Document Number: NXXX
Date: 2014-07-29
Revises: N4040
Editor: Andrew Sutton
University of Akron
asutton@uakron.edu

Working Draft, C++ Extensions for Concepts

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

Contents

1 General [intro]

1.1 Scope [general.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 templates based on those constraints. These extensions include new syntactic forms and modifications to existing language semantics.

2 International Standard, ISO/IEC 14882, provides important context and specification for this Technical Specification. This document as 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.

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.

— ISO/IEC 1482:2014, *Programming Languages - C++*

> **Editor's note:** The TS will formally refer to the ISO/IEC document defining the C++14 programming language. Until that document is published, the paper targets the current working draft NXXX

2 ISO/IEC 1482:2014 is herein after called the *C++ Standard*. References to clauses within the C++ Standard are written as "C++ §3.2".

1.3 Terms and definitions [intro.defns]

1 For the purposes of this document, the terms and definitions given in the C++ Standard and the following apply.

1.4 Implementation compliance [intro.compliance]

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

1.5 Acknowledgments [intro.ack]

1 The design of this specification is based, in part, on a concept specification of the algorithms part of the C++ standard library, known as ``The Palo Alto" TR (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 TR and this TS, the TR can be seen as a large-scale test of the expressiveness of this TS.

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

2 Lexical conventions [lex]

2.1 Keywords [lex.key]

In C++ §2.12, Table 4, add the keywords `concept` and `requires`.

5 Expressions [expr]

5.1 Primary expressions [expr.prim]

1 In C++ §5.1.1, add *requires-expression* to the rule, *primary-expression*.

primary-expression:
requires-expression

5.1.2 Lambda expressions [expr.prim.lambda]

Modify C++ §5.1.2/5.

5 The closure type ~~for a non-generic~~ for a *lambda-expression* has a public inline function call operator (C++ §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.~~ For a generic lambda, that function call operator is an abbreviated member function (8.3.5).

Add the following example after those in C++ §5.1.2/5.

5 [*Example:*

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

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

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

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

5.1.3 Requires expressions [expr.req]

1 A *requires-expression* provides a concise way to express syntactic requirements on template arguments.

requires-expression:
`requires` *requirement-parameter-list requirement-body*
requirement-parameter-list:
`(` *parameter-declaration-clause*$_{opt}$ `)`
requirement-body:
`{` *requirement-list* `}`
requirement-list:
requirement
requirement-list requirement
requirement:
simple-requirement
compound-requirement
type-requirement

nested-requirement
simple-requirement:
expression ;
compound-requirement:
`constexpr`*opt* { *expression* } `noexcept`*opt* *trailing-return-type*opt ;
type-requirement:
typename-specifier ;
nested-requirement:
requires-clause ;

2 A *requires-expression* has type `bool`.

3 A *requires-expression* shall not appear outside of a concept definition () or a *requires-clause*.

4 [*Example:* The most common use of *requires-expressions* is to define syntactic requirements in concepts () such as the one below:

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

The concept is defined in terms of the syntactic and type requirements within the *requires-expression*. A *requires-expression* can also be used in a *requires-clause* templates as a way of writing ad hoc constraints on template arguments such as the one below:

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

— *end example*]

5 The *requires-expression* may introduce local arguments via a *parameter-declaration-clause*. These parameters have no linkage, storage, or lifetime. They are used only to write constraints within the *requirement-body* and are not visible outside the closing `}` of the *requirement-body*. The *requirement-parameter-list* shall not include an ellipsis.

6 The *requirement-body* is a sequence of *requirements* separated by semicolons. These *requirements* may refer to local arguments, template parameters, and any other declarations visible from the enclosing context. Each *requirement* introduces a conjunction of one or more atomic constraints (14.9). The kinds of atomic constraints introduced by a *requirement* are:

- A *valid expression constraint* is a predicate on an expression. The constraint is satisfied if and only if the substitution of template arguments into that expression does not result in substitution failure. The result of successfully substituting template arguments into the dependent expression produces a *valid expression*.
- A *valid type constraint* is a predicate on a type. The constraint is satisfied if and only if the substitution of template arguments into that type does not result in substitution failure. The result of successfully substituting template arguments into the dependent type produces an *associated type*.
- A *result type constraint* is a predicate on the result type of a valid expression. Let `E` be a valid expression and `X` be a *trailing-return-type*. The constraint is satisfied if and only if `E` can be used as an argument to an invented function `f`, which has a single function parameter of type `X` and returning `void`. That is, the function call `f(E)` must be a valid expression. [*Note:* Each template parameter referred to by `X` is a template parameter of the invented function `f`. If `X` contains a *constrained-type-specifier* or `auto` specifier, then `f` is a generic function (8.3.5). — *end note*]
- A *constant expression constraint* is satisfied if and only if a valid expression `E` is a constant expression (C++ §5.19).
- An *exception constraint* is satisfied if and only if, for a valid expression `E`, the expression `noexcept(E)` evaluates to `true` (C++ §5.3.7).

7 A *requires-expression* evaluates to `true` if and only the atomic constraints introduced by each *requirement* in the *requirement-list* are satisfied and `false` otherwise. The semantics of each kind of requirement are described in the following sections.

5.1.3.1 Simple requirements [expr.req.simple]

1 A *simple-requirement* introduces a valid expression constraint for its *expression*. The expression is an unevaluated operand (C++ §3.2). [*Example:* The following is requirement evaluates to `true` for all arithmetic types (C++ §3.9.1), and `false` for pointer types (C++ §3.9.2).

```
requires (T a, T b) {
  a + b;  // A simple requirement
}
```

— *end example*]

2 If the expression would always result in a substitution failure, the program is ill-formed. [*Example:*

```
requires () {
  new T[-1];  // error: the valid expression well never be well-formed.
}
```

— *end example*]

5.1.3.2 Type requirements [expr.req.type]

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

```
requires () {
  typename T::inner;         // Required nested type name
  typename Related<T>; // Required alias
}
```

— *end example*]

2 If the required type will always results in a substitution failure, then the program is ill-formed. [*Example:*

```
requires () {
  typename int::X;  // error: int does not have class type
  typename T[-1];   // error: array types cannot have negative extent
}
```

— *end example*]

5.1.3.3 Nested requirements [expr.req.nested]

1 A *nested-requirement* introduces an additional constraint expression 14.9 to be evaluated as part of the satisfaction of the *requires-expression*. The requirement is satisfied if and only if the constraint evaluates to value `true`. [*Example:* Nested requirements are generally used to provide additional constraints on associated types within a *requires-expression*.

```
requires () {
  typename X;
  requires C<X<T>>();
}
```

These requirements are satisfied only when substitution into `X<T>` is successful and when `C<X<T>>()` evaluates to `true`. — *end example*]

5.1.3.4 Compound requirements [expr.req.compound]

1 A *compound-requirement* introduces a conjunction of one or more constraints pertaining to its *expression*, depending on the syntax used. This set includes:

- — a valid expression constraint,
- — an optional associated type constraint
- — an optional result type constraint,
- — an optional constant expression constraint, and
- — an optional an exception constraint.

A *compound-requirement* is satisfied if and only if every constraint in the set is satisfied. The required valid expression is an unevaluated operand (C++ §3.2) except in the case when the `constexpr` specifier is present. These other requirements are described in the following paragraphs.

2 The brace-enclosed *expression* in a *compound-requirement* introduces a valid expression constraint. Let `E` be the valid expression resulting from successful substitution.

3 The presence of a *trailing-return-type* introduces a result type constraint on `E`.

4 If the `constexpr` specifier is present then a constant expression constraint is introduced for the valid expression `E`.

5 If the `noexcept` specifier is present, then an exception constraint is introduced for the valid expression `E`.

6 [*Example:*

```
template<typename I>
  concept bool Inscrutible() { ... }

requires(T x) {
  {x++}; #1
  {*x} -> typename T::r; #2
  {f(x)} -> const Inscrutible& #3
  {g(x)} noexcept -> auto& #4
  constexpr {T::value}; #5
  constexpr {T() + T()} -> T #6;
}
```

Each of these requirements introduces a valid expression constraint on the expression in its enclosing braces. Requirement #1 introduces no additional constraints. It is equivalent to a *simple-requirement* containing the same expression. Requirement #2 `*x` introduces a result type constraint though its *trailing-return-type*, `typename T::r`. The required valid expression `*x` must be usable as an argument to the invented function:

```
template<class T>
  void z1(typename T::r);
```

Requirement #3 also introduces a result type constraint on its required valid expression `f(x)`. This expression must be usable as an argument to the invented generic function:

```
void z2(const Instrutible&)
```

Requirement #4 introduces a result type constraint and an exception constraint. The required valid expression `g(x)` must be usable as as an argument to the invented generic function:

```
void z3(auto&);
```

Additionally, `g(x)` must not propagate exceptions. Requirement #5 introduces a constant expression constraint: `T::value` must be a constant expression. The requirement in #6 introduces a result type constraint and a constant expression constraint. The required valid expression `T() + T()` must be usable as an argument to the invented function:

```
template<class T>
  void z4(T);
```

The valid expression must also be a constant expression. — *end example*]

7 Declarations [dcl.dcl]

7.1 Specifiers [dcl.spec]

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

1 *decl-specifier:*
    `concept`

7.1.6 Type specifiers [dcl.type]

7.1.6.2 Simple type specifiers [dcl.type.simple]

Add *constrained-type-specifier* to the grammar for *simple-type-specifier*s in C++ §7.1.6.2.

1 *simple-type-specifier:*
    constrained-type-specifier
constrained-type-specifier:
    nested-name-specifier$_{opt}$ *constrained-type-name*
constrained-type-name:
    concept-name
    partial-concept-id
concept-name:
    identifier
partial-concept-id:
    concept-name < *template-argument-list* >

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

Modify C++ §7.1.6.4/1 as follows:

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

Add the following paragraph after C++ §7.1.6.4/1.

2 When a *simple-type-specifier* is a *simple-template-id*, its *template-argument-list* may contain multiple occurrences of the the `auto` *type-specifier*. A placeholder type shall not appear in a non-deduced context (C++ §14.8.2.5). [*Example:*

```
namespace N {
  template<typename T> struct Wrap;
  template<typename T> Wrap 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 { };
Size<0> s;

N::Wrap<auto> a = N::make_wrap(0.0);    // OK
Pair<auto, auto> p = make_pair(0, 'a'); // OK
auto::Wrap<int> x = N::make_wrap(0);    // error: auto used in a non-deduced context
Size<sizeof(auto)> y = s;               // error: auto used in a non-deduced context
```

— *end example*]

Modify C++ §7.1.6.4/2 to read:

2 ~~The~~A placeholder type can appear with a function declarator in the *decl-specifier-seq*, *type-specifier-seq*, *conversion-function-id*, or *trailing-return-type*, in any context where such a declarator is valid. The `auto` *type-specifier* can also appear in the *decl-specifier-seq* of a *parameter-declaration* of function declarator. 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 type, the return type of the function is deduced from return statements in the body of the function, if any.

Modify C++ §7.1.6.4/3 as follows:

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

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

— *end example*] Similarly, if the `auto` *type-specifier* appears in the parameter types of a function declarator, that is an *abbreviated function* (8.3.5) [*Example:*

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

— *end example*]

Add the following paragraph after C++ §7.1.6.4/3.

4 The `auto` *type-specifier* can appear in the *trailing-return-type* of a *compound-requirement* in a *requires-expression* (5.1.3.4). [*Example:*

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

— *end example*]

Modify C++ §7.1.6.4/4. The examples in the original text are unchanged and therefore omitted.

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

```
( expression-list )
```

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

Modify C++ §7.1.6.4/7.

5 When a variable declared using a placeholder type is initialized, or a `return` statement occurs in a function declared with a return type that contains a placeholder type, the deduced return type or variable type is determined from the type of its initializer. In the case of a return with no operand, the initializer is considered to be `void()`. Let `T` be the declared type of the variable or return type of the function. ~~If the placeholder is the `auto` *type-specifier*,~~ If `T` contains any occurrences of the `auto` *type-specifier*, the deduced type is determined using the rules for template argument deduction. If the deduction is for a return statement and the initializer is a *braced-init-list* (C++ §8.5.4), the program is ill-formed. Otherwise, obtain `P` from `T` by replacing ~~the occurrences~~ each occurrence of `auto` with ~~either~~ a new invented type template parameter ~~`U`~~ or, if the initializer is a *braced-init-list* and `auto` is a *decl-specifier* in the *decl-specifier-seq* of the variable declaration, replace that single occurrence of `auto`~~,~~ with `std::initializer_list<U>` where `U` is an invented template type parameter. Deduce a value for ~~`U`~~ each invented template type parameter in `P` using the rules of template argument deduction from a function call (C++ §14.8.2.1), where `P` is a function template parameter type and the initializer is the corresponding argument. If the deduction fails, the declaration is ill-formed. Otherwise, the type

deduced for the variable or return type is obtained by substituting the deduced ~~U~~ values for each invented template parameter into P. [*Example:*

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

auto x1 = { 1, 2 };                 // OK: decltype(x1) is std::initializer_list<int>
auto x2 = { 1, 2.0 };               // error: cannot deduce element type
auto& x3 = 12;                      // OK: decltype(x3) is const int&
const auto* p = &x3;                // OK: decltype(p) is const int*
Vec<auto> v1 = make_vec({1, 2, 3}); // OK: decltype(v1) is Vec<int>
Vec<auto> v2 = {1, 2, 3};           // error: cannot deduce element type
```

— *end example*] [*Example:*

```
const auto& i = expr;
```

The type of i is the deduced type of the parameter u in the call f(expr) of the following invented function template:

```
template <class U> void f(const U& u);
```

— *end example*] [*Example:* Similarly, the type of p in the following program

```
template<typename F, typename S> struct Pair;

Pair<const auto&, auto*> p = expr;
```

is the deduced type of the parameter x in the call of g(expr) of the following invented function template:

```
template<typename U1, typename U2> void g(Pair<const U1&, const U2*> x);
```

— *end example*]

7.1.6.5 Constrained type specifiers [dcl.spec.constr]

Add this section to C++ §7.1.6.

1 A *constrained-type-specifier* designates a placeholder type that will be replaced later by deduction from the *expression* in a *compound-requirement* or function argument. This deduction succeeds only when the deduced type satisfies the constraints introduced by the *constrained-type-specifier*. A *constrained-type-specifier* also signifies that a lambda is a generic lambda or that a function is an abbreviated function.

2 When a *simple-type-specifier* is a *simple-template-id*, its *template-argument-list* may contain multiple *constrained-type-specifier*s. A *constrained-type-specifier* shall not appear in a non-deduced context. [*Example:*

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

namespace N {
  template<typename T> struct Wrap;
}
template<typename T, typename U> struct Pair;

void f1(N::Wrap<C>);    // OK
void f2(Pair<C, D<0>>); // OK
void f3(C::Wrap<D<1>>); // error: C used in a non-deduced context
```

— *end example*]

3 A *constrained-type-specifier* can appear with a function declarator in the *decl-specifier-seq* or in the *decl-specifier-seq* of a *parameter-declaration* in that declarator, in any context where such a declarator is valid.

4 If *constrained-type-specifier* appears in the type of a *parameter-declaration* of a *lambda-expression*, the lambda is a generic lambda (5.1.2). Similarly, if a *constrained-type-specifier* appears in the type of a *parameter-declaration* of a function declaration, the function is an abbreviated function (8.3.5). [*Example:*

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

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

— *end example*]

5 A *constrained-type-specifier* can also appear in the *trailing-return-type* of a *compound-requirement* in a *requires-expression* (5.1.3). [*Example:*

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

template<typename T> concept bool Iter() {
  return requires(T i) {
    {*i} -> const C&; // OK
  };
}
```

— *end example*]

6 A program that uses a *constrained-type-specifier* in a context not explicitly allowed by this section is ill-formed.

7 When an *identifier* is a *concept-name* it refers to a set of concept definitions (7.1.7) called the *candidate concept set*. Only type concepts are included in the candidate concept set of *constrained-type-specifier*. [*Note:* The candidate concept set of a *concept-name* in a *constrained-parameter* includes non-type concepts (14.1). — *end note*] If the candidate concept set is empty, the program is ill-formed. [*Note:* The candidate concept set 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<int N> concept bool C() { return true; }                  // #3
template<typename T> concept bool D = true;                        // #4
template<template<typename> class X> concept bool P = true;        // #5

void f(C); // OK: the concept-name C may refer to both #1 and #2
void g(D); // OK: the concept-name D refers only to #4
void h(P); // error: no matching concept definitions for P
```

In the declaration of `f(C)`, the candidate concept set corresponding to the *concept-name*, `C`, does not include #3 because its first *template-parameter* of that concept definition is a non-type template parameter. Likewise, in the declaration of `h(P)`, #5 is not included because its prototype parameter is not a template *type-parameter*. — *end example*]

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

```
template<typename T, typename U> concept bool C() { return true; }
template<typename T, int N = 0> concept bool Seq = true;

void f(C<int>);
void f(Seq<3>);
void f(Seq<>);
```

— *end example*]

9 The concept definition designated by a *constrained-type-specifier* is determined by forming a *template-id* from the *concept-name* and a sequence of *template-arguments*. Let `C` be the *concept-name* in the *constrained-type-specifier*, and let `T` be an invented template *type-parameter* corresponding to the placeholder type that the *type-specifier* designates. When the *constrained-type-name* is a *concept-name*, the *template-id* is formed as `C<T>`. When the *constrained-type-name* is a *partial-concept-id* whose *template-argument-list* is A_1, A_2, ... A_n, the *template-id* is formed as `C<T,` A_1, A_2, ... A_n`>`. If the *template-id* refers to a single concept declaration in the candidate concept set, that concept is the one designated by the

constrained-type-specifier. Otherwise, the program is ill-formed. [*Note:* A *template-id* may not refer to a template specialization if the template arguments do not match the template's declared parameters (14.3). *— 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 P() { return true; }
template<int T> concept bool P() { return true; }

void f1(const C*); // OK: C designates #1
void f2(C<char>);  // OK: C<char> designates #2
void f3(C<3>);     // error: no matching concept for C<3> (mismatched template arguments)
void g1(P);        // error: resolution of P is ambiguous (P refers to two concepts)
```

— end example]

10 The use of a *constrained-type-specifier* in the type of a *parameter-declaration* associates a constraint (14) with the entity for which that parameter is declared. In the case of a generic lambda, the associated constraint appertains to the member function call operator of the closure type (5.1.2). For an abbreviated function declaration, the associated constraint appertains to that function (8.3.5). The use of a *constrained-type-specifier* in the *trailing-return-type* of a *compound-requirement* includes an associated constraint in the conjunction of constraints introduced by that requirement (5.1.3.4).

11 The constraint associated by a *constrained-type-specifier* is formed from the *template-id* used to determine the designated concept. The associated constraint is $\texttt{C<T, A}_1\texttt{, A}_2\texttt{, ... A}_n\texttt{>}$ when the designated concept is a variable concept and $\texttt{C<T, A}_1\texttt{, A}_2\texttt{, ... A}_n\texttt{>()}$ when the designated concept is a function concept. [*Note:* $\texttt{A}_1\texttt{, A}_2\texttt{, ... A}_n$ is empty if the *constrained-type-name* is a *concept-name*. *— end note*] [*Example:*

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

void f(C&);     // associates C<T1> with f
void g(D<int>); // associates D<T2, int> with g
```

In the associated constraints, `T1` and `T2` are invented template type parameters corresponding to the *constrained-type-specifier* in their respective declarations. *— end example*]

12 When multiple *constrained-type-specifier*s appear in the type of a *parameter-declaration* or a *trailing-return-type*, the associated or introduced constraints are a conjunction of the constraints associated by each *constrained-type-specifier*. These constraints are evaluated in the order in which they appear. [*Example:*

```
template<typename T> concept bool C = true;
template<typename T, typename U> concept bool D = true;
tempkate<typename A, typename B> class Pair;

void f(Pair<C, D<int>>& p);
```

The constraint associated by the type of the parameter `p` are:

```
C<T1> && D<T2, int>
```

where `T1` and `T2` are the invented template type parameters corresponding to the *constrained-type-specifier*s `C` and `D<int>`. *— end example*]

7.1.7 `concept` specifier [dcl.spec.concept]

1 The `concept` specifier shall be applied only to the definition of a function template or variable template. A function template definition having the `concept` specifier is called a *function concept*. 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.

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

3 Every concept definition is also a `constexpr` declaration (C++ §7.1.5).

4 A function concept has the following restrictions:

- The template must be unconstrained.
- The return type must be `bool`.
- The declaration shall have a *parameter-declaration-clause* equivalent to `()`.
- The function shall not be recursive.
- The function body shall consist of a single `return` statement whose *expression* shall be a *constraint-expression*.

[*Example:*

```
template<typename T>
  concept bool C1() { return true; }  // OK
template<typename T>
  concept int C2() { return 0; }      // error: return type is not bool
template<typename T>
  concept bool C3(T) { return true; } // error: must have no parameters
```

— end example]

5 A variable concept has the following restrictions:

- The template must be unconstrained.
- The declared type must be `bool`.
- The declaration must have an initializer.
- The initializer shall be a *constraint-expression*.

[*Example:*

```
template<typename T>
  concept bool V1 = true;  // OK
template<typename T>
  concept bool V2 = 3 + 4; // error: initializer is not a constraint-expression
template<Integral T>
  concept bool V3 = true;  // error: constrained template
concept bool V4 = 0;       // error: not a template
```

— end example]

6 A program that declares an explicit instantiation, an explicit specialization, or a partial specialization of a concept definition is ill-formed. [*Example:*

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

template concept bool C<char>; // error: explicit instantiation of a concept
template<>
  concept bool C<int> = true;  // error: explicit specialization of a concept
template<typename T>
  concept bool C<T*> = true;   // error: partial specialization of a concept
```

— end example]

7 [*Note:* The prohibitions against overloading and specialization prevent users from subverting the constraint system by providing a meaning for a concept that differs from the one computed by evaluating its constraints. *— end note*]

8 Declarators [dcl.decl]

Modify C++ §8/1 as follows:

1 A declarator declares a single variable, function, or type within a declaration. The *init-declarator-list* appearing in a declaration is a comma-separated sequence of declarators, each of which may ~~have an initializer~~ have constraints, an initializer, or both.

init-declarator:

declarator requires-clause$_{opt}$ initializer$_{opt}$

Insert the following paragraphs.

2 A *requires-clause* in an *init-declarator* shall only appear with a function declarator (8.3.5). If present, the *requires-clause* associates its *constraint-expression* with the declared function (14). [*Example:*

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

void f1(auto x) requires C<decltype(x)>; // OK
void f2(int x) requires C<int>;          // OK
auto n requires C<decltype(n)> = 'a';    // error: constrained variable declaration
```

— *end example*]

3 The names of parameters in a function declarator are visible in the *constraint-expression* of the *requires-clause*.
[*Example:*

```
void f(int n) requires sizeof(n) == 4; // OK
```

— *end example*]

8.3 Meaning of declarators [dcl.meaning]

8.3.5 Functions [dcl.fct]

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 (13). All declarations for a function shall agree exactly in ~~both~~ the return type, ~~and~~ the parameter-type-list, and asssociated constraints, if any (14).

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 type; otherwise, it is parsed as part of the *parameter-declaration-clause*.

Add the following paragraphs after C++ §8.3.5/15.

16 An abbreviated function (7.1.6.4) is equivalent to a function template (14.5.6) whose *template-parameter-list* includes one invented type *template-parameter* for each occurrence of a placeholder type, designated by either the `auto` *type-specifier* (7.1.6.4) or by a *constrained-type-specifier* (7.1.6.5), in the *parameter-declaration-clause*, in order of appearance. If the placeholder type is designated by a *constrained-type-specifier*, the invented template parameter is a *constrained-parameter* (14.1) declared with same *constrained-type-specifier*. The invented type *template-parameter* is a parameter pack if the corresponding *parameter-declaration* declares a function parameter pack (8.3.5) and the type of the parameter contains only one placeholder type. If the type of the function parameter that declares a function parameter pack contains more than one placeholder type, the program is ill-formed. The adjusted function parameters of an abbreviated function are derived from the *parameter-declaration-clause* by replacing each occurrence of a placeholder type with the name of the corresponding invented type *template-parameter*. [*Example:*

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

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

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

void foo(Pair<auto, auto>...); // error: multiple placeholder types in a parameter pack
```

— *end example*]
[*Example:*

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

void g1(const C1*, C2&);
void g2(Vec<C1>&);
void g3(C1&...);
void g4(Vec<D<int>>);

template<C1 T, C2 U> void g1(const T*, U&); // equivalent to g1(const C1*, C2&)
template<C1 T> void g2(Vec<T>&);            // equivalent to g2(Vec<C1>&)
template<C1... Ts> void g3(Ts&...);         // equivalent to g3(C1&...)
template<D<int> T> void g4(Vec<T>);         // equivalent to g4(Vec<D<int>>)
```

— *end example*]

17 All placeholder types introduced using the same *constrained-type-specifier* have the same invented template type parameter. [*Example:*

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

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

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

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

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

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

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

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

The types of `a`, `b`, and `c` are distinct invented template type parameters even though the constraints associated by the each of the *constrained-type-specifier*s (7.1.6.5) are equivalent.

```
template<typename...> struct Tuple;

void f5(Tuple<C&, C*, N::C> p);
```

The type of `p` is `Tuple<T&, T*, U>` where `T` and `U` are the invented template type parameters corresponding to `C` and `N::C`, respectively. — *end example*]

18 A function template can be an abbreviated function. The invented *template-parameter*s are added to the *template-parameter-list* after the explicitly declared *template-parameter*s. [*Example:*

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

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

— *end example*]

8.4 Function definitions [dcl.fct.def]

8.4.1 In general [dcl.fct.def.general]

Modify the *function-definition* syntax in C++ §8.4.1 to include a *requires-clause*.

1 *function-definition:*
attribute-specifier-seq$_{opt}$ *decl-specifier-seq*$_{opt}$ *declarator virt-specifier-seq*$_{opt}$ *requires-clause*$_{opt}$ *function-body*

Add the following paragraph.

9 If present, the *requires-clause* associates its *constraint-expression* with the function (14).

9 Classes [class]

9.2 Class members [class.mem]

In C++ §9.2, modify the *member-declarator* syntax.

1 *member-declarator:*
declarator virt-specifier-seq$_{opt}$ *pure-specifier-seq*$_{opt}$ *requires-clause*$_{opt}$

Insert the following paragraphs after C++ §9.2/8.

9 A *requires-clause* shall only appear in a *member-declarator* if its *declarator* is a function declarator. The *requires-clause* associates its *constraint-expression* with the function.

10 The names of parameters in a function declarator are visible in the *constraint-expression* of the *requires-clause*.

[*Example:*

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

struct A {
  A(int*) requires C<int>;  // OK
  void f() requires C<int>; // OK
  int x requires C<T>       // error: constrained member viarable
};
```

— end example]

11

9.3 Member functions [class.mfct]

9.3.1 Nonstatic member functions [class.mfct.non-static]

Modify paragraph 6 to read:

6 A non-static member function may be declared *virtual (C++ §10.3) or pure virtual (C++ §10.4)*, *unless the member function has associated constraints (14), in which case the program is ill-formed*.

13 Overloading [over]

Modify paragraph 1.

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) are called *overloaded declarations*. Only function and function template declarations can be overloaded; variable and type declarations cannot be overloaded.

Modify paragraph 3. The itemized list in this paragraph is omitted.

3 [*Note:* As specified in 8.3.5, function declarations that have equivalent parameter declarations and associated constraints, if any (14), declare the same function and therefore cannot be overloaded: ... *— end note*]

13.2 Declaration matching [over.dcl]

Modify paragraph 1. The existing example is omitted from this text.

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

13.3 Overload resolution [over.match]

13.3.1 Viable functions [over.match.viable]

Modify paragraph 1.

1 From the set of candidate functions constructed for a given context (C++ §), 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 (13.3.2). The selection of viable functions considers their associated constraints, if any (14), and their relationships between arguments and function parameters other than the ranking of conversion sequences.

Insert the following paragraph after paragraph 1.

2 First, for a function to be viable, if it has associated constraints (14), those constraints shall be satisfied (14.9).

Modify paragraph 2 as follows:

3 ~~First~~Second, to be a viable function ...

Modify paragraph 3 as follows:

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

13.3.2 Best viable function [over.match.best]

Modify the last item in the list in paragraph 1 and extend it with a new item.

1 Define ICS*i*(`F`) as follows:

— ...

— `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 14.5.6.2~~.~~, or, if not that,

— `F1` and `F2` are member functions of a class template specialization, `T`, and `M1` and `M2` are the member function declarations in the template of `T` that correspond to `F1` and `F2`, respectively, and `M1` is more constrained than `M2` according to the partial ordering of constraints described in 14.9.

14 Templates [temp]

Modify the *template-declaration* grammar in C++ §14/1.

1 *template-declaration:*
  `template <` *template-parameter-list* `>` *requires-clause$_{opt}$* *declaration*
  nested-name-specifier$_{opt}$ *concept-name* `{` *introduction-list* `}` *declaration*
requires-clause:
  `requires` *constraint-expression*
introduction-list:
  introduced-parameter
  introduction-list `,` *introduced-parameter*
introduced-parameter:
  `...`$_{opt}$ *identifier*

Add the following paragraphs after C++ §14/6.

2 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 *concept introduction*, a *concept-name* and following *introduction-list*.

3 The concept designated by the *concept-name* is determined by the *introduction-list*. Let `C` be a *concept-name* and `I1`, `I2`, ..., `In` be a sequence of *identifiers* in the *introduced-parameters* of an *introduction-list*. If the *template-id*, `C<I1, I2, ..., In>`, refers to a single concept declaration, then that concept is the one designated by `C`. Otherwise, the program is ill-formed. [*Example:*

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

Eq{T} void f1(T, T);    // OK: Eq{T} designates #1
Eq{A, B} void f2(A, B); // OK: Eq{A, B} designates #2
```

It is possible to overload function concepts in such a way that a *concept-name* can designate multiple concepts.

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

C{X} void f(); // error: resolution of  C{X} is ambiguous
```

— *end example*]

4 Each *identifier*, `I`, in the *introduced-parameters* of the *introduction-list* is declared to be a template parameter that matches the corresponding template parameter, `P`, in the *template-parameter-list* of the concept designated by the *concept-name*.

- If `P` is a template *type-parameter* declared with either the `class` or `typename` keyword, `I` is declared as a template *type-parameter* using the same keyword;
- if `P` is a template *type-parameter* that declares a class template, `I` is declared as a class template with the template parameters of `P`;
- if `P` is a non-type *template-parameter*, `I` is declared as a non-type *template-parameter* having the same type as `P`;
- if `P` is a template parameter pack, the *identifier*, `I`, shall be preceded by an ellipsis, and is declared as a template parameter pack.

An *introduced-parameter* shall not contain an ellipsis if its corresponding template parameter does not declare a template parameter pack. [*Example:*

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

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

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

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

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

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

— *end example*]

5 6 [*Note:* A concept referred to be a *concept-name* may have template parameters with default template arguments. An *introduction-list* may omit *identifier*s for a corresponding template parameter if it has a default argument. However, only the *introduced-parameter*s are declared as template parameters. [*Example:*

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

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

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

7 The *introduction-list* shall not be empty.

8 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*]

9 [*Note:* A constrained member function template of a constrained class template can be defined outside of its class definition by nested introductions. [*Example:*

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

C{T} struct X {
  D{U} void f();
};

C{T} D{U} void X<T>::f() { } // OK: definition of f()
```

— *end example*] — *end note*]

10 A *template-declaration* declared by a concept introduction can also be an abbreviated function (8.3.5). The invented template parameters introduced by the presence of `auto` *type-specifier*s or *constrained-type-specifier*s in the *parameter-declaration-clause* are added to the list of template parameters introduced by the the *introduction-list*. [*Example:*

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

C{T} void f(T, D);
```

```
template<C T, D __D> void f(T, __D); // OK: redeclaration of f(T, D)
```

— *end example*] [*Example:*

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

C{T} struct X {
  void f(D);
  D{U} void g(U, C);
};

C{T} void X<T>::f(D) { } // OK: definition of X<T>::f(D);
                         // f is a function template with one invented
                         // template type-parameter

C{T} D{U} void X<T>::g(U, C) { } // OK: definition of X<T>::g(U, C);
                                 // g is a function template with two template
                                 // type parameters: one introduced (U) and
                                 // one invented
```

— *end example*]

11 The introduction of a sequence of template parameters, `T1, T2, ..., Tn`, by a *concept-name*, `C`, associates a constraint with the *template-declaration*. That constraint is `C<T1, T2, ..., Tn>` when `C` designates a variable concept and `C<T1, T2, ..., Tn>()` when `C` designates a function concept. If an *introduced-parameter* declares a template parameter pack, its corresponding template argument in the associated constraint is a pack expansion (C++ §14.5.3).
[*Example:*

```
template<typename A, typename B, int C> concept bool C = true;
template<typename A, typename... Args> concept bool D = true;

C{X, Y, Z} struct S;  // associates C<X, Y, Z> with S
D{P, ...Qs} struct T; // associates D<P, Qs...> with T
```

— *end example*]

12 A *template-declaration*'s *associated constraints* are a conjunction of all constraints introduced by

- a concept introduction,
- a *requires-clause* following a *template-parameter-list*,
- any constrained template parameters (14.1) in the declaration's *template-parameter-list*,
- any *constrained-type-specifier*s in the *decl-specifier-seq* of a *parameter-declaration* in a function declaration or definition (7.1.6.5),
- a *requires-clause* appearing after the *declarator* of an *init-declarator* (8), *function-definition* (8.4.1), or *member-declarator* (9.2), or
- some combination these.

A *template-declaration*, `T`, whose constraints are introduced using any combination of these mechanisms is equivalent to another *template-declaration*, `E`, whose template parameters are declared explicitly and as unconstrained template parameters, and `E` has a single *requires-clause* whose *constraint-expression* is equivalent to the associated constraints of `T` (14.5.6.1). [*Note:* This section describes how constrained template declarations can be equivalently written using alternative syntax in order to generate a canonical spelling of a template's associated constraints. [*Example:*

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

// all of the following declarations are equivalent:
void g(C);
template<C T> void g(T);
C{T} void g(T);
template<typename T> requires C<T> void g(T);
```

The last declaration includes the canonical spelling of the associated constraints for all declarations of `g(T)` as the *constraint-expression* of its *requires-clause*. — *end example*] The paragraphs below define the rules that make these declarations equivalent. — *end note*]

13 14 When *template-declaration* is declared by a concept introduction, it is equivalent to a *template-declaration* whose *template-parameter-list* is defined according to the rules for introducing template parameters above, and the equivalent declaration has a *requires-clause* whose *constraint-expression* is equivalent to constraint associated by the concept introduction. [*Example:*

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

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

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

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

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

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

— *end example*]

15 When a *template-declaration*, `T`, is explicitly declared with *template-parameter-list* that has constrained template parameters (14.1), it is equivalent to a *template-declaration*, `E`, with the same template parameters, except that all constrained parameters are replaced by unconstrained parameters matching the corresponding prototype parameter designated by the *constrained-type-specifier* (14.1). The declaration, `E`, has a *requires-clause* whose *constraint-expression* is the conjunction of the constraints associated by the constrained template parameters in `T` (14.1). The order in which the introduced constraints are evaluated is the same as the order in which the constrained template parameters are declared. If the original declaration, `T`, includes a *requires-clause*, its *constraint-expression* is evaluated after the constraints associated by the constrained template parameters in `E`. [*Example:*

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

template<C1 A, C2 B> struct S;
template<C1 A, C2 B> struct R;
template<C1 T> requires C2<sizeof(T)> void f(T);

template<typename X, int Y>
  requires C1<X> && C2<Y>
    struct S; // OK: redeclaration of S

template<typename X, int Y>
  requires C2<Y> && C1<X>
    struct R; // error: redeclaration of R with different constraints
```

```
template<typename T>
  requires C1<T> && C2<sizeof(T)>
    void f(T); // OK: redeclaration of f(T)
```

— *end example*]

16 When the declaration is an abbreviated function, it is equivalent to a *template-declaration* whose template parameters are declared according to the rules in 8.3.5. [*Example:*

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

void f(C, C, D);

template<C T, D U>
  void f(T, T, U);   // OK: redeclaration of f(C, C, D)

template<typename T, typename U>
  requires C<T> && D<U>()
    void f(T, T, U): // OK: also a redeclaration of f(C, C, D)
```

— *end example*]

17 An abbreviated function can also be declared as a *template-declaration*. The constraints associated by *constrained-type-specifier*s in the *parameter-declaration-clause* of the function declaration are evaluated after those introduced by *constrained-type-specifier*s in the *template-parameter-list* and the following *requires-clause*, if present. This is also the case for an abbreviated function that is declared is declared with a concept introduction. [*Example:*

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

template<C T> requires P<T> void g1(T, D);
template<C T> void g2(T, D);

template<typename T, typename U>
  requires C<T> && P<T> && D<U>()
    void g1(T, U);      // OK: redeclaration of g1(T, D)

template<C T, D U>
  requires P<T>     // associated constraints are C<T> && D<U>() && P<T>
    void g1(T, U);  // error: ill-formed, no diagnostic required;

C{T} void g2(T, D); // OK: redeclaration of g2(T, D)
```

The second declaration of `g1(T, U)` is ill-formed (no diagnostic required) because it is functionally equivalent to the first declaration, but not equivalent. — *end example*]

18 A *trailing requires-clause* is a *requires-clause* that appears after the *declarator* in an *init-declarator* (8), *function-definition* (8.4.1), or *member-declarator* (9.2). When a constrained function template or member function template declared with a trailing *requires-clause* is equivalent to a declaration in which the *constraint-expression* of the trailing *requires-clause* is evaluated after all other associated constraints. [*Example:*

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

template<C T> template<typename U>
    requires D<U> && D<T>
      void S<T>::f(U) { } // OK: definition of S<T>::f(U)
```

```
template<C T> template<typename U, typename __P>
    void S<T>::f(U) requires D<U> && D<T> { } // error: redefinition of S<T>::f(U)
```

The second definition if `S<T>::f(U)` is an error because its declaration is equivalent to the first. — *end example*]

14.1 Template parameters [temp.param]

Modify the *template-parameter* grammar in C++ §14.1/1 as follow.

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

Modify C++ §14.1/2 as follows.

2 There is no semantic difference between `class` and `typename` in a *template-parameter*. `typename` followed by an *unqualified-id* names a template type parameter. `typename` followed by a *qualified-id* denotes the type in a non-type ~~*parameter-declaration*~~ *constrained-or-non-type-parameter*.

Modify C++ §14.1/15 as follows.

15 If a *template-parameter* is a *type-parameter* with an ellipsis prior to its optional identifier or is a ~~*parameter-declaration*~~ *constrained-or-non-type-parameter* that declares a parameter pack (8.3.5), then the template-parameter is a template parameter pack (). A template parameter pack that is a ~~*parameter-declaration*~~ *constrained-or-non-type-parameter* whose type contains one or more unexpanded parameter packs is a pack expansion.

Add the following paragraphs after C++ §14.1/15. .

16 A *constrained template parameter* is a *constrained-parameter* whose *decl-specifier-seq* contains a *constrained-type-specifier*. A *constrained-parameter* defines its identifier to be a template parameter that matches in kind the first template parameter, called the *prototype parameter*, of the concept designated by the *constrained-type-specifier*. [*Example:*

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

template<C1 T> void f();       // T is a type parameter
template<C2 X> void g();       // X is a template with one type parameter
template<P N> void x();        // N has type int
template<const P* N> void y(); // N has type const int*
```

— *end example*]

17 If the prototype parameter is a type parameter (including template template parameters), then the *decl-specifier-seq* of the constrained parameter shall consist of only the *constrained-type-specifier*. [*Example:*

```
template<const C1> // Error: declares a const-qualified type parameter
  struct S;
```

— *end example*]

18 The declared *template-parameter* is a template parameter pack if the prototype parameter declares a template parameter pack. In such cases, the *declarator-id* or *abstract-declarator* of the *constrained-parameter* shall also include an ellipsis. [*Example:*

```
template<typename... Ts>
  concept bool X = ...;

template<X... Xs> void f(); // Xs is a parameter pack
template<X Xs> void g();    // Error: must X must include ...
```

— *end example*]

19 If the *constrained-parameter* declares a type parameter, then the *constrained-initializer* is parsed as a *type-id*. Otherwise, it is parsed as a *initializer-clause*. [*Example:*

```
template<C1 T = int> void p(); // Ok
template<P N = 0> void q();    // Ok
template<P M = int> void r();  // Error: int is not an expression
```

— *end example*]

20 The declaration of a *constrained-parameter* introduces a new constraint on the template declaration. The constraint is formed by substituting the declared *template-parameter* as the first template argument of the concept declaration designated by the *constrained-type-specifier* in the *constrained-parameter* declaration. If the *constrained-type-specifier* is a *partial-concept-id*, its template arguments are substituted after the declared *template-parameter*. If the designated concept is a function concept, then the introduced constraint is a function call. [*Example:*

```
template<C1 T> void f1(); // requires C1<T>
template<C2 U> void f2(); // requires C2<U>
template<P N> void f3(); // requires P<N>
```

— *end example*]

21 If the *constrained-parameter* declares a template parameter pack, the formation of the constraint depends on whether the designated concept designated by the parameter's *constrained-type-specifier* is variadic. Let `T` be the declared parameter, `C` be the designated concept, and `Args...` be a sequence of template arguments from a *partial-concept-id*, possibly empty. If `C` is a variadic concept, then the associated constraint is a *template-id* of the form `C<T..., Args...>`. Otherwise, if `C` is not a variadic concept, the associated constraint is a conjunction of sub-constraints `C<Ti, Args...>` for each `Ti` in the parameter pack `T`. If `C` is a function concept, each introduced constraint or sub-constraint is adjusted to be a call expression of the form `C<X, Args...>()` where `X` is either the template parameter pack `T` or an element `Ti`. [*Example:*

```
template<typename... Ts> concept bool P = ...;
template<typename T> concept bool U = ...;


template<P... Xs> void f4();   // requires P<Xs...>
template<U... Args> void f5(); // requires U<Args0> && U<Args1> && ... && U<Argsn>
```

Here, `Args0`, `Args1`, etc. denote elements of the template argument pack `Args` used as part of the introduced constraint. — *end example*]

14.2 Template names [temp.names]

1 Insert the following paragraphs after C++ §14.2/7.

2 If a *template-id* refers to a specialization of a constrained template declaration, the template's associated constraints are checked by substituting the *template-arguments* into the constraints and evaluating the resulting expression. If the substitution results in an invalid type or expression, or if the associated constraints evaluate to `false`, then the program is ill-formed.
[*Example:*

```
template<typename T> concept bool True = true;
template<typename T> concept bool False = false;

template<False T> struct S;
template<True T> using Ptr = T*;
```

```
S<int>* x;   // Error: int does not satisfy the constraints of False.
Ptr<int> z;  // Ok: z has type int*
```

— *end example*] [*Note:* Checking the constraints of a constrained class template does not require its instantiation. This guarantees that a partial specialization cannot be less specialized than a primary template. This requirement is enforced during name lookup, not when the partial specialization is declared. — *end note*]

3

14.3 Template arguments [temp.arg]

14.3.1 Template template arguments [temp.arg.template]

1 Modify C++ §14.3.3.

2 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 the associated constraints of `P` shall subsume the associated constraints of `A` (14.9). [*Example:*

```
template<typename T>
  concept bool X = has_x<T>::value;
template<typename T>
  concept bool Y = X<T> && has_y<T>::value;
template<typename T>
  concept bool Z = Y<T> && has_z<T>::value;

template<template<Y> class C>
  class temp { ... };

template<X T> class x;
template<Z T> class z;

temp<x> s1; // OK: X is subsumed by Y
temp<z> s2; // Error: Z subsumes Y
```

The template `x` is a valid argument for `temp` because any template arguments satisfying `Y` will also satisfy `X`. That is, all uses of `x` by `temp` should result in well-formed programs. The template `y` is not valid because some template arguments satisfying `Y` may not satisfy `Z`. — *end example*]

14.5 Template declarations [temp.decls]

14.5.1 Class templates [temp.class]

Insert the following paragraph after C++ §14.5.1/3.

1 When a member of a constrained class template is defined outside of its class template definition, it shall be specified with the *template-parameters* and associated constraints of the class template.
[*Example:*

```
template<typename T> concept bool Con = ...;

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

template<typename T>
```

```
  requires Con<T>
    void S<T>::f() { } // Ok: parameters and constraints match

template<typename T>
  void S<T>::g() { } // Error: no declaration of g() in S<T>
```

— *end example*]

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

1 Add the following paragraphs after C++ §14.5.1.1.

2 A member function of a class template whose declarator contains a *requires-clause* is a *constrained member function*. [*Example:*

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

— *end example*]

3 Constraints on member functions are instantiated as needed during overload resolution, not when the class template is instantiated (C++ §14.7.1). [*Note:* Constraints on member functions do not affect the declared interface of a class. That is, a constrained copy constructor is still a copy constructor, even if it will not be viable for a specialization of the class template. — *end note*]

4 A constrained member function of a class template may be defined outside of its class template definition. Its definition shall be specified with the constraints of its declaration.
[*Example:* Consider possible definitions of the constrained member function `S<T>f` from above.

```
template<typenamae T>
  void S<T>::f() { }                 // Error: no declaration of f() in S<T>.

template<typenamae T>
  void S<T>::f() requires C<T>() { } // Ok: defines S<T>::f
```

— *end example*]

14.5.2 Member templates [temp.mem]

1 Insert the following paragraph after C++ §14.5.2/1.

2 A constrained member template defined outside of its class template definition shall be specified with the *template-parameter*s and constraints of the class template followed by the template parameters and constraints of the member template.
[*Example:*

```
template<typename T> concept bool Foo = ...;
template<typename T> concept bool Bar = ...; // Different than Foo

template<Foo T>
  struct S {
    template<Bar U> void f(U);
    template<Bar U> void g(U);
  };

template<Foo T> template<Bar U> void S<T>::f(U); // Ok
template<Foo T> template<Foo U> void S<T>::g(U); // Error: no g() declared in S
```

The template constraints in the definition of `g` do not match those in its declaration. — *end example*]

14.5.4 Friends [temp.friend]

1 Add the following paragraphs after C++ §14.5.4/9.

2 A *constrained friend* of a class or class template is a constrained class template, constrained function template, a constrained ordinary or generic (non-member) function definition. [*Example:* When `C` is a type concept, all of the following are valid constrained friend declarations.

```
template<typename T>
  struct X {
    template<C U>
      friend void f(X x, U u) { }    // Constrained function template

    template<C W>
      friend struct Z { };           // Constrained class template

    friend bool operator==(X a, X b) // Constrained ordinary function
      requires C<T>() { return true; }

    friend void g(X a, C b) { }      // Constrained generic function
  };
```

Note that `g` is a generic function because the the parameter `b` has a *constrained-type-specifier*. — *end example*]

3 A non-template friend function shall not be constrained unless the function's parameter or result type depends on a template parameter. [*Example:*

```
template<typename T>
  struct S {
    friend void f(int n) requires C<T>(); // Error: cannot be constrained
  };
```

— *end example*]

4 A constrained non-template friend function shall not declare a specialization. [*Example:*

```
template<typename T>
  struct S {
    friend void f<>(T x) requires C<T>(); // Error: declares a specialization

    friend void g(T x) requires C<T>() { } // OK: does not declare a specialization
  };
```

— *end example*]

5 As with constrained member functions, constraints on non-template friend functions are not instantiated during class template instantiation.

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

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

1 Modify C++ §14.5.5.1/2.

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 (C++ §14.8.2) , and the deduced template arguments satisfy the constraints of the partial specialization, if any (14.9).

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

1 Modify C++ §14.5.5.2/1.

2 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 (C++ §14.5.6.2):

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

[*Example:*

```
template<typename T>
  concept bool Integer = is_integral<T>::value;
template<typename T>
  concept bool Unsigned_integer = Integer<T> && is_unsigned<T>::value;

template<typename T> class S { };
template<Integer T> class S<T> { };          // #1
template<Unsigned_integer T> class S<T> { }; // #2

template<Integer T> void f(S<T>);          // A
template<Unsigned_integer T> void f(S<T>); // B
```

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

14.5.6 Function templates [temp.fct]

14.5.6.1 Function template overloading [temp.over.link]

1 Modify C++ §14.5.6.1/6.

2 A function template can be overloaded either by (non-template) functions of its name or by (other) function templates of the same name. When a call to that name is written (explicitly, or implicitly using the operator notation), template argument deduction 14.8.2, ~~and~~ checking of any explicit template arguments C++ § , and checking of associated constraints 14.9 are performed for each function template to find the template argument values (if any) that can be used with that function template to instantiate a function template specialization that can be invoked with the call arguments. For each function template, if the argument deduction and checking succeeds, the template-arguments (deduced and/or explicit) are used to synthesize the declaration of a single function template specialization which is added to the candidate functions set to be used in overload resolution. If, for a given function template, argument deduction fails, no such function is added to the set of candidate functions for that template. The complete set of candidate functions includes all the synthesized declarations and all of the non-template overloaded functions of the same name. The synthesized declarations are treated like any other functions in the remainder of overload resolution, except as explicitly noted in C++ §.

3 Modify C++ §14.5.6.1

4 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, and constraints 14.9 that are equivalent using the rules described above to compare expressions involving template parameters.

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

1 Modify C++ §14.5.6.2/2.

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 the two templates have identical template parameter lists and equivalent return types and parameter lists, then partial ordering selects the template whose associated constraints subsume but are not equivalent to the associated constraints of the other 14.9. A constrained template is always selected over an unconstrained template.

14.7 Template instantiation and specialization [temp.spec]

14.7.1 Implicit instantiation [temp.inst]

1 Insert the following paragraph after C++ §14.7.1/1.

2 The implicit instantiation of a class template does not cause the instantiation of the associated constraints of constrained member functions.

14.7.2 Explicit instantiation [temp.explicit]

Add the following paragraph:

14 If the explicit instantiation names a constrained function template, member function, or member function template, then the explicit specialization shall have associated constraints that are equivalent to those of the associated constraints of the template declaration (14) after substituting the specified or deduced template arguments into the template's associated constraints. Explicit instantiations of class templates and variable templates cannot be constrained; such declarations simply refer to their respective template declarations as if the templates were unconstrained. The template arguments of an explicit specialization are not required to satisfy the associated constraints of the declaration's template declaration (if any). [*Example:*

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

void f(C) { }

struct Effable { void f(); };

template void f(double);                     // error: no matching declaration
template void f(Effable) requires C<Effable>; // OK
template void f(int) requires C<int>;         // OK
```

— *end example*] [*Example:* The explicit instantiation of a constrained class template does not include a specification of associated constraints.

```
template<C T> struct Vec { };

template struct Vec<int>; // OK

Vec<int> v;              // error: invalid type
```

While the explicit instantiation of `Vec<int>` is valid, the use of that type is not, since `int` does not satisfy `C`. — *end example*] [*Example:*

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

template void S<int>::f();              // OK: explicit specialization of #1
template void S<int>::f() requires C<T>; // OK: explicit specialization of #2
```

In the last declaration, the *requires-clause* is need to determine which declaration is being explicitly instantiated. — *end example*]

14.7.3 Explicit specialization [temp.expl.spec]

1 Insert the following paragraphs under C++ §14.7.3.

2 A constrained template declaration or constrained member function of a class template can be declared by a declaration introduced by `template<>`.

3 The *template arguments* of a *simple-template-id* that names an explicit specialization of a constrained template declaration must satisfy that template's associated constraints (14). [*Example:* `C` is the type concept defined in the previous section.

```
template<C T>
  struct S1 { };

struct X { void c(); }

template<> S1<X> { };   // OK: X satisfies C
template<> S1<int> { }; // Error: int does not satisfy C
```

— *end example*]

4 An explicit specialization of a constrained member function (14.5.1.1) shall not include a a *requires-clause*. [*Example:*

```
template<typename T>
  struct S2 {
    void f(T) requires C<T>;
  };

template<> void S2<X>::f(T a) { }               // OK
template<> void S2<X>::f(T a) requires C<X> { } // Error: extra requires-clause
```

— *end example*]

14.8 Function template specialization [temp.fct.spec]

14.8.2 Template argument deduction [temp.deduct]

After C++ §14.8.2/5, add the following paragraph:

6 If the template has associated constraints, the template arguments are substituted into those associated constraints and evaluated. If the substitution results in an invalid type or expression, or if the associated constraints evaluate to `false`, type deduction fails.

14.9 Template constraints [temp.constr]

1 Add this as a new section under C++ §14.

2 Certain contexts require expressions that satisfy additional requirements as detailed in this sub-clause. Expressions that satisfy these requirements are called *constraint expression*s or simply *constraint*s.

constraint-expression:
logical-or-expression

3 A *logical-or-expression* is a *constraint-expression* if, after substituting template arguments, the resulting expression

- is a constant expression,
- has type `bool`, and
- both operands `P` and `Q` in every subexpression of a constraint of the form `P || Q` or `P && Q` have type `bool`.

[*Note:* A *constraint-expression* defines a subset of constant expressions over which certain logical implications can be proven during translation. The requirement that operands to logical operators have type `bool` prevents constraint expressions from finding user-defined overloads of those operators and possibly subverting the logical processing required by constraints. — *end note*]

4 A program that includes an expression not satisfying these requirements in a context where a *constraint-expression* is required is ill-formed.

5 [*Example:* Let `T` be a dependent type, `C` be a unary function concept, `P`, `Q`, and `R` be value-dependent expressions whose type is `bool`, and `M` and `N` be integral expressions. All of the following expressions can be used as constraints:

```
C<T>()
has_trait<T>::value // only if value is a bool member
P && Q
P || (Q && R)
M == N              // only if the result type is bool
has_trait<T>::value // only if value is a bool member
M < N               // only if the result type is bool
M + N >= 0
P || !(M < N)
true
false
```

An expression of the form `M + N` is not a valid constraint when the arguments have type `int` since the expression's type is not `bool`. Using this expression as a constraint would make the program ill-formed. — *end example*]

6 A subexpression of a *constraint-expression* that calls a function concept or refers to a variable concept is a *concept check*. A concept check is not evaluated; it is simplified according to the rules described in this section.

7 Certain subexpressions of a *constraint-expression* are considered *atomic constraint*s. A constraint is atomic if it is not:

- a *logical-or-expression* of the form `P || Q`,
- a *logical-and-expression* of the form `P && Q`,
- a concept check,
- a *requires-expression*, or
- a subexpression of an atomic constraint.

The valid expression constraints, valid type constraints, result type constraints, and exception constraints introduced by a *requires-clause* are also atomic constraints.
[*Example:*

```
has_trait<T>::value
M < N
M + N >= 0
true
false
```

— *end example*]
[*Note:* A concept check is not an atomic expression. — *end note*]

8 Constraints are *simplified* by reducing them to expressions containing only logical operators and atomic constraints. Concept checks and *requires-expression*s are replaced by simplified expressions. [*Note:* An implementation is not required to normalize the constraint by rewriting in e.g., disjunctive normal form. — *end note*]

9 A concept check that calls a function concept is simplified by substituting the explicit template arguments into the named function body's return expression. A concept check that refers to a variable concepts is simplified by substituting the template arguments into the variable's initializer.

10 A *requires-expression* is simplified by replacing it with the conjunction of constraints introduced by the *requirement*s its *requirement-list*. [*Note:* Certain atomic constraints introduced by a *requirement* have no explicit syntactic representation in the C++. — *end note*]

11 [*Example:* Let `P` and Q be variable templates that are atomic constraints.

```
template<typename T>
  concept bool P_and_Q() { return P<T> && Q<T>; }

template<typename T>
  concept bool P_or_Q = P<T> || Q<T>;

template<typename T>
  concept bool C = P_and_Q<T> &&
                 requires(T x) { x.p() -> int; };

template<typename X>
  requires P_and_Q<X>() void f();

template<typename X>
  requires P_or_Q<X> void g();

template<typename X>
  requires C<X> void h();
```

The associated constraints of `f` are simplified to the expression `P<X> && Q<X>`, and the associated constraints of `g` are simplified to `P<X> || Q<X>`. The associated constraints of `h` are:

```
P<X> && Q<X>
  && /* requires x.p() for all x of type X* /
  && /* requires that x.p() convert to int */
```

— *end example*]

12 A constraint is *satisfied* if, after substituting template arguments, it evaluates to `true`. Otherwise, the constraint is *unsatisfied*.

13 For a mapping *M* from a set *X* of atomic constraints to boolean values, let *G(M)* be the mapping from constraints to boolean values such that *G(M)(C)* is the result of substituting each atomic constraint A within *C* for *M(A)*. For two constraints `P` and `Q`, let *X* be the set of all atomic constraints that appear in `P` and `Q`. `P` is said to *subsume* `Q` if, for every mapping *M* from members of *X* to boolean values for which *M(A) = M(B)* whenever *A* and *B* are equivalent, either *G(M)(P)* is false or *G(M)(Q)* is true (or both).

14 Two *constraint-expressions* `P` and `Q` are *logically equivalent* if and only if `P` subsumes `Q` and `Q` subsumes `P`.

Publish to new window