note:* An expression constraint is introduced by the *expression* in either a *simple-requirement* (5.1.3.1) or *compound-requirement* (5.1.3.3) of a *requires-expression*. — *end note*] [*Example:*

```
template<typename T> concept bool C = requires (T t) { ++t; };
```

The concept `C` introduces an expression constraint for the expression `++t`. The type argument `int` satisfies this constraint because the the expression `++t` is valid after substituting `int` for `T`. — *end example*]

2 An expression constraint `P` subsumes another expression constraint `Q` if and only if the *expressions* of `P` and `Q` are equivalent (14.6.6.1).

14.10.2.3 Type constraints [temp.constr.atom.type]

1 A *type constraint* is an atomic constraint that specifies a requirement on the formation of a type `T` through the substitution of template arguments. A type constraint is satisfied if and only `T` is non-dependent, meaning that the substitution yielding `T` did not fail. [*Note:* A type constraint is introduced by the *typename-specifier* in a *type-requirement* of a *requires-expression* (5.1.3.2). — *end note*] [*Example:*

```
template<typename T> concept bool C = requires () { typename T::type; };
```

The concept `C` introduces a type constraint for the type name `T::type`. The type `int` does not satisfy this constraint because substitution of that type into the constraint results in a substitution failure; `typename int::type` is ill-formed. — *end example*]

2 A type constraint that names a class template specialization does not require that type to be complete (3.9).

3 A type constraint `P` subsumes another type constraint `Q` if and only if the types in `P` and `Q` are equivalent (14.4).

14.10.2.4 Implicit conversion constraints [temp.constr.atom.conv]

1 An *implicit conversion constraint* is an atomic constraint that specifies a requirement on the implicit conversion of an *expression* `E` to a type `T`. The constraint is satisfied if and only if `E` is implicitly convertible to `T` (Clause 4).

[*Note:* A conversion constraint is introduced by a *trailing-return-type* in a *compound-requirement* when the *trailing-return-type* contains no placeholders (5.1.3.3). — *end note*]

[*Example:*

```
template<typename T> concept bool C =
  requires (T a, T b) {
    { a == b } -> bool;
  };
```

The *compound-requirement* in the *requires-expression* of `C` introduces two atomic constraints: an expression constraint for `a == b`, and the implicit conversion constraint that the expression `a == b` is implicitly convertible to `bool`. — *end example*]

2 An implicit conversion constraint `P` subsumes another implicit conversion constraint `Q` if and only if the *expression*s of `P` and `Q` are equivalent (14.6.6.1) and the types of `P` and `Q` are equivalent (14.4).</p>

14.10.2.5 Argument deduction constraints [temp.constr.atom.deduct]

1 An *argument deduction constraint* is an atomic constraint that specifies a requirement on the usability of an *expression* `E` as an argument to an invented abbreviated function template `F` (8.4.1), where `F` has a single parameter formed from a type that includes placeholders (7.1.6.4, 7.1.6.5). [*Note:* An argument deduction constraint is introduced by a *trailing-return-type* in a *compound-requirement* when the *trailing-type-specifier-seq* contains at least one placeholder (5.1.3.3). — *end note*] [*Example:*

```
template<typename T>
concept bool C1() { return true; }

template<typename T>
concept bool C2() { return requires(T t) { {*t} -> const C1& x; }; }
```

The invented function template for the *compound-requirement* in `C2` is:

```
void F(const C1& x);
```

— *end example*] The constraint is satisfied if and only if `F` is selected by overload resolution for the call `F(E)` (13.3). [*Note:* Overload resolution selects `F` only when template argument deduction succeeds and `F`'s associated constraints are satisfied. — *end note*]

2 An argument deduction constraint `P` subsumes another argument deduction constraint `Q` if and only if the *expressions* of `P` and `Q` are equivalent (14.6.6.1), and the types of `P` and `Q` are equivalent (14.4).

14.10.2.6 Exception constraints [temp.constr.atom.noexcept]

1 An *exception constraint* is an atomic constraint for an expression `E` that is satisfied if and only if the expression `noexcept(E)` is `true` (5.3.7). [*Note:* Constant expression constraints are introduced by a *compound-requirement* that includes the `noexcept` specifier (5.1.3.3). — *end note*]

2 An exception constraint `P` subsumes another exception constraint `Q` if and only if the *expression*s of `P` and `Q` are equivalent (14.6.6.1).

14.10.3 Partial ordering by constraints [temp.constr.order]

1 In order to determine if a constraint `P` subsumes a constraint `Q`, transform `P` into disjunctive normal form, and transform `Q` into conjunctive normal form[1]. The disjunctive normal form of `P` subsumes the conjunctive normal form of `Q` if and only if every disjunctive clause in `P`i subsumes each conjunctive clause in `Q`j. A disjunctive clause `P`i subsumes a conjunctive clause `Q`j when each atomic constraint in `P`i subsumes any atomic constraint in `Q`j. The rules for determining whether one atomic constraint subsumes another are defined for each kind of atomic constraint (14.10.2). [*Example:* Let `A` and `B` be predicate constraints (14.10.2.1). The constraint `A` ∧ `B`

1) A constraint is in disjunctive normal form when it is a disjunction of clauses where each clause is a conjunction of atomic constraints. Similarly, a constraint is in conjunctive normal form when it is a conjunction of clauses where each each clause is disjunction of atomic constraints. [*Example:* Let `A`, `B`, and `C` be atomic constraints. The constraint `A` ∧ (`B` ∨ `C`) is in conjunctive normal form. Its conjunctive clauses are `A` and (`B` ∨ `C`). The disjunctive normal form of the constraint `A` ∧ (`B` ∨ `C`) is (`A` ∧ `B`) ∨ (`A` ∧ `C`). Its disjunctive clauses are (`A` ∧ `B`) and (`A` ∧ `C`). — *end example*]

subsumes `A`, but `A` does not subsume `A ∧ B`. The constraint `A` subsumes `A ∨ B`, but `A ∨ B` does not subsume code `A`. Also, any constraint `P` subsumes itself. — *end example*]

2 The subsumption relation defines a partial ordering on constraints. This partial ordering is used to determine

(2.1) — the best viable candidate of non-template functions (13.3.3),

(2.2) — the address of a non-template function (13.4),

(2.3) — the matching of template template arguments (14.4.1),

(2.4) — the partial ordering of class template specializations (14.6.5.2), and

(2.5) — the partial ordering of function templates (14.6.6.2).

3 When two declarations `D1` and `D2` are partially ordered by their normalized constraints, `D1` is *more constrained* than `D2` if

(3.1) — `D1` and `D2` are both constrained declarations and `D1`'s associated constraints subsume but are not subsumed by those of `D2`; or

(3.2) — `D1` is constrained and `D2` is unconstrained.

[*Example:*

```
template<typename T> concept bool C1 = requires(T t) { --t; };
template<typename T> concept bool C2 = C1<T> && requires(T t) { *t; };

template<C1 T> void f(T);       // #1
template<C2 T> void f(T);       // #2
template<typename T> void g(T); // #3
template<C1 T> void g(T);       // #4

f(0);        // selects #1
f((int*)0);  // selects #2
g(true);     // selects #3 because C1<bool> is not satisfied
g(0);        // selects #4
```

— *end example*]

4 A declaration `D1` is *at least as constrained* as another declaration `D2` when `D1` is more constrained than `D2`, and `D2` is not more constrained than `D1`.

14.10.4 Constraint expressions [temp.constr.expr]

1 Certain contexts require expressions that can be transformed into constraints through the process of *normalization*.

constraint-expression:
    logical-or-expression

2 A *constraint-expression* is *normalized* by forming a constraint as follows.

(2.1) — The normalized form of `(P)` is the normalized form of `P`.

(2.2) — The normalized form of `P || Q` is the disjunction (14.10.1) of the normalized form of `P` and the normalized form of `Q`.
If, after substitution, overload resolution (13.3) selects a user-declared `operator||`, the program is ill-formed.

(2.3) — The normalized form of `P && Q` is the conjunction (14.10.1) of the normalized form of `P` and the normalized form of `Q`.
If, after substitution, overload resolution (13.3) selects a user-declared `operator&&`, the program is ill-formed.

(2.4) — The normalized form of a function call of the form `C<A1, A2, ..., AN>()`, where `A1, A2, ..., AN` is a sequence of template arguments and `C` names a function concept (7.1.7), is the result of substituting the template arguments into the expression returned by `C`.

(2.5) — The normalized form of an *id-expression* of the form `C<A1, A2, ..., AN>` where `A1, A2, ..., AN` is a sequence of template arguments and `C` names a variable concept (7.1.7) is the result of substituting the template arguments into the initializer of `C`.

(2.6) — The normalized form of a *requires-expression* (5.1.3) is the conjunction of constraints (14.10.1) introduced by the body of that expression.

(2.7) — Otherwise, `E` is a predicate constraint (14.10.2.1). After substitution, `E` shall be a converted constant expression of type `bool`.

[*Note:* A *constraint-expression* defines a subset of constant expressions over which certain logical implications can be deduced during translation.

The prohibition against user-defined logical operators is intended to prevent the subversion of the logic used to partially order constraints (14.10.3). — *end note*]

[*Example:*

```
template<typename T> concept bool C1() { return sizeof(T) == 1; }
template<typename T> concept bool C2 = C1<T>() && 1 == 2;
template<typename T> concept bool C3 = requires () { typename T::type; };

// Expression // Constraints
C2<char>           sizeof(char) == 1 /* and */ 1 == 2
C3<int>            /* type constraint for int::type */
3 + 4              // error: not a constraint
(bool)(3 + 4)      (bool)(3 + 4)
```

In the normalized constraints, the expressions `sizeof(char) == 1`, `1 == 2`, and `(bool)(3 + 4)` are predicate constraints (14.10.2.1).

The concept `C3` is normalized to a single type constraint (14.10.2.3) for the (ill-formed) type `int::type`.

The expression `3 + 4` is not a *constraint-expression* because it does not satisfy the requirements for being normalized into a predicate constraint. — *end example*]

14.10.5 Resolution of *constrained-type-specifier*s [temp.constr.resolve]

1 Whenever an *identifier* is a *concept-name*, it is necessary to determine a single concept referred to by the use of that name. *Concept resolution* is the process of selecting a concept from a set of concept definitions referred to by a concept name.

Concept resolution is performed when a *constrained-type-specifier* appears in the declaration of an abbreviated function (8.4.1) or generic lambda (5.1.2), in the *trailing-return-type* of a *compound-requirement*, the *constrained-type-specifier* of a *constrained-parameter*, or in a *template-introduction*.

2 A concept is selected from a set of concepts based on a sequence of template argument patterns and a sequence of explicit template arguments. A *template argument pattern* is a kind of template argument that is used to match the type and form of a template parameter from a concept definition. A template argument pattern can be a pack expansion. A concept is selected from the set by matching the template argument patterns and explicit template arguments against the template parameters of that concept.

3 When selecting a concept for a *constrained-type-specifier*, there is a single template argument pattern. If the *constrained-type-specifier* appears in the declaration of a *constrained-parameter*

(14.1) and is followed by an ellipsis, the template argument pattern is a pack expansion. If the *constrained-type-specifier* is a *partial-concept-id* the explicit *template-arguments* are those in the *partial-concept-id*. When selecting a concept for a *concept-name* in a *template-introduction*, there is one template argument pattern for each *introduced-parameter*. If an *introduced-parameter* is preceded by an ellipsis, its corresponding template argument pattern is a pack expansion. [*Example:*

```
template<typename T> concept bool C1() { return true; }
template<typename T, typename U> concept bool C1() { return true; }
template<typename... T> concept bool C2() { return true; }
%
void f1(C1);
void f2(C1<int>);
```

In the resolution of `C1` required by the declaration of `f1`, there is a single template argument pattern and zero explicit template arguments. For `f2` there is a single template argument pattern and the single explicit template argument, `int`.

```
C1{T} void f3(T);
C1{T, U} void f4(T);
```

In the resolution required by the declaration of `f3`, there is a single template argument pattern; there are two in the resolution required by `f2`. There are zero explicit template arguments in the resolutions of `f1` and `f2`.

```
C2{...T} void f5();
```

There is a single template argument pattern in the resolution of `f2`, and it is a pack expansion. *— end example*]

4 For each concept `C` in the concept set, each template argument in the combined sequence of template argument patterns and explicit template arguments is matched against the corresponding template parameter in the *template-parameter-list* of `C` as follows. A template argument pattern that is not a parameter pack matches a non-pack template parameter of any type and form. A template argument pattern that is a parameter pack matches a template parameter pack whose pattern is any form. The remaining explicit arguments are matched against parameters as specified in 14.4. If any template argument patterns or explicit template arguments do not match the corresponding parameter, `C` is removed from the set. If a single concept remains, it is the one selected by concept resolution. Otherwise, the program is ill-formed. </p> [*Example:*

```
template<typename T> concept bool C() { return true; }             // #1
template<typename T, typename U> concept bool C() { return true; } // #2
template<typename T> concept bool P() { return true; }
template<int T> concept bool P() { return true; }
template<<typename... Ts> concept bool Q = true;

void f1(const C*);   // OK: C selects #1
void f2(C<char>);    // OK: C<char> selects #2
void f3(C<3>);       // error: no matching concept for C<3> (mismatched template arguments)
void g1(P);          // error: resolution of P is ambiguous (P refers to two concepts)
Q{...Ts} void q1(); // OK: selects Q
Q{T} void q2();      // error: no matching concept (mismatched template arguments)
```

— end example]

5 For the selected template, the set of template parameters corresponding to the matched template argument patterns are called the *selected template parameters*. In a *template-introduction* (14.2), these are used to derive the declarations of introduced parameters.

14.10.6 Constraint formation from *constrained-type-specifier*s [temp.constr.form]

1 When a parameter of an abbreviated function template is declared with a *constrained-type-specifier*, or in the declaration of a *constrained-parameter*, the *constrained-type-specifier* associates a *constraint-expression* with the respective function or template declaration. The formation of that *constraint-expression* is derived from the *constrained-type-specifier*, the designated concept selected by concept resolution (14.10.5), and an invented template parameter or a declared template parameter, called the *target template parameter*. When the *constrained-type-specifier* appears in the declaration of a function parameter, the target template parameter is the one invented for the abbreviated function template (8.4.1).

When the *constrained-type-specifier* appears in the declaration of a *constrained-parameter*, the target template parameter is the declared template parameter.

2 Let `C` be the concept designated by the *constrained-type-specifier* (including its *nested-name-specifier*, if any), let `P` be the prototype parameter of the designated concept, and let `X` be the target template parameter. Form a new template argument `A` from `X` as as follows. If `X` declares a parameter pack, and `P` declares a parameter pack, `A` is a pack expansion of `X`. Otherwise `A` is a template argument referring to `X`. Form a *template-id* `TT` as follows. When the *constrained-type-specifier* is a *partial-concept-id*, `TT` is `C<A, Args>` where `Args` is the sequence of *template-arguments* in the *partial-concept-id*. Otherwise, `TT` is `C<A, Args>`. Form an expression `E` from `TT`. If `C` refers to a variable concept, `E` is the *id-expression* `TT`. If `C` refers to a function concept, `E` is the function call `TT()`.

3 The formed constraint is the fold expression `E1 && E2 && ... && E`N when the prototype parameter `P` declares a template parameter pack and the target template parameter `X` does not. Otherwise, the *constraint-expression* is `E`.

[*Example:*

```
template<typename T> concept bool C1 = true;
template<typename T, typename U> concept bool C2() { return true; }
template<typename... Ts> concept bool C3;

template<int> struct X { };

void f1(C1&);      // associates C1<T1> with f1
void f2(C2<int>); // associates C2<T2, int>() with f2
void f3(C1...);    // associates (C1<T1> && ...) with f3
void f4(C3...);    // associates C3<T1...> with f4
```

Here, `T1` and `T2` are invented type template parameters corresponding to the prototype parameter of their respective designated concepts.