ISO/IEC JTC1 SC22 WG21 N 4549
Date: 2015-07-27
ISO/IEC TS 19217
ISO/IEC JTC1 SC22
Secretariat: ANSI

Programming Languages — C++ Extensions for Concepts

Langages de programmation — Extensions C++ pour les concepts

Warning

This document is not an ISO International Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an International Standard.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Document type: Technical Specification
Document stage: (30) Committee
Document Language: E

Contents

List of Tables

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 N4191 defines "fold expressions", which are used to define constraint expressions resulting from the use of *constrained-parameter*s that declare template parameter packs. This feature is not present in ISO/IEC 14882:2014, but it is planned to be included in the next revision of that International Standard. The specification of that feature is included in this document.

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 hereafter called the *C++ Standard*. The numbering of Clauses, sections, and paragraphs in this document reflects the numbering in the C++ Standard. References to Clauses and sections not appearing in this Technical Specification refer to the original, unmodified text in the C++ Standard.

1.3 Terms and definitions [intro.defs]

Modify the definitions of "signature" to include associated constraints (14.10.2). This allows different translation units to contain definitions of functions with the same signature, excluding associated constraints, without violating the one definition rule (3.2). That is, without incorporating the constraints in the signature, such functions would have the same mangled name, thus appearing as multiple definitions of the same function.

1.3.1 [defns.signature]

signature

<function> name, parameter type list (8.3.5), ~~and~~ enclosing namespace (if any), and any associated constraints (14.10.2)

[*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 (8.3.5), enclosing namespace (if any), return type, ~~and~~ template parameter list, and any associated constraints (14.10.2)

1.3.3 [defns.signature.member]

signature
<class member function> name, parameter type list (8.3.5), class of which the function is a member, *cv*-qualifiers (if any), ~~and~~ *ref-qualifier* (if any), and any associated constraints (14.10.2)

1.3.4 **[defns.signature.member.templ]**
signature
<class member function template> name, parameter type list (8.3.5), 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 (14.10.2)

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 Feature-testing recommendations [intro.features]

1 An implementation that provides support for this Technical Specification shall define the feature test macro(s) in Table A.

Table A — Feature-test macro(s)

Macro name	Value
`__cpp_concepts`	201507

1.6 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 (5.1.4, 5.2.8, 5.3.3, 5.3.7).

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:
  literal
  `this`
  `(` *expression* `)`
  id-expression
  lambda-expression
  fold-expression
  requires-expression

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

8 *nested-name-specifier:*
  `::`
  type-name `::`
  namespace-name `::`
  decltype-specifier `::`
  `auto ::`
  constrained-type-name `::`
  nested-name-specifier *identifier* `::`
  nested-name-specifier `template`$_{opt}$`::`

Add a new paragraph after paragraph 11:

12 In a *nested-name-specifier* of the form `auto::` or `C::`, where `C` is a *constrained-type-name*, that *nested-name-specifier* designates a placeholder that will be replaced later according to the rules for placeholder deduction in 7.1.6.4. If a placeholder designated by a *constrained-type-specifier* is not a placeholder type, the program is ill-formed. [*Note:* A *constrained-type-specifier* can designate a placeholder for a non-type or template (7.1.6.4.2). — *end note*] The replacement type deduced for a placeholder shall be a class or enumeration type. [*Example:*

```
template<typename T> concept bool C = sizeof(T) == sizeof(int);
template<int N> concept bool D = true;

struct S1 { int n; };
struct S2 { char c; };
struct S3 { struct X { using Y = int; }; };

int auto::* p1 = &S1::n; // auto deduced as S1
int D::* p2 = &S1::n;    // error: D does not designate a placeholder type
int C::* p3 = &S1::n;    // OK: C deduced as S1
char C::* p4 = &S2::c;   // error: deduction fails because constraints are not satisfied
```

```
void f(typename auto::X::Y);
f(S1()); // error: auto cannot be deduced from S1()
f<S3>(0); // OK
```

In the declaration of `f`, the placeholder appears in a non-deduced context (14.8.2.5). It may be replaced later through the explicit specification of template arguments. — *end example*]

Add a new paragraph after paragraph 13:

14 A program that refers explicitly or implicitly to a function with associated constraints that are not satisfied (14.10.2), other than to declare it, is ill-formed. [*Example:*

```
void f(int) requires false;

f(0);                          // error: cannot call f
void (*p1)(int) = f;           // error: cannot take the address of f
decltype(f)* p2 = nullptr;     // error: the type decltype(f) is invalid
```

In each case the associated constraints of `f` are not satisfied. In the declaration of `p2`, those constraints are required to be satisfied even though `f` is an unevaluated operand (Clause 5). — *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 one or more placeholders (7.1.6.4) appear in the parameter-type-list 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 (8.3.5). 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.~~ The closure type for a generic lambda has a public inline function call operator member template that is an abbreviated function template whose parameters and return type are derived from the *lambda-expression*'s *parameter-declaration-clause* and *trailing-return-type* according to the rules in (8.3.5).

Add the following example after those in paragraph 5 in the C++ Standard.

[*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 Fold expressions [expr.prim.fold]

Add this section after 5.1.2.

1 A fold expression performs a fold of a template parameter pack (14.6.3) over a binary operator.

fold-expression:
`(` *cast-expression fold-operator* `...` `)`
`(` `...` *fold-operator cast-expression* `)`
`(` *cast-expression fold-operator* `...` *fold-operator cast-expression* `)`

fold-operator: one of

```
+    -    *    /    %    ^    &    |    <<    >>
+=   -=   *=   /=   %=   ^=   &=   |=   <<=   >>=   =
==   !=   <    >    <=   >=   &&   ||   ,     .*    ->*
```

2 An expression of the form `(... ` *op* `e)` where *op* is a *fold-operator* is called a *unary left fold*. An expression of the form `(e` *op* `...)` where *op* is a *fold-operator* is called a *unary right fold*. Unary left folds and unary right folds are collectively called *unary folds*. In a unary fold, the *cast-expression* shall contain an unexpanded parameter pack (14.6.3).

3 An expression of the form `(e1` *op1* `...` *op2* `e2)` where *op1* and *op2* are *fold-operator*s is called a *binary fold*. In a binary fold, *op1* and *op2* shall be the same *fold-operator*, and either `e1` shall contain an unexpanded parameter pack or `e2` shall contain an unexpanded parameter pack, but not both. If `e2` contains an unexpanded parameter pack, the expression is called a *binary left fold*. If `e1` contains an unexpanded parameter pack, the expression is called a *binary right fold*. [*Example:*

```
template<typename ...Args>
bool f(Args ...args) {
  return (true && ... && args); // OK
}

template<typename ...Args>
bool f(Args ...args) {
  return (args + ... + args); // error: both operands contain unexpanded parameter packs
}
```

— *end example*]

5.1.4 Requires expressions [expr.prim.req]

Add this section to 5.1.

1 A *requires-expression* provides a concise way to express requirements on template arguments. A 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*$_{opt}$ *requirement-body*

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

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

requirement-seq:
 requirement
 requirement-seq requirement

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

2 A *requires-expression* defines a constraint (14.10) based on its parameters (if any) and its nested *requirement*s.

3 A *requires-expression* has type `bool` and is an unevaluated expression (5). [*Note:* A *requires-expression* is transformed into a constraint in order to determine if it is satisfied (14.10.2). — *end note*]

4 A *requires-expression* shall appear only within a concept definition (7.1.7), or within the *requires-clause* of a *template-declaration* (Clause 14) or function declaration (8.3.5). [*Example:* A common use of *requires-expression*s is to define requirements in concepts such as the one below:

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

A *requires-expression* can also be used in a *requires-clause* 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 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* (8.3.5). 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 *requirements*. The *parameter-declaration-clause* of a *requirement-parameter-list* shall not terminate with an ellipsis. [*Example:*

```
template<typename T>
  concept bool C1() {
    requires(T t, ...) { t; }; // error: terminates with an ellipsis
  }

template<typename T>
  concept bool C2() {
    requires(T t, void (*p)(T*, ...)) // OK: the parameter-declaration-clause of
    { p(t); };                        // the requires-expression does not terminate
  }                                   // with an ellipsis
```

— *end example*]

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 a constraint (14.10) to the conjunction of constraints defined by the *requires-expression*. Constraints are appended in the order in which they are written.

7 The substitution of template arguments into a *requires-expression* may result in the formation of invalid types or expressions in its requirements. In such cases, the constraints corresponding to those requirements are not satisfied; it does not cause the program to be ill-formed. If the substitution of template arguments into a *requirement* would always result in a substitution failure, the program is ill-formed; no diagnostic required. [*Example:*

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

— *end example*]

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

simple-requirement:
    expression ;

1 A *simple-requirement* introduces an expression constraint (14.10.1.3) for its *expression*. [*Note:* An expression constraint asserts the validity of an expression. — *end note*]

[*Example:*

```
template<typename T> concept bool C =
  requires (T a, T b) {
    a + b;  // an expression constraint for a + b
  };
```

— *end example*]

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

type-requirement:
    `typename` *nested-name-specifier*$_{opt}$ *type-name* ;

1 A *type-requirement* introduces a type constraint (14.10.1.4) for the type named by its optional *nested-name-specifier* and *type-name*. [*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 member name
    typename S<T>;     // required class template specialization
    typename Ref<T>;   // required alias template substitution
  };
```

— *end example*]

5.1.4.3 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 constraints for the *expression* `E`. The order in which those constraints are introduced is:

(1.1) — the *compound-requirement* introduces an expression constraint for `E` (14.10.1.3);

(1.2) — if the `noexcept` specifier is present, the *compound-requirement* appends an exception constraint for `E` (14.10.1.7);

(1.3) — if the *trailing-return-type* is present, the *compound-requirement* appends one or more constraints derived from the type `T` named by the *trailing-return-type*:

(1.3.1) — if `T` contains one or more placeholders (7.1.6.4), the requirement appends a deduction constraint (14.10.1.6) of `E` against the type `T`.

(1.3.2) — otherwise, the requirement appends two constraints: a type constraint on the formation of `T` (14.10.1.4) and an implicit conversion constraint from `E` to `T` (14.10.1.5).

[*Example:*

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

The *compound-requirement* in `C1` introduces an expression constraint for `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 for `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: an 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*]

5.1.4.4 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 a predicate constraint (14.10.1.2) for 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` (14.10.1.2). — *end example*] [*Note:* The *constraint-expression* of the predicate constraint introduced by a *nested-requirement* is later normalized for the purposes of determining constraint satisfaction (14.10.2) and partial ordering (14.10.3). — *end note*]

7 Declarations [dcl.dcl]

7.1 Specifiers [dcl.spec]

Extend the *decl-specifier* production in paragraph 1 to include the `concept` specifier.

1 The specifiers that can be used in a declaration are

> *decl-specifier:*
> *storage-class-specifier*
> *type-specifier*
> *function-specifier*
> `friend`
> `typedef`
> `constexpr`
> `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:*
> *nested-name-specifier*$_{opt}$ *type-name*
> *nested-name-specifier* `template` *simple-template-id*
> `char`
> `char16_t`
> `char32_t`
> `wchar_t`
> `bool`
> `short`
> `int`
> `long`
> `signed`
> `unsigned`
> `float`
> `double`
> `void`
> `auto`
> *decltype-specifier*
> *constrained-type-specifier*

Modify paragraph 2 to begin:

1 ~~The `auto` specifier is a placeholder for a type to be deduced (7.1.6.4).~~ The `auto` specifier and *constrained-type-specifier*s are placeholders for values (type, non-type, template) to be deduced (7.1.6.4).

Add *constrained-type-specifier*s to the table of *simple-type-specifier*s in Table 10.

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

Extend this section to allow for *constrained-type-specifier*s as a new syntax for designating placeholders. The section is refactored so that placeholders are introduced in this section, deduction rules are defined in subsection 7.1.6.4.1, and the meaning of *constrained-type-specifier*s is described in 7.1.6.4.2.

Table 10 — *simple-type-specifier*s and the types they specify

Specifier(s)	Type
type-name	the type named
simple-template-id	the type as defined in 14.3
⋮	⋮
auto	placeholder for a type to be deduced
decltype(*expression*)	the type as defined below
constrained-type-specifier	placeholder for value (type, non-type, template) to be deduced

Replace paragraph 1 with the text below.

1 The *type-specifier*s `auto` and `decltype(auto)` and *constrained-type-specifier*s designate a placeholder (type, non-type, or template) that will be replaced later, either through deduction or an explicit specification. The `auto` and `decltype(auto)` *type-specifier*s designate placeholder types; a *constrained-type-specifier* can also designate placeholders for values and templates. [*Note:* The deduction of placeholders is done through the invention of template parameters as described in 7.1.6.4.1 and 8.3.5. — *end note*] Placeholders are also used to signify that a lambda is a generic lambda (5.1.2), that a function declaration is an abbreviated function template (8.3.5), or that a *trailing-return-type* in a *compound-requirement* (5.1.4.3) introduces an argument deduction constraint (14.10.1.6). [*Note:* A *nested-name-specifier* can also include placeholders (5.1). Replacements for those placeholders are determined according to the rules in this section. — *end note*]

Modify paragraph 2 to allow *constrained-type-specifier*s with function declarators, except in the declared return type.

2 ~~The placeholder type~~Placeholders 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* (8.3.5), that specifies the declared return type of the function. If the declared return type of the function contains a placeholder, the return type of the function is deduced from return statements in the body of the function, if any. In a function declarator of the form `auto D -> T` where `T` contains placeholders, the initial `auto` does not designate a placeholder.

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

3 If ~~the `auto` *type-specifier*~~ a placeholder 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 a placeholder appears in a parameter type of a function declaration, the function declaration declares an abbreviated function template (8.3.5). [*Example:*

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

— *end example*]

Add the following after paragraph 3 to describe when *constrained-type-specifier*s in the return type refer to template parameters.

4 A *constrained-type-specifier* `C1` within the declared return type of an abbreviated function template declaration does not designate a placeholder if its introduced *constraint-expression* (7.1.6.4.2) is determined to be equivalent, using the rules in 14.6.6.1 for comparing expressions, to the introduced *constraint-expression* for a *constrained-type-specifier* `C2` in the *parameter-declaration-clause* of that function declaration. Instead, `C1` is replaced by the template parameter invented for `C2` (8.3.5). [*Example:*

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

template<typename... T> struct Tuple;

C const& f1(C);       // has one template parameter and no deduced return type
Tuple<C...> f2(C);    // has one template parameter and a deduced return type
```

In the declaration `f1`, the *constraint-expression* introduced by the *constrained-type-specifiers* in the *parameter-declaration-clause* and return type are equivalent ; they would both introduce the expression `C<T>`, for some invented template parameter `T`. In `f2`, the use of `C` in the return type would introduce the *constraint-expression* `(C<T> && ...)`, which is distinct from the *constraint-expression* `C<T>` introduced by the invented *constrained-parameter* (14.1) for the *constrained-type-specifier* in the *parameter-declaration-clause* according to the rules in 8.3.5. — *end example*]

Add the following after paragraph 4 to allow the use of `auto` in the *trailing-return-type* of a *compound-requirement*. Also, disallow the use of `decltype(auto)` with function parameters and deduction constraints.

5 If a placeholder appears in the *trailing-return-type* of a *compound-requirement* in a *requires-expression* (5.1.4.3), that return type introduces an argument deduction constraint (14.10.1.6). [*Example:*

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

— *end example*]

6 The `decltype(auto)` *type-specifier* shall not appear in the declared type of a *parameter-declaration* or the *trailing-return-type* of a *compound-requirement.*

Modify paragraph 4 (paragraph 7, here) to allow multiple placeholders within a variable declaration, but disallowing *constrained-type-specifiers*.

7 The type of a variable declared using a placeholder ~~`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*~~ A placeholder 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* of such a variable 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*. [*Example:*

```
auto x = 5;                 // OK: x has type int
const auto *v = &x, u = 6; // OK: v has type const int*, u has type const int
```

```
static auto y = 0.0;        // OK: y has type double
auto int r;                 // error: auto is not a storage-class-specifier
auto f() -> int;            // OK: f returns int
auto g() { return 0.0; }    // OK: g returns double
auto h();                   // OK: h's return type will be deduced when it is defined
```

— *end example*]

Add the following declarations to the example in the previous paragraph.

```
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) { }  };

void (auto::* p1)(auto) = &Size<0>::f;   // OK: p1 has type void(Size<0>::*)(int)
Pair<auto, auto> p2 = make_pair(0, 'a'); // OK: p2 has type Pair<int, char>
N::Wrap<auto> a = N::make_wrap(0.0);     // OK: a has type Wrap<double>
auto::Wrap<int> x = N::make_wrap(0);     // error: failed to deduce value for auto
Size<sizeof(auto)> y = Size<0>{};        // error: failed to deduce value for auto

template<typename T> concept bool C = true;
template<typename T> concept bool D = false;
C z1 = 0;                  // OK: z1 has type int
D z2 = 0;                  // error: constraints not satisfied
C cf1() { return 0.0; };   // OK: cf1 returns double
D cf2() { return 0.0; };   // error: constraints not satisfied
auto cf3() -> C;           // OK: cf3's return type will be deduced when it is defined
```

Update paragraph 6 (paragraph 9, here) to disallow placeholders in other contexts.

8 A program that uses ~~auto or decltype(auto)~~ placeholders in a context not explicitly allowed in this section is ill-formed.

7.1.6.4.1 Deducing replacements for variables and return types [dcl.spec.auto.deduct]

Factor the deduction rules for `auto` into a new subsection.

1 When a variable declared using a placeholder is initialized, or a `return` statement occurs in a function declared with a return type that contains a placeholder, 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* or a *constrained-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:

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

(1.2) — otherwise, replace each occurrence of a placeholder in the variable or return type with a new invented type template parameter according to the rules for inventing template parameters for placeholders in 8.3.5.

Deduce a value for ~~U~~ each invented type template 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. If any placeholders in the declared type were introduced by a *constrained-type-specifier*, then define C to be a *constraint-expression* as follows:

(1.3) — if there is single *constrained-type-specifier*, then C is the *constraint-expression* introduced by the invented template *constrained-parameter* (14.1) corresponding to that *constrained-type-specifier*;

(1.4) — otherwise, C is the *logical-and-expression* (5.14) whose operands are the *constraint-expression*s introduced by the invented template *constrained-parameter*s corresponding to each *constrained-type-specifier*, in order of appearance.

If the normalized constraint for C (14.10.2) is not satisfied by the values deduced for the declared type, 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:*

```
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 in the C++ Standard.

[*Example:*

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

template<typename... Ts> struct Tuple { };
template<typename... Ts> auto make_tup(Ts... args) { return Tuple<Ts...>{}; }

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: type deduction fails
Tuple<auto...> v3 = make_tup(0, 'a'); // OK: decltype(v3) is Tuple<int, char>
```

— *end example*]

Add the following after the second example in paragraph 7 in the C++ Standard.

[*Example:*

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

— *end example*]

[*Example:*

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

const C* cv = expr;
```

The type of `cv` is deduced from the parameter `p1` in the call `f1(expr)` of the following invented function:

```
template<C T> void f1(const T* p1);
```

— *end example*]

[*Example:*

```
auto cf(int) -> Pair<C, C> { return expr; }
```

The return type of `cf` is deduced from the parameter `p2` in the call `f2(expr)` of the following invented function:

```
template<C T> void f2(Pair<T, T>);
```

Both *constrained-type-specifier*s in the return type of `cf` correspond to the same invented template parameter. — *end example*]

Copy paragraphs 8-15 from 7.1.6.4 in the C++ Standard into this section. Modify paragraph 8 (here, 2) to read:

2 If the *init-declarator-list* contains more than one *init-declarator*, they shall all form declarations of variables. The type of each declared variable is determined as described above, and if the type that replaces ~~the placeholder type~~ the declared variable type or return type is not the same in each deduction, the program is ill-formed.

Add add the following examples to that paragraph.

[*Example:*

```
Pair<auto, auto> p1 = make_pair(0, 0),
                 *p2 = &p1;             // OK: replacement type is Pair<int, int>
Pair<auto, auto> p3 = make_pair(0, 'a'),
                 p4 = make_pair('a', 0); // error: different replacement types
```

— *end example*]

Modify paragraph 9 (here, 3).

3 If a function with a declared return type that contains ~~a placeholder type~~ placeholders has multiple return statements, the return type is deduced for each return statement. If the type deduced is not the same in each deduction, the program is ill-formed.

Modify the text of paragraph 10 (here, 4).

4 If a function with a declared return type that uses ~~a placeholder type~~ placeholders has no return statements, the return type is deduced as though from a return statement with no operand at the closing brace of the function body.

Modify the first sentence of paragraph 11 (here, 5).

5 If the type of an entity with an undeduced placeholder ~~type~~ is needed to determine the type of an expression, the program is ill-formed.

Modify the text of paragraph 13 (here, 7).

7 Redeclarations or specializations of a function or function template with a declared return type that uses ~~a placeholder type~~ placeholders shall also use that placeholder, not a deduced type. If a placeholder is designated by a *constrained-type-specifier*, redeclarations or specializations shall use the same *constrained-type-specifier*.

Add the following examples to that paragraph.

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

template<typename T> auto cf(T) -> C1; // #1
template<typename T> C1 cf(T);         // #2, redeclaration of #1
template<typename T> C2 cf(T);         // error: redeclared with different placeholder
```

Modify paragraph 14 (here, 8).

8 A function declared with a return type that uses ~~a placeholder type~~ placeholders shall not be virtual (10.3)

Modify paragraph 15 (paragraph 8, here) to read:

8 An explicit instantiation declaration (14.8.2) does not cause the instantiation of an entity declared using ~~a placeholder type~~ placeholders, but it also does not prevent that entity from being instantiated as needed to determine its type.

7.1.6.4.2 Constrained type specifiers [dcl.spec.auto.constr]

Add this section to 7.1.6.4.

1 A *constrained-type-specifier* designates a placeholder (type, non-type, or template) and introduces an associated constraint (14.10.2).

constrained-type-specifier:
  qualified-concept-name

qualified-concept-name:
  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*$_{opt}$`>`

[*Example:*

```
template<typename T> concept bool C1 = false;
template<int N> concept bool C2 = false;
template<template<typename> class X> concept bool 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);               // 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*]

2 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 set of concepts referred to by C includes both #1 and #2;
           // concept resolution (14.10.4) selects #1.
void g(D); // OK: the concept-name D refers only to #3
```

— *end example*]

3 A *partial-concept-id* is a *concept-name* followed by a sequence of *template-argument*s. [*Example:*

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

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

— *end example*]

4 The concept designated by a *constrained-type-specifier* is the one selected according to the rules for concept resolution in 14.10.4.

5 [*Note:* The *constraint-expression* introduced by a *constrained-type-name* is the one introduced by the invention of a *constrained-parameter* (14.1). The rules for inventing template parameters corresponding to placeholders in the type of a variable or the declared return type of a function are described in 7.1.6.4.1. The rules for inventing template parameters corresponding to placeholders in the *parameter-declaration-clause* of a *lambda-expression* (5.1.2) or function declaration (8.3.5) are described in 8.3.5. The rules for inventing a template parameter corresponding to placeholders in the *trailing-return-type* of a *compound-requirement* are described in 14.10.1.6. — *end note*]

7.1.7 concept specifier [dcl.spec.concept]

Add this 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 shall have no *exception-specification* and is treated as if it were specified with `noexcept(true)` (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). A concept definition shall not be declared with the `thread_local`, `inline`, `friend`, or `constexpr` specifiers, nor shall a concept definition have associated constraints (14.10.2).

3 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]

4 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.

5 A function concept has the following restrictions:

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

(5.2) — The declared return type shall have the type `bool`.

(5.3) — The declaration's parameter list shall be equivalent to an empty parameter list.

(5.4) — The declaration shall have a *function-body* equivalent to `{ return E; }` where `E` is a *constraint-expression* (14.10.1.3).

[*Note:* Return type deduction requires the instantiation of the function definition, but concept definitions are not instantiated; they are normalized (14.10.2). *— end note*] [*Example:*

```
template<typename T>
  concept int F1() { return 0; }      // error: return type is not bool
template<typename T>
  concept auto F2() { return true; }  // error: return type is deduced
template<typename T>
  concept bool F3(T) { return true; } // error: not an empty parameter list
```

— *end example*]

6 A variable concept has the following restrictions:

(6.1) — The declared type shall have the type `bool`.

(6.2) — The declaration shall have an initializer.

(6.3) — 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 V2 = 0;       // error: not a template

template<typename T> concept bool C = true;

template<C T>
  concept bool V3 = true;  // error: constrained template declared as a concept
```

— *end example*]

7 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 *declarator*s to allow the specification of constraints on function declarations.

declarator:
ptr-declarator
noptr-declarator parameters-and-qualifiers trailing-return-type *$requires\text{-}clause_{opt}$*

parameters-and-qualifiers:
(*parameter-declaration-clause*) *$cv\text{-}qualifier\text{-}seq_{opt}$*
$ref\text{-}qualifier_{opt}$ $exception\text{-}specification_{opt}$ $attribute\text{-}specifier\text{-}seq_{opt}$ $requires\text{-}clause_{opt}$

Add the following paragraphs at the end of this section.

4 The optional *requires-clause* (14.10.2) in a *declarator* shall be present only when the declarator declares a function (8.3.5), and that *requires-clause* shall not precede a *trailing-return-type*. When present in a *declarator*, the *requires-clause* is called the *trailing requires-clause*. [*Example:*

```
void f1(int a) requires true;          // OK
auto f2(int a) -> bool requires true;  // OK
auto f3(int a) requires true -> bool;  // error: requires-clause precedes trailing-return-type
void (*pf)() requires true;            // error: constraint on a variable
void g(int (*)() requires true);       // error: constraint on a parameter-declaration

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

— end example]

8.3 Meaning of declarators [dcl.meaning]

8.3.5 Functions [dcl.fct]

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

1 D1 (*parameter-declaration-clause*) *$cv\text{-}qualifier\text{-}seq_{opt}$*
$ref\text{-}qualifier_{opt}$ $exception\text{-}specification_{opt}$ $attribute\text{-}specifier\text{-}seq_{opt}$ $requires\text{-}clause_{opt}$

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

2 D1 (*parameter-declaration-clause*) *$cv\text{-}qualifier\text{-}seq_{opt}$*
$ref\text{-}qualifier_{opt}$ $exception\text{-}specification_{opt}$ $attribute\text{-}specifier\text{-}seq_{opt}$ $requires\text{-}clause_{opt}$

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 (14.10.2).

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 C++ Standard.

6 The return type, the parameter-type-list, the *ref-qualifier*, and the *cv-qualifier-seq*, but not the default arguments (8.3.6), associated constraints (14.10.2), 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 ~~`auto`~~ a placeholder (7.1.6.4); 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). 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, according to the rules below. [*Note:* Template parameters are also invented to deduce the type of a variable or the return type of a function when the declared type contains placeholders (7.1.6.4.1). — *end note*]

17 Each template parameter is invented as follows.

(17.1) — If the placeholder is designated by the `auto` *type-specifier*, then the invented template parameter is a type *template-parameter*.

(17.2) — Otherwise, the placeholder is designated by a *constrained-type-specifier*, and the invented parameter is a *constrained-parameter* (14.1) whose *qualified-concept-name* is that of the *constrained-type-specifier*.

(17.3) — If the placeholder appears in the *decl-specifier-seq* of a function parameter pack (14.6.3), or the *type-specifier-seq* of a *type-id* that is a pack expansion, the invented template parameter is a template parameter pack.

All placeholders designated by *constrained-type-specifiers* whose corresponding *constrained-parameters* would introduce equivalent *constraint-expressions* (14.1), using the rules for comparing expressions in 14.6.6.1, 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 abbr(C, D<0>);
```

The *constrained-type-specifiers* `C` and `D<0>` correspond to distinct invented template parameters in the declaration of `abbr`.

```
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 `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-specifiers* `E`, `E<>`, and `E<0>` all associate the *constraint-expression* `E<T, 0>`, where `T` is an invented template type parameter.

```
void f5(C head, C... tail);
```

The types of `head` and `tail` are different. Their respective introduced *constraint-expressions* are `C<T>` and `(C<U> && ...)`, where `T` is the template parameter invented for `head` and `U` is the template parameter invented for `tail` (14.1).

18 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. [*Note:* Equivalent function template declarations declare the same function template (14.6.6.1). — *end note*] [*Example:*

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

void f1(const auto&, auto);
void f2(Vec<auto*>...);
void f3(Tuple<auto...>);
void f4(auto (auto::*)(auto));

template<typename T, typename U> void f1(const T&, U);              // redeclaration of f1
template<typename... T> void f2(Vec<T*>...);                        // redeclaration of f2
template<typename... Ts> void f3(Tuple<Ts...>);                     // redeclaration of f3
template<typename T, typename U, typename V> void f4(T (U::*)(V)); // redeclaration of f4

template<typename T> concept bool C1 = true;
template<typename T> concept bool C2 = true;
template<typename T, typename U> concept bool C3 = true;
template<typename... Ts> concept bool C4 = true;

void g1(const C1*, C2&);
void g2(Vec<C1>&);
void g3(C1&...);
void g4(Vec<C3<int>>);
void g5(C4...);
void g6(Tuple<C4...>);
void g7(C4 p);
void g8(Tuple<C4>);

template<C1 T, C2 U> void g1(const T*, U&); // redeclaration of g1
template<C1 T> void g2(Vec<T>&);            // redeclaration of g2
template<C1... Ts> void g3(Ts&...);         // redeclaration of g3
template<C3<int> T> void g4(Vec<T>);        // redeclaration of g4
template<C4... Ts> void g5(Ts...);          // redeclaration of g5
template<C4... Ts> void g6(Tuple<Ts...>);   // redeclaration of g6
template<C4 T> void g7(T);                  // redeclaration of g7
template<C4 T> void g8(Tuple<T>);           // redeclaration of g8
```

— *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*]

19 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: redeclaration of f(Array<auto, N>*)
```

— *end example*]

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 (14.10.2), the program is ill-formed. [*Example:*

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

— 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 (14.10.2) 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 8.3.5, function declarations that have equivalent parameter declarations and associated constraints, if any (14.10.2), 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 (14.10.2).

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 (14.10.2), 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, those constraints shall be satisfied (14.10.2).

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 C++ Standard 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 associated constraints are not satisfied (14.10.2). ~~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)() = &f;          // selects #2
void (*pg)() = &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). The optional *requires-clause* following a *template-parameter-list* allows the specification of constraints (14.10.2) on template arguments (14.4).

14.1 Template parameters [temp.param]

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

1 *template-parameter:*
type-parameter
parameter-declaration
constrained-parameter

constrained-parameter:
qualified-concept-name ... *identifier*$_{opt}$
qualified-concept-name *identifier*$_{opt}$ *default-template-argument*$_{opt}$

default-template-argument:
`=` *type-id*
`=` *id-expression*
`=` *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 a *parameter-declaration*. If the *type-specifier-seq* of a *parameter-declaration* is a *constrained-type-specifier* (7.1.6.4.2), 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 the *qualified-concept-name* (7.1.6.4.2) in the *constrained-parameter*. The designated concept is selected by the rules for concept resolution described in 14.10.4. Let `X` be the prototype parameter of the designated concept. The declared template parameter is determined by the kind of `X` (type, non-type, template) and the optional ellipsis in the *constrained-parameter* as follows.

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

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

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

(9.4) — If the *qualified-concept-name* is followed by an ellipsis, then the declared parameter is a template parameter pack (14.6.3).

[*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 T> void f5();      // OK: T is a type template-parameter
template<C5... Cs> f6();       // OK: Cs is a template parameter pack of chars
```

— *end example*]

10 A *constrained-parameter* introduces a *constraint-expression* (14.10.2). The expression is derived from the *qualified-concept-name* `Q` in the *constrained-parameter*, its designated concept `C`, and the declared template parameter `P`.

(10.1) — First, form a template argument `A` from `P`. If `P` declares a template parameter pack (14.6.3) and `C` is a variadic concept (7.1.7), then `A` is the pack expansion `P...`. Otherwise, `A` is the *id-expression* `P`.

(10.2) — Then, form a *template-id* `TT` based on the *qualified-concept-name* `Q`. If `Q` is a *concept-name*, then `TT` is `C<A>`. Otherwise, `Q` is a *partial-concept-id* of the form `C<A1, A2, ..., AN>`, and `TT` is `C<A, A1, A2, ..., AN>`.

(10.3) — Then, form an *expression* `E` as follows. If `C` is a variable concept (7.1.7), then `E` is the *id-expression* `TT`. Otherwise, `C` is a function concept and `E` is the function call `TT()`.

(10.4) — Finally, if `P` declares a template parameter pack and `C` is not a variadic concept, `E` is adjusted to be the *fold-expression* `(E && ...)` (5.1.3).

`E` is the introduced *constraint-expression*. [*Example:*

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

template<C1 T> struct s1;      // associates C1<T>
template<C1... T> struct s2;   // associates (C1<T> && ...)
template<C2... T> struct s3;   // associates C2<T...>()
template<C3<int> T> struct s4; // associates C3<T, int>
```

— *end example*]

Insert the following paragraph after paragraph 9 to require that the kind of a *default-argument* matches the kind of its *constrained-parameter*.

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

template-introduction:
  qualified-concept-name { *introduction-list* }

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

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

A *template-introduction* declares a template whose sequence of *template-parameter*s are derived from a *qualified-concept-name* (7.1.6.4.2) and the sequence of *introduced-parameter*s in its *introduction-list*.

2 The concept designated by the *qualified-concept-name* is selected by the concept resolution rules described in 14.10.4. Let `C` be the designated concept. The template parameters declared by a *template-introduction* are derived from its *introduced-parameter*s and the template parameter declarations of `C` to which those *introduced-parameter*s are matched as wildcards according to the rules in 14.10.4. For each *introduced-parameter* `I`, declare a template parameter using the following rules:

(2.1) — Let `P` be the template parameter declaration in `C` corresponding to `I`. If `P` does not declare a template parameter pack (14.6.3), `I` shall not include an ellipsis.

(2.2) — If `P` declares a template parameter pack, adjust `P` to be the pattern of that pack.

(2.3) — Declare a template parameter according to the rules for declaring a *constrained-parameter* in 14.1, using `P` as the prototype parameter and with no ellipsis.

(2.4) — If `I` includes an ellipsis, then the declared template parameter is 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;
template<typename... Ts> concept bool C3 = true;

C1{A, B, ...C} // OK: A is declared as typename A,
  struct S1;   // B is declared as int B, and
```

```
                    // C is declared as typename ... C

C2{T} void f();     // OK: T is declared as template<typename> class T
C2{...Ts} void g(); // error: the template parameter corresponding to Ts
                    // is not a template parameter pack

C3{T} struct S2;      // OK: T is declared as typename T
C3{...Ts} struct S2; // OK: Ts is declared as typename ... Ts
```

— *end example*]

3 A concept referred to by a *qualified-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. 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
```

— *end example*]

4 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*]

5 A *template-introduction* introduces a *constraint-expression* (14.10.2). This expression is derived from the *qualified-concept-name* `C` in the *template-introduction* and the sequence of *introduced-parameter*s.

(5.1) — First, form a sequence of template arguments `A1, A2, ..., A`N corresponding to the *introduced-parameter*s `P1, P2, ..., P`N. For each *introduced-parameter* `P`, form a corresponding template argument `A` as follows. If `P` includes an ellipsis, then `A` is the pack expansion `P...` (14.6.3). Otherwise, `A` is the *id-expression* `P`.

(5.2) — Then, form an expression `E` as follows. If `C` designates a variable concept (7.1.7), then `E` is the *id-expression* `C<A1, ..., A`N`>`. Otherwise, `C` designates a function concept and `E` is the function call `C<A1, ..., A`N`>()`.

`E` is the introduced *constraint-expression*. [*Example:*

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

C1{A, B} struct s1;     // associates C1<A, B>
C2{A, B} struct s2;     // associates C2<A, B>()
C3{...Ts} struct s3;    // associates C3<Ts...>
C3{X, ...Y} struct s4; // associates C3<X, Y...>
```

— *end example*]

6 A template declared by a *template-introduction* can also be an abbreviated function template (8.3.5). 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;

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

— *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 the *template-name* of a *simple-template-id* names a constrained non-function template or a constrained template *template-parameter*, but not a member template that is a member of an unknown specialization (14.7), and all *template-arguments* in the *simple-template-id* are non-dependent 14.6.2.4, the associated constraints of the constrained template shall be satisfied. (14.10.2). [*Example:*

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

template<C1 T> struct S1 { };
template<C1 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<C1 T> class X>
  struct S4 {
    X<int> x; // error: constraints not satisfied (#1)
  };

template<typename T>
  struct S5 {
    using Type = typename T::template MT<char>; // #2
  };

template<typename T> concept bool C2 = sizeof(T) == 1;

template<C2 T> struct S { };

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

In #1, the error is caused by the substitution of `int` into the associated constraints of the template parameter `X`. In #2, no constraints can be checked for `typename T::template MT<char>` because `MT` is a member of an unknown specialization. — *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-parameter*s. Note that the examples following this paragraph in the C++ Standard 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-parameter*s, their types are equivalent (14.6.6.1), and for template *template-parameter*s, each of their corresponding *template-parameter*s matches, recursively. When `P`'s *template-parameter-list* contains a template parameter pack (14.6.3), 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 C++ Standard.

[*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 and P have equivalent constraints
S<Y> s2; // error: P is not at least as constrained Y (Y is more constrained than P)
S<Z> s3; // OK: P is at least as constrained as 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 (14.10.2), and *exception-specification*s of function templates and default arguments, associated constraints, and *exception-specification*s 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-specification*s.

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-parameter*s and associated constraints (14.10.2) 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.

1 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 (14.10.2) of the class template followed by the *template-parameters* and associated constraints 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]

14.6.3 Variadic templates [temp.variadic]

Add *fold-expressions* to the list of contexts in which pack expansion can occur.

— ...

— In a *fold-expression* (5.1.3); the pattern is the *cast-expression* that contains an unexpanded parameter pack.

Modify paragraph 7 to exclude *fold-expressions* from producing a comma-separated list of elements.

The instantiation of a pack expansion ~~that is not a `sizeof...` expression~~ that is neither a `sizeof...` expression nor a *fold-expression* produces a list $E_1, E_2, ..., E_N$, where N is the number of elements in the pack expansion parameters. Each E_i is generated by instantiating the pattern and replacing each pack expansion parameter with its ith element.

Add the following paragraphs at the end of this section.

9 The instantiation of a *fold-expression* produces:

(9.1) — $((E_1\ op\ E_2)\ op\ \cdots)\ op\ E_N$ for a unary left fold,

(9.2) — $E_1\ op\ (\cdots\ op\ (E_{N-1}\ op\ E_N))$ for a unary right fold,

(9.3) — $(((E\ op\ E_1)\ op\ E_2)\ op\ \cdots)\ op\ E_N$ for a binary left fold, and

(9.4) — $E_1\ op\ (\cdots\ op\ (E_{N-1}\ op\ (E_N\ op\ E)))$ for a binary right fold.

In each case, *op* is the *fold-operator*, N is the number of elements in the pack expansion parameters, and each E_i is generated by instantiating the pattern and replacing each pack expansion parameter with its ith element. For a binary fold-expression, E is generated by instantiating the *cast-expression* that did not contain an unexpanded parameter pack. [*Example:*

```
template<typename ...Args>
  bool all(Args ...args) { return (... && args); }

bool b = all(true, true, true, false);
```

Within the instantiation of `all`, the returned expression expands to `((true && true) && true) && false`, which evalutes to `false`. — *end example*] If N is zero for a unary fold-expression, the value of the expression is shown in Table B; if the operator is not listed in Table B, the instantiation is ill-formed.

Table B — Value of folding empty sequences

Operator	Value when parameter pack is empty
`*`	`1`
`+`	`int()`
`&`	`-1`
`|`	`int()`
`&&`	`true`
`||`	`false`
`,`	`void()`

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 have associated constraints (14.10.2), 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> void g0(T);
template<C1 T> void g1(T);
template<C2 T> void 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 only 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 (14.9.2.6) 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 (14.8), 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). If substitution fails, the program is ill-formed. — *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.

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

(8.1) — ...

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

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

(8.4) — ...

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 (14.9.2) , and the deduced template arguments satisfy the constraints of the partial specialization, if any (14.10.2).

Add the following example to the end of paragraph 2.

[*Example:*

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

A<S, S, 1>    a6; // uses #6
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 (14.10.2) 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 (14.10.2) 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 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, and

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

Two function templates are *functionally equivalent* if they are equivalent except that one or more expressions that involve template parameters in the return types, ~~and~~ parameter lists, and associated constrains (14.10.2) 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 (14.10.2) 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.

14.7.2 Dependent names [temp.dep]

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

Add the following paragraph to this section.

7 A *fold-expression* is type-dependent.

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

Modify paragraph 4 to include *fold-expressions* in the set of value-dependent expressions.

4 Expressions of the following form are value-dependent:

`sizeof ... (` *identifier* `)`
fold-expression

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 (14.10.2), 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 (14.10.2). [*Note:* The satisfaction of constraints is determined during name 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, both constructors 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]

Add the following note after paragraph 7.

8 [*Note:* An explicit instantiation of a constrained template shall satisfy that template's associated constraints (14.10.2). The satisfaction of constraints is determined during name lookup for explicit instantiations in which all template arguments are specified (14.3), or for explicit instantiations of function templates, during template argument deduction (14.9.2.6) when one or more trailing template arguments are left unspecified. — *end note*]

Modify paragraph 8 to ensure that only members whose constraints are satisfied are explicitly instantiated during class template specialization. The note in the C++ Standard 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, if any, of that member are satisfied by the template arguments of the explicit instantiation (14.10.2), except as described below.

14.8.3 Explicit specialization [temp.expl.spec]

Add the following note after paragraph 10.

11 [*Note:* An explicit specialization of a constrained template shall satisfy that template's associated constraints (14.10.2). The satisfaction of constraints is determined during name lookup for explicit specializations in which all template arguments are specified (14.3), or for explicit specializations of function templates, during template argument deduction (14.9.2.6) when one or more trailing template arguments are left unspecified. — *end note*]

14.9 Function template specializations [temp.fct.spec]

14.9.2 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 (14.10.2), 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*]

14.9.2.6 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 (14.10.2).

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.

Add the following example to paragraph 3.

[*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*]

14.10 Template constraints [temp.constr]

Add this section after 14.9.

1 [*Note:* This section defines the meaning of constraints on template arguments. The abstract syntax, satisfaction rules, and equivalence rules are defined in 14.10.1. Constraints are associated with declarations in 14.10.2. Declarations are partially ordered by their associated constraints (14.10.3). — *end note*]

14.10.1 Constraints [temp.constr.constr]

1 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*] There are several different kinds of constraints:

(1.1) — conjunctions (14.10.1.1),

(1.2) — disjunctions (14.10.1.1),

(1.3) — predicate constraints (14.10.1.2),

(1.4) — expression constraints (14.10.1.3),

(1.5) — type constraints (14.10.1.4),

(1.6) — implicit conversion constraints (14.10.1.5),

(1.7) — argument deduction constraints (14.10.1.6),

(1.8) — exception constraints (14.10.1.7), and

(1.9) — parameterized constraints (14.10.1.8)

2 In order for a constrained template to be instantiated (14.8), its associated constraints shall be *satisfied* (14.10.2). [*Note:* The satisfaction of constraints on class templates, alias templates, and variable templates is required when referring to a template specialization (14.3). The satisfaction of constraints on functions and function templates is required during overload resolution (13.3.2). The satisfaction of constraints introduced by *constrained-type-specifier*s (7.1.6.4.2) in the type of a variable or declared return type of a function is required when values are deduced for their corresponding placeholders (7.1.6.4). — *end note*] Determining if a constraint is satisfied entails the substitution of template arguments into that constraint. The rules for determining the satisfaction of different kinds of constraints are defined in the following subsections.

3 Determining when one declaration is more specialized than another requires knowing when two constraints are equivalent (14.10.3). The rules for determining the equivalence of different kinds of constraints are defined in the following subsections. Two constraints that are not equivalent are *functionally equivalent* if, for any given set of template arguments, either both constraints are satisfied or both constraints are unsatisfied.

14.10.1.1 Logical operations [temp.constr.op]

1 There are two binary 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 $\land$ and disjunction is spelled using the symbol $\lor$. The operands of these operations are called the left and right operands. In the constraint $A \land B$, A is the left operand, and B is the right operand. Grouping of constraints is shown using parentheses. — *end note*]

2 A *conjunction* is a constraint taking two operands. A conjunction of constraints is satisfied if and only if both operands are satisfied. The satisfaction of a conjunction's operands are evaluated left-to-right; if the left operand is not satisfied, template arguments are not substituted into the

right operand, and the constraint is not satisfied. If the left and right operands of a conjunction are predicate constraints (14.10.1.2), let P and Q be the expressions of those constraints resulting from substitution. If the expression `P && Q` results in a call to a user-declared `operator&&`, the program is ill-formed. [*Example:*

```
template<typename T>
  constexpr bool get_value() { return T::value; }

template<typename T>
  requires sizeof(T) > 1 && get_value<T>()
    void f(T);    // has associated constraint sizeof(T) > 1 ∧ get_value<T>()

void f(int);

f('a'); // OK: calls f(int)
```

In the satisfaction of the associated constraints (14.10.2) of `f`, the constraint `sizeof(char) > 1` is not satisfied; arguments are not substituted into the right operand of the conjunction. Such a substitution would cause this program to be ill-formed since `get_value<char>()` produces an invalid expression that is not in the immediate context (14.9.2. — *end example*]

3 A conjunction C is equivalent to another conjunction D if and only if the left operands of C and D are equivalent and the right operands of C and D are equivalent.

4 A *disjunction* is a constraint taking two operands. A disjunction of constraints is satisfied if and only if either operand is satisfied or both operands are satisfied. The satisfaction of a disjunction's operands are evaluated left-to-right; if the left operand is satisfied, template arguments are not substituted into the right operand, and the constraint is satisfied. If the left and right operands of a conjunction are predicate constraints (14.10.1.2), let P and Q be the expressions of those constraints resulting from substitution. If the expression `P || Q` results in a call to a user-declared `operator||`, the program is ill-formed.

5 A disjunction C is equivalent to another disjunction D if and only if the left operands of C and D are equivalent and the right operands of C and D are equivalent.

6 [*Note:* The prohibition against user-declared logical operators disallows *constraint-expressions* (14.10.2) whose evaluation disagrees with the satisfaction of its derived constraint. That is, for any atomic predicate constraints A and B, whose respective expressions are `PA` and `PB`, the conjunction $A \wedge B$ is satisfied if and only if the *constraint-expression* `PA && PB` evaluates to `true`. Likewise, the disjunction $A \vee B$ is satisfied if and only if the *constraint-expression* `PA || PB` evaluates to `true`. — *end note*]

14.10.1.2 Predicate constraints [temp.constr.pred]

1 A *predicate constraint* is a constraint that evaluates a constant expression `E` (5.19). [*Note:* Predicate constraints allow the definition of template requirements in terms of constant expressions. This allows the specification of constraints on non-type template arguments and template template arguments. — *end note*] [*Note:* A predicate constraint is introduced by the *constraint-expression* of a *requires-clause* (14.10.2), or as the associated constraint of a *constrained-parameter* (14.1) or *template-introduction* (14.2). — *end note*] After substitution, `E` shall have type `bool`. The constraint is satisfied if and only if `E` evaluates to `true`. [*Example:*

```
template<typename T>
  concept bool C = sizeof(T) == 4 && !true; // requires predicate constraints
                                             // sizeof(T) == 4 and !true

template<typename T>
```

```
  struct S {
    constexpr explicit operator bool() const { return true; }
  };

template<typename T>
  requires S<T>{}
    void f(T);

f(0); // error: constraints cannot be satisfied because the
      // expression S<int>{} does not have type bool
```

No conversions are applied to predicate constraints. — *end example*]

2 A predicate constraint A is equivalent to another predicate B if and only if the expressions of A and B are equivalent using the rules described in 14.6.6.1 to compare expressions.

14.10.1.3 Expression constraints [temp.constr.expr]

1 An *expression constraint* is a 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* (5.1.4.1) or *compound-requirement* (5.1.4.3) 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 expression `++t` is valid after substituting `int` for `T`. — *end example*]

2 An expression constraint A is equivalent to another expression constraint B if and only if the *expression*s of A and B are equivalent using the rules described in 14.6.6.1 to compare expressions.

14.10.1.4 Type constraints [temp.constr.type]

1 A *type constraint* is a 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 not ill-formed, 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* (5.1.4.2). — *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 A is equivalent to another type constraint B if and only if the types in A and B are equivalent according to the rules in 14.4.

14.10.1.5 Implicit conversion constraints [temp.constr.conv]

1 An *implicit conversion constraint* is a 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 (5.1.4.3). — *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 A is equivalent to another implicit conversion constraint B if and only if the *expressions* of A and B are equivalent using the rules in 14.6.6.1 to compare expressions, and the types of A and B are equivalent according to the rules in 14.4.

14.10.1.6 Argument deduction constraints [temp.constr.deduct]

1 An *argument deduction constraint* is a constraint that specifies a requirement that the type of an *expression* `E` can be deduced from a type `T`, when `T` includes one or more placeholders (7.1.6.4). [*Note:* An argument deduction constraint is introduced by a *compound-requirement* (5.1.4.3) having a *trailing-return-type* that contains one or more placeholders. In such a constraint, `E` is the *expression* of the *compound-requirement*, and `T` is the type specified by the *trailing-return-type*. — *end note*]

2 To determine if an argument deduction constraint is satisfied, invent an abbreviated function template `f` with one parameter whose type is `T` (8.3.5). The constraint is satisfied if the resolution of the function call `f(E)` succeeds (13.3). [*Note:* Overload resolution succeeds when values are deduced for all invented template parameters in `f` that correspond to the placeholders in `T`, and the constraints associated by any *constrained-type-specifier*s are satisfied. — *end note*] [*Example:*

```
template<typename T, typename U> struct Pair;

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

template<typename T>
  concept bool C2() { return requires(T t) { {*t} -> Pair<C1&, auto>; }; }

template<C2 T> void g(T);

g((int*)nullptr); // error: constraints not satisfied.
```

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

```
void f(Pair<C1&, auto>);
```

In the call `g((int*)nullptr)`, the constraints are not satisfied because no values can be deduced for the placeholders `C1` and `auto` from the expression `*t` when `t` has type “pointer-to-`int`”. — *end example*]

3 An argument deduction constraint A is equivalent to another argument deduction constraint B if and only if the *expressions* of A and B are equivalent using the rules in 14.6.6.1 to compare expressions, and the types of A and B are equivalent (14.4).

14.10.1.7 Exception constraints [temp.constr.noexcept]

1 An *exception constraint* is a constraint for an expression `E` that is satisfied if and only if the expression `noexcept(E)` is `true` (5.3.7). [*Note:* Exception constraints are introduced by a *compound-requirement* that includes the `noexcept` specifier (5.1.4.3). — *end note*]

2 An exception constraint A is equivalent to another exception constraint B if and only if the expressions of A and B are equivalent using the rules described in 14.6.6.1 to compare expressions.

14.10.1.8 Parameterized constraints [temp.constr.param]

1 A *parameterized constraint* is a constraint that declares a sequence of parameters (8.3.5), called *constraint variables*, and has a single operand. [*Note:* Parameterized constraints are introduced by *requires-expressions* (5.1.4). The constraint variables of a parameterized constraint correspond to the parameters declared in the *requirement-parameter-list* of a *requires-expression*, and the operand of the constraint is the conjunction of constraints. — *end note*] [*Note:* Parameterized constraints have no corresponding C++ syntax. For the purpose of exposition, a parameterized constraint is written as, e.g., $\lambda(\texttt{T x}, \texttt{U y})\ P(\texttt{x}, \texttt{y})$, where `x` and `y` are constraint variables, and $P(\texttt{x}, \texttt{y})$ is that constraint's operand written in terms of `x` and `y`. — *end note*] [*Example:*

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

The concept `Eq` defines the parameterized constraint $\lambda(\texttt{T a}, \texttt{T b})\ P(\texttt{a}, \texttt{b})$ where $P(\texttt{a}, \texttt{b})$ is the conjunction of two expression constraints: `a == b` and `a != b` must be valid expressions (14.10.1.3). — *end example*]

2 A parameterized constraint is satisfied if and only if substitution into the types of its constraint variables does not result in an invalid type, and its operand is satisfied. Template arguments are substituted into the declared constraint variables in the order in which they are declared. If substitution into a constraint variable fails, no more substitutions are performed, and the constraint is not satisfied.

3 Two parameterized constraints A and B are equivalent if and only if their operands are equivalent. Two expressions involving constraint variables are equivalent if they are equivalent according to the rules for expressions described in 14.6.6.1, except that any *identifier*s referring to constraint variables are equivalent if and only if the types of their corresponding declarations are equivalent (14.4).

4 A constraint variable shall not appear as an evaluated operand (5) of a predicate constraint (14.10.1.2). [*Example:*

```
template<typename T>
  concept bool C = requires (T a) {
    requires sizeof(a) == 4; // OK
    requires a == 0;         // error: evaluation of a constraint variable
  }
```

— *end example*]

14.10.2 Constrained declarations [temp.constr.decl]

1 A template declaration (Clause 14) or function declaration (8.3.5) can be constrained by the use of a *requires-clause*. This allows the specification of constraints for that declaration as an expression:

constraint-expression:
    logical-or-expression

2 Constraints can also be associated with a declaration through the use of *template-introduction*s, *constrained-parameter*s in a *template-parameter-list*, and *constrained-type-specifier*s in the parameter-type-list of a function template. Each of these forms introduces additional *constraint-exprssion*s that are used to constrain the declaration. A template's *associated constraints* are defined as a single *constraint-expression* derived from the introduced *constraint-expression*s using the following rules.

(2.1) — If there are no introduced *constraint-expression*s, the declaration is unconstrained.

(2.2) — If there is a single introduced *constraint-expression*, that is the associated constraint.

(2.3) — Otherwise, the associated constraints are formed as a logical AND expression (5.14) whose operands are in the following order:

(2.3.1) — the *constraint-expression* introduced by a *template-introduction* (14.2), and

(2.3.2) — the *constraint-expression* introduced by each *constrained-parameter* (14.1) in the declaration's *template-parameter-list*, in order of appearance, and

(2.3.3) — the *constraint-expression* introduced by a *requires-clause* following a *template-parameter-list* (Clause 14), and

(2.3.4) — the *constraint-expression* introduced by each *constrained-type-specifier* (7.1.6.4.2) in the type of a *parameter-declaration* in a function declaration (8.3.5), in order of appearance, and

(2.3.5) — the *constraint-expression* of a trailing *requires-clause* (Clause 8) of a function declaration (8.3.5).

The formation of the associated constraints for a template declaration establishes the order in which the normalized constraints (defined below) will be compared for equivalence (to determine when one template redeclares another), and the order in which constraints are instantiated when checking for satisfaction (14.10.1). The *constraint-expression*s introduced by *constrained-type-specifier*s in a variable type or in the declared return type of a function are not included in the associated constraints of a template declaration. [*Note:* These constraints are checked during the instantiation of the declaration. — *end 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. [*Example:*

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

void f1(C);
template<C T> void f1(T);
C{T} void f1(T);
template<typename T> requires C<T> void f1(T);
template<typename T> void f1(T) requires C<T>;
```

All declarations of `f1` declare the same function.

```
template<typename T> concept bool C1 = true;
template<typename T> concept bool C2 = sizeof(T) > 0;

template<C1 T> void f2(T) requires C2<T>;                // #1
template<typename T> requires C1<T> && C2<T> void f2(T); // #2, redeclaration of #1
```

The associated constraints of #1 are `C1<T> && C2<T>`, and those of #2 are also `C1<T> && C2<T>`.

```
template<C1 T> requires C2<T> void f3();
template<C2 T> requires C1<T> void f3(); // 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*]

3 Determining the satisfaction of a declaration's associated constraints, and the partial ordering of declarations by those constraints, requires that they are first *normalized*. Normalization transforms a *expression* into a sequence of conjunctions and disjunctions (14.10.1.1) of *atomic constraints*. The atomic constraints are those that have no constraints as operands: predicate constraints (14.10.1.2), expression constraints (14.10.1.3), type constraints (14.10.1.4), implicit conversion constraints (14.10.1.5), argument deduction constraints (14.10.1.6), and exception constraints (14.10.1.3). The *normal form* of an *expression* `X` is defined as follows. First, create a new expression `E` by replacing all subexpressions in `X` that refer to concepts with their corresponding definitions. In particular,

(3.1) — replace all function calls of the form `C<A1, A2, ..., A`N`>()`, where `C` refers to the function concept `D` (7.1.7) selected by the rules for concept resolution (14.10.4), with the result of substituting `A1, A2, ..., A`N into the expression returned by `D`, and

(3.2) — replace all *id-expressions* of the form `C<A1, A2, ..., A`N`>`, where `C` is the variable concept `D` (7.1.7) selected by the rules for concept resolution (14.10.4), with the result of substituting `A1, A2, ..., A`N into the initializer of `D`.

If any such substitution fails, the program is ill-formed. Second, transform the expression `E` into a constraint as follows:

(3.3) — the normal form of an expression `(E)` is the normal form of `E`;

(3.4) — the normal form of an expression `E1 || E2` is the disjunction (14.10.1.1) of the normal forms of `E1` and `E2`;

(3.5) — the normal form of an expression `E1 && E2` is the conjunction of the normal forms of `E1` and `E2`;

(3.6) — the normal form of a *requires-expression*, (5.1.4) having the form

`requires (` *parameter-declaration-clause* `)` *requirement-body*

where the *parameter-declaration-clause* is not equivalent to an empty parameter list, is the parameterized constraint (14.10.1.8) with the same parameters as those in the *parameter-declaration-clause* and whose operand is the normal form of conjunction of constraints introduced by *requirements* in the *requirement-body*;

(3.7) — the normal form of a *requires-expression* having one of the following forms

`requires ( void )` *requirement-body*
`requires ()` *requirement-body*
`requires` *requirement-body*

is the conjunction of constraints introduced by *requirements* in the *requirement-body*;

(3.8) — within a *requires-expression*, any predicate constraint P introduced by a *nested-requirement* (5.1.4.4) is replaced by the normal form of the expression of P;

(3.9) — otherwise, the normal form of `E` is the predicate constraint whose expression is `E`.

A declaration's *normalized constraints* are those yielded by normalizing its associated constraints. [*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; };
template<typename T> concept bool C4 = requires (T x) { ++x; }

template<C2 T> void f1(T);                                    // #1
template<C3 T> void f2(T);                                    // #2
template<C4 T> void f3(T);                                    // #3
template<typename T> requires (bool)3 + 4 void f4(T); // error: invalid constraints (#4)
```

The normalized associated constraints of #1 are `sizeof(T) == 1` $\wedge$ `1 == 2`, those of #2 are the type constraint for `T::type`, those of #3 are the parameterized constraint λ(`T x`) the expression constraint `++x`. In #4, the *constraint-expression* `(bool)3 + 4` is not a valid predicate constraint because it does not have type `bool`. — *end example*]

4 A declaration's associated constraints are satisfied by a set of template arguments if and only if its normalized associated constraints are satisfied by those arguments.

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

1 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 P and Q. [*Example:* Subsumption does not determine if the predicate constraint `N >= 0` (14.10.1.2) subsumes `N > 0` for some integral template argument `N`. — *end example*]

2 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]. Parameterized constraints do not appear in conjunctive or disjunctive normal forms. For the purpose of this transformation, the constraint λ(`T x`) P(`x`) is equivalent to the constraint P(`x`). Then, P subsumes Q if and only if

(2.1) — for every disjunctive clause P_i in the disjunctive normal form of P, P_i subsumes every conjunctive clause Q_j in the conjuctive normal form of `Q`, where

(2.2) — a disjunctive clause P_i subsumes a conjunctive clause Q_j if and only if each atomic constraint in P_i subsumes any atomic constraint Q_j, where

(2.3) — an atomic constraint A subsumes another atomic constraint B if and only if the A and B are equivalent using the rules described in 14.10.1 to compare constraints.

[*Example:* Let A and B be atomic constraints (14.10.1.2). The constraint $A \wedge B$ subsumes A, but A does not subsume $A \wedge B$. The constraint A subsumes $A \vee B$, but $A \vee B$ does not subsume A. Also note that every constraint subsumes itself. — *end example*]

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

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

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

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

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

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

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 $A \wedge (B \vee C)$ is in conjunctive normal form. Its conjunctive clauses are A and $(B \vee C)$. The disjunctive normal form of the constraint $A \wedge (B \vee C)$ is $(A \wedge B) \vee (A \wedge C)$. Its disjunctive clauses are $(A \wedge B)$ and $(A \wedge C)$. — *end example*]

4 When two declarations `D1` and `D2` are partially ordered by their normalized constraints (14.10.2), `D1` is *at least as constrained* as `D2` if

(4.1) — `D1` and `D2` are both constrained declarations and `D1`'s normalized constraints subsume those of `D2`; or

(4.2) — `D2` is unconstrained.

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

[*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*]

14.10.4 Resolution of qualified references to concepts [temp.constr.resolve]

1 *Concept resolution* is the process of selecting a concept from a set of concept definitions referred to by a *qualified-concept-name*, or from a set of declarations including one or more concept definitions referred to by a *simple-template-id* or a *qualified-id* whose *unqualified-id* is a *simple-template-id*. Concept resolution is performed when such a name appears

(1.1) — as a *constrained-type-specifier* (7.1.6.4.2),

(1.2) — in a *constrained-parameter* (14.1),

(1.3) — in a *template-introduction* (14.2), or

(1.4) — within a *constraint-expression* (14.10.2).

Within such a name, let `C` be the *concept-name* or *template-name* that refers to the set of concept definitions.

2 The selection of a concept from this set is done by matching the template parameters of each concept in that set to a sequence of template arguments and *wildcard*s. This sequence is called the *concept argument list*, and its elements are called *concept argument*s. For the purpose this matching, a wildcard can match a template parameter of any kind (type, non-type, template) as described below.

3 The method for determining the concept argument list depends on the context in which `C` appears.

(3.1) — If `C` is part of a *constrained-type-specifier* or *constrained-parameter*, then

(3.1.1) — if `C` is a *constrained-type-name*, the concept argument list is comprised of a single wildcard, or

(3.1.2) — if `C` is the *concept-name* of a *partial-concept-id*, the concept argument list is comprised of a single wildcard followed by the *template-arguments* of that *partial-concept-id*.

(3.2) — If `C` is the *concept-name* in a *template-introduction*. the concept argument list is a sequence of wildcards of the same length as the *introduction-list* of the *template-introduction*.

(3.3) — If `C` appears as a *template-name* of a *simple-template-id*, the concept argument list is the sequence of *template-arguments* of that *simple-template-id*.

4 The selection of a concept from the set referred to by `C` is done by matching the concept argument list against the template parameter lists of each concept in that set. For a concept `CC` in that set to be a viable selection, each argument in the concept argument list is matched against the corresponding template parameters of `CC`. Default template arguments of `CC` (if any) are instantiated for each template parameter that does not correspond to a concept argument. Instantiated default arguments are appended to the concept argument list. If the last declared template parameter of `CC` is not a parameter pack and the number of template parameters of `CC` is greater than the number of concept arguments, `CC` is not a viable selection. Otherwise, concept arguments are matched to template parameters using the following rules:

(4.1) — a template argument matches a template parameter if and only if it matches in kind (type, non-type, template) and type according to the rules in 14.4;

(4.2) — a wildcard matches a template parameter of any kind;

(4.3) — a template parameter pack (14.6.3), matches zero or more concept arguments, provided that each of those arguments matches the pattern of the template parameter pack using the rules above for matching matching concept arguments and template parameters.

If any concept arguments do not match a corresponding template parameter, the concept `CC` is not a viable selection. The concept selected by concept resolution shall be the single viable selection in the set of concepts referred by `C`. [*Example:*

```
template<typename T> concept bool C1() { return true; }             // #1
template<typename T, typename U> concept bool C1() { return true; } // #2
template<typename T> concept bool C2() { return true; }
template<int T> concept bool C2() { return true; }
template<typename... Ts> concept bool C3 = true;

void f1(const C1*); // OK: C1 selects #1
void f2(C1<char>);  // OK: C1<char> selects #2

template<C2<0> T> struct S1; // error: no matching concept for C2<0>,
                             // mismatched template arguments
template<C2 T> struct S2;    // error: resolution of C2 is ambiguous,
                             // both concepts are viable

C3{...Ts} void q1(); // OK: selects C3
C3{T} void q2();     // OK: selects C3
```

— end example]

Annex A (informative) Compatibility [diff]

A.1 C++ extensions for Concepts and ISO C++ 2014 [diff.iso]

1 This subclause lists the differences between C++ with Concepts and ISO C++, by the chapters of this document.

A.1.1 Clause 2: lexical conventions [diff.lex]

2.1
Change: New Keywords
New keywords are added to C++ extensions for Concepts; see 2.1.
Rationale: These keywords were added in order to implement the semantics of the new features. In particular, the `requires` keyword is added to introduce constraints through a *requires-clause* or a *requires-expression*. The `concept` keyword is added to enable the definition of concepts (7.1.7), the normalization of constraints (14.10.2), and the semantic differentiation of *concept-name*s from other *identifier*s.
Effect on original feature: Change to semantics of well-defined feature. Any ISO C++ programs that used any of these keywords as identifiers are not valid C++ programs with Concepts.
Difficulty of converting: Syntactic transformation. Converting one specific program is easy. Converting a large collection of related programs takes more work.
How widely used: Seldom.