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

1 **General** ................................................................. 4
   1.1 Scope ....................................................................... 4
   1.2 Normative references .................................................. 4
   1.3 Implementation compliance .......................................... 4
   1.4 Acknowledgments ....................................................... 4

2 **Lexical conventions** ....................................................... 5
   2.1 Keywords ................................................................... 5

5 **Expressions** ................................................................. 6
   5.1 Primary expressions ...................................................... 6
      5.1.2 Lambda expressions ............................................... 6
   5.1.3 Requires expressions ............................................... 6
      5.1.3.1 Simple requirements ........................................... 7
      5.1.3.2 Type requirements ............................................. 8
      5.1.3.3 Compound requirements ...................................... 8
      5.1.3.4 Nested requirements ......................................... 9

7 **Declarations** ............................................................... 10
   7.1 Specifiers .................................................................. 10
      7.1.6 Type specifiers .................................................... 10
         7.1.6.2 Simple type specifiers ....................................... 10
         7.1.6.4 auto specifier .................................................. 10
         7.1.6.5 Constrained type specifiers ................................. 12
      7.1.7 concept specifier ................................................... 15

8 **Declarators** ................................................................. 17
   8.3 Meaning of declarators ................................................. 17
      8.3.5 Functions .............................................................. 17
   8.4 Function definitions .................................................... 19
      8.4.1 In general ............................................................. 19

9 **Classes** ................................................................. 20
   9.2 Class members .......................................................... 20

10 **Derived classes** ......................................................... 21
   10.3 Virtual functions ........................................................ 21

13 **Overloading** .............................................................. 22
   13.2 Declaration matching ................................................ 22
   13.3 Overload resolution .................................................. 22
      13.3.1 Viable functions ................................................ 22
      13.3.2 Best viable function ............................................ 22
   13.4 Address of overloaded function ................................... 23

14 **Templates** ................................................................. 24
   14.1 Template parameters ................................................ 26
   14.2 Introduction of template parameters ............................. 28
   14.3 Names of template specializations ................................. 29
   14.4 Template arguments ................................................ 30
      14.4.1 Template template arguments ................................ 30
   14.5 Template declarations ................................................ 30
      14.5.1 Class templates .................................................... 30
         14.5.1.1 Member functions of class templates ................... 31
      14.5.2 Member templates ............................................... 31
      14.5.4 Friends ............................................................... 32
      14.5.5 Class template partial specialization ........................ 32
         14.5.5.1 Matching of class template partial specializations .... 33
         14.5.5.2 Partial ordering of class template specializations .... 33
      14.5.6 Function templates ................................................. 34
         14.5.6.1 Function template overloading ............................. 34
         14.5.6.2 Partial ordering of function templates .................. 34
1 General

1.1 Scope

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.

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

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 14882:2014, Programming Languages - C++

ISO/IEC 14882:2014 is herein after called the C++ Standard. References to clauses within the C++ Standard are written as "C++ §3.2".

1.3 Implementation compliance

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 ]

1.4 Acknowledgments

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.

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

2.1 Keywords

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

5.1 Primary expressions

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

primary-expression:
  requires-expression

5.1.2 Lambda expressions

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

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

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

6 The closure type for a non-generic 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, the function call operator is an abbreviated member function, whose parameters and return type are derived according to the rules in 8.3.5.

Add the following example after those in C++ §5.1.2/5. Note that the existing examples in the original document are omitted in this document.

[ Example:
  template<typename T> concept bool C = true;

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

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

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

5.1.3 Requires expressions

1 A requires-expression provides a concise way to express syntactic requirements on template arguments. A syntactic requirement is one that can be checked by name lookup (C++ §3.4) or by checking properties of types and expressions. A requires-expression defines a conjunction of constraints (14.9), which are introduced by its requirements.

requires-expression:
  requires requirement-parameter-list requirement-body
  requirement-parameter-list:
    ( parameter-declaration-clauseopt )


requirement-body:
  { requirement-list }

requirement-list:
  requirement ;
  requirement-list requirement

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

2 A requires-expression has type bool.

3 A requires-expression evaluates to true when all of its introduced constraints are satisfied (14.9), and false otherwise.

4 A requires-expression shall appear only within a concept definition (7.1.7) or a requires-clause following a template-parameter-list (14), init-declarator (8), function-definition (8.4), or member-declarator (9.2). [ Example: The most common use of requires-expressions is to define syntactic requirements in concepts such as the one below:

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

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

```cpp
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 ]

5 A requires-expression may introduce local parameters using a parameter-declaration-clause (8.3.5). 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.

6 A local parameter shall not have a default argument.

7 The requirement-parameter-list shall not include an ellipsis.

8 The requirement-body defines a block scope, and is comprised of a sequence of requirements. These requirements may refer to local parameters, template parameters, and any other declarations visible from the enclosing context. Each requirement appends one or more atomic constraints (14.9) to the conjunction of constraints defined by the requires-expressions.

5.1.3.1 Simple requirements

simple-requirement:
  expression

1 A simple-requirement introduces an expression constraint (14.9.2.2) for its expression. [ Example:

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

— end example ]
The substitution of template arguments into the expression of a simple-requirement may result in an ill-formed expression. However, the resulting program is not ill-formed.

If the expression would always result in a substitution failure, the program is ill-formed. [Example:
```cpp
template<typename T> concept bool C =
    requires () {
        new T[-1]; // error: the expression will never be well-formed
    };
```
— end example]

5.1.3.2 Type requirements

A type-requirement introduces type constraint (14.9.2.3) for the type named by its typename-specifier. [Note: A type requirement asserts the validity of an associated type, either as a member type, a class template specialization, or an alias template. It is not used to specify requirements for arbitrary type-specifiers. — end note] [Example:
```cpp
template<typename T> struct S { }

template<typename T> using Ref = T&;

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

The substitution of template arguments into the typename-specifier of a type-requirement may result in an ill-formed type. However, the resulting program is not ill-formed.

If the required type would always result in a substitution failure, then the program is ill-formed. [Example:
```cpp
template<typename T> concept bool C =
    requires () {
        typename int::X; // error: int does not have class type
    };
```
— end example]

5.1.3.3 Compound requirements

A compound-requirement introduces a conjunction of one or more atomic constraints related to its expression. A compound-requirement appends an expression constraint (14.9.2.2) for its expression, \( \epsilon \). The constexpr specifier, if present, appends a constant expression constraint (14.9.2.6) for \( \epsilon \). The noexcept specifier, if present, appends an exception constraint (14.9.2.7) for \( \epsilon \). If the trailing-return-type is given, and its type (call it \( \tau \)), contains no placeholders (7.1.6.4, 7.1.6.5), the requirement appends two constraints to the conjunction of constraints: a type constraint on the formation of \( \tau \) (14.9.2.3) and an implicit conversion constraint from \( \epsilon \) to \( \tau \) (14.9.2.4). Otherwise, if \( \tau \) contains placeholders, an argument deduction constraint (14.9.2.5) of \( \epsilon \) against the type \( \tau \) is appended to the conjunction of constraints. In this constraint, deduction is performed against an invented function template whose declaration is formed according to the rules for abbreviated functions in 8.3.5 where \( \tau \) is the type of the invented function's only parameter.

[Example:
template<typename I> concept bool C() { return true; }

template<typename D> concept bool D = 
requires(T x) {
{x++}; // #1
{*x} -> typename T::inner; // #2
{f(x)} -> const C&; // #3
{g(x)} noexcept; // #4
constexpr {T::value}; // #5
};

Each of these requirements introduces a valid expression constraint on the expression in its enclosing braces. Requirement #1 only an expression constraint on x++. It is equivalent to a simple-requirement with the same expression. Requirement #2 *x 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. Requirement #3 also introduces two constraints: an expression constraint for f(x), and a deduction constraint requiring that overload resolution succeeds for the call i(f(x)) where g is the following invented function template.

template<typename T> requires C<T>() void i(const T &);

Requirement #4 introduces two constraints: an expression constraint for g(x) and an exception constraint on g(x). Requirement #5 also introduces two constraints: An expression constraint on T::value and a constant expression constraint that T::value.

The substitution of template arguments into the expression of a simple-requirement may result in an ill-formed expression. However, the resulting program is not ill-formed.

5.1.3.4 Nested requirements

...
7 Declarations

7.1 Specifiers

Extend the decl-specifier production to include the concept specifier:

```
1
decl-specifier:
    concept
```

7.1.6 Type specifiers

7.1.6.2 Simple type specifiers

Add constrained-type-specifier to the grammar for simple-type-specifiers.

```
1
simple-type-specifier:
    constrained-type-specifier
```

7.1.6.4 auto specifier

Modify paragraph 1 to extend the use of auto to designate abbreviated functions.

```
1
The auto and decltype(auto) type-specifiers 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. [ Note: The use of the auto type-specifier in a non-deduced
context (C++ §14.8.2.5) will cause the deduction of a value for that placeholder type to fail,
resulting in an ill-formed program. — end note ] [ Example: ]

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

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

— end example ]

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

```
2
The placeholder type can appear with a function declarator in the decl-specifier-seq, type-
specifier-seq, conversion-function-id, or trailing-return-type, in any context where such a
declarator is valid. If the function declarator includes a trailing-return-type (8.3.5), 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 paragraph 3 to allow the use of `auto` within the parameter type of a lambda or function.

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

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

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

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

— end example ]

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

4 If the `auto` type-specifier appears in the trailing-return-type of a compound-requirement in a requires-expression (5.1.3.3), that return type introduces an argument deduction constraint (14.9.2.5). [Example:

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

— end example ]

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

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++ §3.6.3), 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` of the `decl-specifier-seq`, and 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

```cpp
(expression-list)
```

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

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

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 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:

- replace each occurrence of `auto` in the variable type with a new invented type template parameter, or
- when the initializer is a braced-init-list and `auto` is a decl-specifier of the decl-specifier-seq of the variable declaration, replace that occurrence of `auto` with std::initializer_list<U>

where `U` is an invented template type parameter.

Deduce a value for `U` each invented template type parameter in `P` using the rules of template argument deduction from a function call (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::initalizer_list<T>) { return Vec<T>{}; }  
  
auto x1 = { 1, 2 };  // OK: decltype(x1) is std::initializer_list<int>  
auto x2 = { 1, 2.0 };  // error: cannot deduce element type  
auto& x3 = *x1.begin();  // OK: decltype(x3) is int&  
const auto* p = &x3;  // OK: decltype(p) is const int*  
Vec:auto> v1 = make_vec({1, 2, 3});  // OK: decltype(v1) is Vec<int>  
Vec:auto> v2 = {1, 2, 3};  // error: cannot deduce element type  
  
— end example ] [ 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;  
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);  
is the deduced type of the parameter x in the call of g(make_pair(fn, &S::mfn)) of the following invented function template:  
  
template<class T1, class T2, class T2, class T3, class T4, class T5, class T6>  
void g(Pair<T1(*)(T2, T3), T4 (T5::*)(T6)>);  

— end example ]

7.1.6.5 Constrained type specifiers
[dcl.spec.constr]

Add this section to C++ §7.1.6.

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

  constrained-type-specifier:  
    nested-name-specifier opt constrained-type-name

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

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

  template<typename T, int N> class Array { };  
  template<typename T, template<typename> class A> class Stack { };  
  template<typename T> class Alloc { };
A constrained-type-specifier can appear in many of the same places as the auto type-specifier, is subject to the same rules, and has equivalent meaning in those contexts. In particular, a constrained-type-specifier can appear in the following contexts with the given meaning:

- a parameter type of a function declaration, signifying an abbreviated function (8.3.5);
- a parameter of a lambda, signifying a generic lambda (5.1.2);
- the parameter type of a template parameter, signifying a constrained template parameter (14.1);
- the trailing-return-type of a compound-requirement (5.1.3.3), signifying an argument deduction constraint (14.9.2.5).

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

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

--- end example]
```

---

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

```cpp
void f1(N::Wrap<C1>); // OK
void f2(Pair<C1, C2<0>>); // OK
void f3(Pred<C3>); // OK
void f4(C1::Wrap<C2<1>>); // OK: but deduction of C1 will always fail

--- end note]
```

---

A concept-name refers to a set of concept definitions (7.1.7) called the candidate concept set. If that 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<typename T> concept bool D = true; // #3

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 #3

— end example }

5 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; }

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

— end example ]

6 The concept designated by a constrained-type-specifier is resolved by determining the viability of each concept in the candidate concept set. For each candidate concept in that set, TT is a template-id formed as follows: let c be the concept-name for the candidate concept set, and let x be a template argument that matches the type and form (14.4) of the prototype parameter (7.1.7) of the candidate concept. The template x is called the deduced concept argument. If the constrained-type-name in the constrained-type-specifier is a concept-name, TT is formed as C<x>. Otherwise, the constrained-type-name is a partial-concept-id whose template-argument-list is A₁, A₂, ..., Aₙ, and TT is formed as C<x₁, A₁, A₂, ..., Aₙ>. The candidate concept is a viable candidate concept when all template-arguments of TT match the template parameters of that candidate (14.4). If, after determining the viability of each concept, there is a single viable candidate concept, that is the concept designated by the constrained-type-specifier. Otherwise, the program is ill-formed. [ 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; } // #3

template<typename T> concept bool C() { return true; } // #1

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

— end example ]

7 The use of a constrained-type-specifier in the type of a parameter-declaration associates a constraint-expression (14.9.4) with the entity for which that parameter is declared. In the case of a generic lambda, the association is with the member function call operator of the closure type (5.1.2). For an abbreviated function declaration, the association is with that function (8.3.5). When a constrained-type-specifier appears in the trailing-return-type of a compound-requirement it associates a constraint-expression with the invented function template of the introduced argument deduction constraint (14.9.2.5). When a constrained-type-specifier is used in a template-parameter, the constrained it associates a constraint-expression with the template-declaration in which the template-parameter is declared.

8 The constraint-expression associated by a constrained-type-specifier is derived from the template-id (called TT above) used to determine the viability of the designated concept (call it b). The constraint is formed by replacing the deduced concept argument x in TT with a template argument, A. That template argument is defined as follows:

— when the constrained-type-specifier appears in the type of a parameter-declaration of a function or lambda, or when the when the constrained-type-specifier appears in the trailing-return-type of a compound-requirement, A is the name of the invented template parameter corresponding to the placeholder in that (possibly invented) function template (8.3.5);
— when the constrained-type-specifier appears in a template-parameter declaration, A is the name of the declared parameter (14.1).
Let $T_{T2}$ be a template-id formed as follows. If the template parameter $A$ declares a template parameter pack, and $D$ is a variadic concept (7.1.7), $T_{T2}$ is formed by replacing the deduced concept argument $X$ in $T_T$ with the pack expansion $A...$. Otherwise $T_{T2}$ is formed by replacing $X$ with $A$. Let $E$ be the id-expression $T_{T2}$ when the $D$ is a variable concept, and the function call $T_{T2}()$ when the $D$ is a function concept. If the template parameter $A$ declares a template parameter pack, and $D$ is not a variadic concept, then the associated constraint is the fold expression $(E && ...)$. Otherwise, the associated constraint is the expression $E$. [Example:

```cpp
template<typename T> concept bool C = true;
template<typename T, typename U> concept bool D() { return true; }
template<int N> concept bool Num = true;
void f(C&);              // associates C<T1> with f
void g(D<int>);          // associates D<T2, int>() with g
void h(X<Num>);          // associates Num<M> with h
template<C T> struct s1; // associates C<T> with s1
```

Here, $T_1$ and $T_2$ are invented type template parameters corresponding to the prototype parameter of their respective designated concepts. Likewise, $M$ is an invented non-type template parameter corresponding to the prototype parameter of $Num$.

Two constrained-type-specifiers are said to be equivalent if their associated constraint-expressions are equivalent according to the rules in 14.9.4.

— 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, declared in namespace scope (C++ §3.3.6). A function template definition having the concept specifier is called a function concept. A function concept is a non-throwing function (C++ §15.4). 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:

```cpp
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>
    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 (C++ §7.1.5).

3 No storage specifiers shall appear in a declaration with the concept specifier. Additionally, a concept definition shall not include the friend or constexpr specifiers.

4 A concept definition shall be unconstrained. [Note: A concept defines a total mapping from its template arguments to the values true and false. — end note ]

5 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.

6 A function concept has the following restrictions:

   — No function-specifiers shall appear in its declaration ().

§ 7.1.7
— The declared return type shall be the non-deduced type 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 (14.9.4).

[Note: Return type deduction requires the instantiation of the function definition, but concept definitions are not instantiated; they are normalized (14.9.4). — end note]

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

— end example ]
```

7 A variable concept has the following restrictions:
— The declared type shall be bool.
— The declaration shall have an initializer.
— The initializer shall be a constraint-expression.

[Example:
```
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 declared as a concept
  concept bool V4 = 0;     // error: not a template

— end example ]
```

8 A program that declares an explicit instantiation (14.7.2), an explicit specialization (14.7.3), or a partial specialization of a concept definition is ill-formed. [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

Modify the definition of the init-declarator production in C++ §8/1 as follows:

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-clauseopt initializeropt

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(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:
  template<typename T> concept bool C = true;
  void f(auto x) requires C<decltype(x)>;
  void g(int n) requires sizeof(n) == 4;
  — end example ]

8.3 Meaning of declarators

8.3.5 Functions

Refactor the grammar for parameter-declarations in paragraph 3 in order to support the definition of template-parameters in Clause 14.

parameter-declaration:
  attribute-specifier-seqopt decl-specifier-seq declarator
  attribute-specifier-seqopt decl-specifier-seq declarator = initializer-clause
  attribute-specifier-seqopt decl-specifier-seq abstract-declaratoropt =
    initializer-clause
  basic-parameter-declaration
  basic-parameter-declaration = initializer
  attribute-specifier-seqopt decl-specifier-seq abstract-declaratoropt

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

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 associated constraints, if any (14).

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

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

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

16 An abbreviated function is a function whose parameter-type-list includes one or more placeholders (7.1.6.4, 7.1.6.5). An abbreviated function is equivalent to a function template (14.5.6) whose template-parameter-list includes one invented template-parameter for each occurrence of a placeholder in the parameter-declaration-clause, in order of appearance. If the placeholder is designated by the auto type-specifier, then the corresponding invented template parameter is a type template-parameter. Otherwise, the placeholder is designated by a constrained-type-specifier, and the corresponding invented parameter matches the type and form of the prototype parameter (7.1.7) of the concept designated by the constrained-type-specifier. The invented template-parameter is a parameter pack if the corresponding parameter-declaration declares a function parameter pack and the type of the parameter contains only one placeholder. If the type of the function parameter that declares a function parameter pack containing more than one placeholder, 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 with the name of the corresponding invented template-parameter. If the replacement of a placeholder with the name of a template parameter results in an invalid parameter declaration, the program is ill-formed. [Example:

```cpp
template<typename T> class Vec { }

template<typename T, typename U> class Pair { }

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

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

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

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

— end example

] [ Example:

```cpp

template<int N> concept bool Num = true;

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

The equivalent and erroneous declaration would have this form:

```cpp

template<int N> void h(N*); // error: invalid type
```
All placeholders introduced constrained-type-specifier that are equivalent according to the definition in 7.1.6.5 have the same invented template parameter. [ Example:

```cpp
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;
```

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

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

```cpp
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`.

```cpp
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.

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

The types of `a`, `b`, and `c` are the same since constrained-type-specifiers `E`, `E<>`, and `E<0>` all associate the constraint-expression, `E<T, 0>`, where `T` is an invented template type parameter. — end example ]

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

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

```cpp
    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.

```
function-definition:
    attribute-specifier-seqopt decl-specifier-seqopt declarator virt-specifier-seqopt requires-clauseopt function-body
```

Add the following paragraph.

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

9.2 Class members

Modify the grammar in 9.2 to allow a requires-clause for member declarations.

1

member-declarator:
   declarator virt-specifier-seqopt requires-clauseopt pure-specifier-seqopt

Insert the following after paragraph 8, explaining where a requires-clause can appear and that its constraint-expression can refer to parameters.

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 member function. [Example:

struct A {
   A(int*) requires true;  // OK: constrained constructor
   ~A() requires true;     // OK: constrained destructor
   void f() requires true; // OK: constrained member function
   int x requires true;    // error: constrained member variable
};

— end example ]

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

10.3 Virtual functions

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), the program is ill-formed. If a constrained member function overrides a virtual function in any base class, the program is ill-formed.

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

struct B {
    virtual void f();
};

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

— end example ]
13 Overloading

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

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, declare the same function and therefore cannot be overloaded: ... — end note]

13.2 Declaration matching

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.

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.

13.3 Overload resolution

13.3.1 Viable functions

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 (C++ §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 (13.3.2). The selection of viable functions considers their associated constraints, if any, and their relationships between arguments and function parameters other than the ranking of conversion sequences.

Insert the following paragraph after paragraph 1; this introduces new a criterion for determining if a candidate is viable. Also, update the beginning of the subsequent paragraphs to account for the insertion.

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

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

4 Second Third, for \( f \) to be a viable function ...

13.3.2 Best viable function

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

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,
— The parameter-type-lists of F1 and F2 are the same and F1 is more constrained than F2 according to the partial ordering of constraints described in 14.9.3.

13.4 Address of overloaded function

The introduction of constraints modifies the rules for determining which function is selected when taking the address of an overloaded function. Insert a new paragraph before paragraph 4.

4 If a single constrained function is selected, and the constraints are not satisfied (14.9), the program is ill-formed.

Modify paragraph 4 (paragraph 5 in this document) to incorporate constraints in the selection of an overloaded function when its address is taken. Also add the following example after that in the original document (not shown here).

5 If more than one function is selected, any function template specializations in the set are eliminated if the set also contains a function that is not a function template specialization. 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 14.9.3. Furthermore, 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 14.5.6.2. After such eliminations, if any, there shall remain exactly one selected function.

[ Example:

```c
void f() { }                // #1
void f() requires true { }; // #2
void g() requires false;

void (*pf)() = &f;         // selects #2
void (*pg)() = &g;         // error: g() cannot be selected

— end example ]
```
14 Templates

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

```
template-declaration:
  template < template-parameter-list > requires-clause? declaration
  template-introduction declaration
requires-clause:
  requires constraint-expression
```

Add the following paragraphs after paragraph 6.

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 associated constraints of a template-declaration are the logical && of all constraint-expressions introduced by:
- a concept introduction, and
- a requires-clause following a template-parameter-list, and
- any constrained template parameters (14.1) in the declaration's template-parameter-list, and
- any constrained-type-specifiers in the decl-specifier-seq of a parameter-declaration in a function declaration or definition (7.1.6.5), and
- a requires-clause appearing after the declarator of an init-declarator (8), function-definition (8.4.1), or member-declarator (9.2).

Let T1 be a template-declaration with associated constraints. T1 is equivalent to another template-declaration (call it T2) whose template parameters are declared explicitly as unconstrained template parameters, and T2 has a single requires-clause whose constraint-expression is equivalent to the associated constraints of T1 (14.9.4). T2 is said to be the canonical declaration of all declarations that are equivalent to it according to the rules below. [ Example:

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

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

The last declaration is the canonical declaration of g(T). — end example ]

When a template-declaration is declared by a template introduction (14.2), its canonical declaration is a template-declaration whose template-parameter-list is defined according to the rules for introducing template parameters in 14.2, and the equivalent declaration has a requires-clause whose constraint-expression is formed as follows. Let T = a template-id formed as C<II, I2, ..., In, D1, D2, ..., Dn> where C is the name of the designated concept, I1, I2, ..., In is the sequence of introduced template parameters, and D1, D2, ..., Dn is the (possibly empty) sequence of instantiated default template arguments needed to form the template-id that refers to C. If an introduced parameter declares a template parameter pack, the corresponding template argument in the T = is a pack expansion (C++ §14.5.3). If C is a variable concept, then the constraint-expression is the id-expression T. If C is a function concept, then the constraint-expression is the function call T(). [ Example:

```c
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

9 When a template-declaration (call it T1) is explicitly declared with template-parameter-list that has constrained template parameters (14.1), its canonical declaration is a template-declaration (call it T2) 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 (7.1.6.5). The declaration, T2, has a requires-clause whose constraint-expression is the conjunction of the constraint-expressions associated by the constrained template parameters in T1. 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 T1 includes a requires-clause, its constraint-expression is evaluated after the constraints associated by the constrained template parameters in T2. [ Example:

template<typename> concept bool C1 = true;
template<int> concept bool C2 = true;
template<C1 A, C2 B> struct S;
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 T>
requires C1<T> && C2<sizeof(T)>
void f(T); // OK: redeclaration of f(T)

— end example ]

10 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. The associated constraints of the abbreviated function are evaluated in the order in which they appear. [ Example:

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

void f(C, C, D);

void f<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: redeclaration of g1(T, D)

— end example ]

11 An abbreviated function can also be declared as a template-declaration. The constraints associated by constrained-type-specifiers in the parameter-declaration-clause of the function declaration are evaluated after those introduced by constrained-type-specifiers 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:

```cpp
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)
C{T} void g2(T, D); // OK: redeclaration of g2(T, D)
— end example ]

12 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 is declared with a trailing requires-clause it is equivalent to a declaration in which the constraint-expression of the trailing requires-clause is evaluated after all other associated constraints. [ Example:

```cpp
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>
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

Modify the template-parameter grammar in C++ §14.1/1 in order to allow constrained template parameters.

1

```cpp
template-parameter:
    parameter-declaration
    non-type-or-constrained-parameter

non-type-or-constrained-parameter:
    basic-parameter-declaration
    basic-parameter-declaration = initializer
    basic-parameter-declaration = type-id
    basic-parameter-declaration = id-expression
```

Update the wording in C++ §14.1/2 as follows.

© ISO/IEC N4205 § 14.1 26
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.

Insert the following paragraphs after paragraph 3 in order to distinguish between a template parameter that declares a non-type parameter and a template-parameter that declares a constrained parameter, which may declare a type parameter.

When a non-type-or-constrained-parameter has the following form:

```plaintext
constrained-type-specifier ...opt identifier opt
```

it declares a constrained template parameter. Otherwise the parameter is a non-type template-parameter.

If the `auto` type-specifier appears in the the parameter type of a non-type-or-constrained-parameter, the program is ill-formed. The program is also ill-formed if a constrained-type-specifier appears anywhere in the basic-parameter-declaration and the form of that declaration does not match the form above.

```plaintext
Example:

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

```plaintext
template<C T> struct S1;     // OK: T is a constrained template parameter
template<int N> struct S2;  // OK: N is a non-type template parameter
template<auto X> struct S2; // error: auto in template parameter
template<const D N> void f1(); // error: D is used with a const-qualifier
template<D* N> void f2();     // error: N declares a pointer-to-D
```text

— end example ]

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

A constrained template parameter declares a template parameter whose type and form match that of the prototype parameter of the concept designated by its constrained-type-specifier. The designated concept is found using the rules in 7.1.6.5. Let I be the declarator-id of the basic-parameter-declaration, and let P be the prototype parameter of the designated concept. The declared template parameter is determined by the type and form of P and the identifier and optional ellipsis in I.

- If P is a type `template-parameter` declared with the `class` or `typename`, the declared parameter is a type `template-parameter`.
- If P is a non-type `template-parameter`, the declared parameter is a non-type `template-parameter` having the same type as P.
- If P is a template `template-parameter`, the declared parameter is a template `template-parameter` having the same `template-parameter-list` as P.
- If P declares a template parameter pack, I shall include an ellipsis, and the declared parameter is a template parameter pack.

```plaintext
Example:

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

```plaintext
template<C T> struct S1;     // OK: T is a constrained template parameter
template<int N> struct S2;  // OK: N is a non-type template parameter
template<auto X> struct S2; // error: auto in template parameter
template<const D N> void f1(); // error: D is used with a const-qualifier
template<D* N> void f2();     // error: N declares a pointer-to-D
```text

— end example ]

© ISO/IEC N4205

§ 14.1
14.2 Introduction of template parameters

A template introduction provides a convenient way of declaring different templates that have the same template parameters and constraints.

```
template-introduction:
  nested-name-specifier opt concept-name introduction-list

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

introduced-parameter:
  ...
  opt identifier
```

A template introduction declares a sequence of template-parameters, which are derived from a concept-name and the sequence of identifiers in its introduction-list.

The concept designated by the concept-name (call it C) is determined by the introduction-list. The concept-name C refers to a set of concept definitions. A concept CC in that set is viable if CC declares at least as many template parameters as there are identifiers in the introduction-list, and all template parameters in excess of the number of identifiers are declared with default template arguments. If only one concept in that set is viable, that is the concept designated by C. Otherwise, the program is ill-formed. [ Example: It is possible to overload function concepts in such a way that a concept-name can designate multiple concepts.]

```
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
```

For each introduced-parameter I in an introduction-list, and for its corresponding template parameter in the template-parameter-list of the designated concept (call it P), declare a new template parameter using the rules for declaring a constrained parameter in 14.1 by using I as a declarator-id and P as the prototype parameter. However, if I contains an ellipsis, P shall declare a template parameter pack. [ Example:]

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

C1(A, B, ...C) // OK: A is declared as typename A
  struct s;   // B is declared as int B
  // C is declared as typename... C

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

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

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

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

— end example ]

[ temp.intro ]
template<typename A, typename B = bool>
concept bool C() { return true; }
C(T) void f(T); // OK: f(T) is a function template with
// a single template type parameter T

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

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

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

Array<double, 0> s1; // OK
Array<double> s2;    // error: Array takes two template arguments
— end example ]

6 [ Note: The introduction of a sequence of template parameters by a concept-name associates a constraint with the template-declaration according to the rules describe in 14. — end note ]

14.3 Names of template specializations

Add the following paragraph to require the implicit instantiation of default template arguments for non-function templates.

8 When a simple-template-id does not name a function, a default template-argument is implicitly instantiated (14.7.1) in a context that requires the value of that default argument. [ Example:

template<typename T, typename U = int> struct S { };

S<bool>* p; // the type of p is S<bool, int>*
The default argument for U is instantiated to form the type S<bool, int>*, but the definition of that class template specialization is not implicitly instantiated (14.7.1); it is not used in a context that requires a complete type. — end example ]

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

9 When a simple-template-id names a constrained class template, variable template or alias template, and all template-arguments in the template-id are non-dependent, (C++ §14.6.2), the associated constraints are checked against those template-arguments (14.9). If, as a result of checking, the associated constraints are not satisfied, the program is ill-formed. [ Example:

template<typename C> = false;

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

S<int>* p; // error: constraints not satisfied
Ptr<int> p; // error: constraints not satisfied
— end example ]
14.4 Template arguments

14.4.1 Template template arguments

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

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 more constrained than A according to the rules in 14.9. Two template parameters match if they are of the same kind (type, non-type, template), for non-type template parameters, their types are equivalent (14.5.6.1), and for template template parameters, each of their corresponding template-parameters matches, recursively. When P's template-parameter-list contains a template parameter pack (C++ §14.5.3), the template parameter pack will match zero or more template parameters or template parameter packs in the template-parameter-list of A with the same type and form as the template parameter pack in P (ignoring whether those template parameters are template parameter packs).

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

[Example:

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

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

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

template<typename T> struct X { };  

template<typename T> struct Y { };  

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

14.5 Template declarations

14.5.1 Class templates

Modify paragraph 3 to require template constraints for out-of-class definitions of members of constrained templates. Note that the example in the original document is omitted. The example in this paragraph is to be added after the omitted example.

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

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

§ 14.5.1
template<C T> struct S {
    void f();
    void g();
    template<D U> struct Inner;
};

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

The declaration of S<T>::g() does not match because it does not have the associated constraints of S. — end example ]

14.5.1.1 Member functions of class templates

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 ]

14.5.2 Member templates

Modify paragraph 1 in order to account for constrained member templates of (possibly) constrained class templates. Add the example in this document after the example in the original document, which is omitted here.

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 of the class template followed by the template-parameters and associated constraints of the member template. [ 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<C2 U>
        void S<T>::f(U) { } // OK
    template<C1 T> template<typename U>
        void S<T>::g(U) { } // error: no matching function in S<T>

The associated constraints in the definition of g() do not match those in of its declaration. — end example ]
14.5.4 Friends

Add the following paragraphs to explain the meaning of constrained friend declarations.

A friend function template may be constrained, except:

— if the friend declares, but does not define, a non-template function to be a friend of a class template, the associated constraints of that friend function declaration shall be non-dependent (C++ §14.6.2);

— if the friend declares a function template specialization to be a friend of a class template, the declaration shall not have associated constraints, and the template arguments of that specialization shall satisfy the associated constraints of the template referred to by the friend declaration (14.8.2.6).

Example:

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

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

template<typename T>
struct S {
    void f1() requires true; // OK
    void f2() requires C1<T>; // error: constraints refer to T
    void g1<T>(T);           // OK
    void g2<T>(T);           // error: constraints not satisfied
};
```

— end example ]

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

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

— end example ] In the instantiation of such a class template, the template arguments are substituted into the constraints but not evaluated. Constraints are checked (14.9) only when that function is considered as a viable candidate for overload resolution (13.3.1). — end note ]

14.5.5 Class template partial specialization

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

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

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

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

— end example ]

Modify the 3rd item in the list of paragraph 8 to allow constrained class template partial specializations like #6. Note that all other items in that list are elided.

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

— ...
In an unconstrained class template partial specialization, the argument list of the specialization shall not be identical to the implicit argument list of the primary template.

14.5.5.1 Matching of class template partial specializations

Modify paragraph 2; constraints must be satisfied in order to match a partial specialization. Add the example given here to the (omitted) example in the original document.

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). [Example:

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

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

--- end example]
```

14.5.5.2 Partial ordering of class template specializations

Modify paragraph 1 so that constraints are considered in the partial ordering of class template specializations. Add the example at the end of this paragraph to the (omitted) example in the original document.

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

[Example:

```cpp
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<typename T> class S<T> { }; // #1
template<typename T> class S<T> { }; // #2

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

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

§ 14.5.5.2
14.5.6 Function templates

14.5.6.1 Function template overloading

Modify paragraph 6 to account for constraints on function templates.

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 are:

— declared in the same scope,
— have the same name,
— have identical template parameter lists,
— have return types and parameter lists that are equivalent using the rules described above to compare expressions involving template parameters, and
— have associated constraints that are equivalent using the rules in 14.9.4 to compare constraint-expressions.

Two function templates are functionally equivalent if they are equivalent except that one or more expressions that involve template parameters in the return types and parameter lists are functionally equivalent using the rules described above to compare expressions involving template parameters

— one or more expressions that involve template parameters in the return types and parameter lists are functionally equivalent using the rules described above to compare expressions involving template parameters, or if not that,
— both function templates have associated constraints that are functionally equivalent but not equivalent, using the rules in 14.9.4 to compare constraint-expressions.

If a program contains declarations of function templates that are functionally equivalent but not equivalent, the program is ill-formed; no diagnostic is required.

14.5.6.2 Partial ordering of function templates

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

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 14.9.3.

14.7 Template instantiation and specialization

14.7.1 Implicit instantiation

Add the following paragraph after paragraph 1 in order to explain the how constrained members are instantiated.

When a constrained member of a class is instantiated, new constraints for the instantiated declaration are formed by substituting the template arguments into the associated constraints of that member. The resulting expression is not evaluated after this substitution. If the substitution fails, the program is ill-formed. [ Note: The evaluation of constraints happens during lookup or overload resolution (13). Preserving the spelling of the substituted constraint also allows constrained member function to be partially ordered by those constraints according to the rules in 14.9. — end note ] [ Example:
template<typename T> concept bool C = sizeof(T) > 2;
template<typename T> concept bool D = C && 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

The instantiation of S<char> produces a class template specialization having the constructors, S<char>::S() requires C<char> and S<char>::S() requires D<char>. Even though neither constructor will be selected by overload resolution, they remain a part of the class template specialization. This also has the effect of suppressing the implicit generation of a default constructor (C++ §12.1).

template<typename T> struct S2 {
    void f() requires T::value == 1;
};

S2<int> s; // error: substitution failure in definition of S2<int>

— end example ]

14.7.2 Explicit instantiation

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

14 If the explicit instantiation names a class template specialization or variable template specialization of a constrained template, then the template-arguments in the template-id of the explicit instantiation shall satisfy the template’s associated constraints (14.9). [ Example:

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

template<C T> struct S { }

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

— end example ]

15 When an explicit instantiation refers to a specialization of a function template (14.8.2.6), that template’s associated constraints shall be satisfied by the template arguments of the explicit instantiation. [ Example:

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

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

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

— end example ]

§ 14.7.2
14.7.3 Explicit specialization

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

12 The template-arguments in the template-id of an explicit specialization of a constrained class template or constrained variable template shall satisfy the associated constraints of that template, if any (14.9). [ Example:

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

Example: `template<> S<char> { }`; // OK  
Example: `template<> S<char[L2]> { }`; // error: constraints not satisfied

— end example ]

13 When determining the function template referred to by an explicit specialization of a function template (14.8.2.6), the associated constraints of that template (if any) shall be satisfied (14.9) by the template arguments of the explicit specialization. [ Example:

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

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

— end example ]

14.8 Function template specializations

14.8.2 Template argument deduction

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), the template arguments are substituted into the constraints without evaluating the resulting expression. If this substitution results in an invalid type, then the resulting expression is defined to be false. The resulting constraints are associated with the function template specialization. [ Note: The constraints associated with the function template specialization are checked (14.9) in order to determine if the specialization is a viable candidate (13.3.1), and not at the point of substitution. — end note ]

14.8.2.6 Deducing template arguments from a function declaration

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

2 Remove, from the set of function templates considered, all those whose associated constraints (if any) are not satisfied by the deduced template arguments (14.9).

Update paragraph 2 (now paragraph 3) to accommodate the new wording.
Add this section after 14.8.

There are two logical operators on constraints: conjunction and disjunction.

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

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

Any constraint that is not a conjunction or disjunction is an atomic constraint.
14.9.2.1 Predicate constraints

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

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

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

A predicate constraint `P` subsumes another predicate constraint `Q` if and only if `P` and `Q` are equivalent *constraint-expressions* (14.9.4). [Example: The predicate `M >= 0` does not subsume the predicate `M > 0` because they are not equivalent *constraint-expressions*. — end example]

14.9.2.2 Expression constraints

An *expression constraint* is an atomic constraint that specifies a requirement on the formation of an expression (call it `E`) through substitution of template arguments. An expression constraint is satisfied if `E` is non-dependent, meaning that the substitution yielding `E` did not fail. Within an expression constraint, `E` is an unevaluated operand (5). [Note: An expression constraint is introduced by the expression in either a simple-requirement (5.1.3.1) or compound-requirement (5.1.3.3) of a requires-expression. — end note] [Example: The concept `C` introduces an expression constraint for the expression `++t`.

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

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

An expression constraint `P` subsumes another expression constraint `Q` if and only if the expressions of `P` and `Q` are equivalent (14.5.6.1).

14.9.2.3 Type constraints

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

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

The type `int` does not satisfy this constraint because substitution of that type into the constraint results in a substitution failure; `int::type` is ill-formed. — end example]

A type requirement that names a class template specialization does not require that type to be complete (C++ §3.9).

A type requirement that names a class template specialization does not require that type to be complete (C++ §3.9).

14.9.2.4 Implicit conversion constraints

A *implicit conversion constraint* is an atomic constraint that specifies a requirement on the implicit conversion of an expression (call it `E`) to a type (call it `T`). The constraint is satisfied if and only if `E` is implicitly convertible to `T` (C++ §4). [Note: A conversion constraint is introduced by a trailing-return-type in a compound-requirement when its type contains no placeholders (5.1.3.3). — end note] [Example:
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 

An implicit conversion constraint \( P \) subsumes another implicit conversion constraint \( Q \) if and only if the expressions of \( P \) and \( Q \) are equivalent (14.5.6.1) and the types of \( P \) and \( Q \) are equivalent (C++ §14.4).

14.9.2.5 Argument deduction constraints

1 An argument deduction constraint is an atomic constraint that specifies a requirement on the usability of an expression (call it \( E \)) as an argument to an invented function template (call it \( F \)) that has a single parameter whose type (call it \( T \)) is written in terms of \( F \)'s template parameters. The constraint is satisfied if and only if \( E \) is selected by overload resolution for the call \( F(E) \) (13.3). [ Note: Overload resolution selects \( F \) only when template argument deduction succeeds and \( F \)'s associated constraints are satisfied. — end note ] [ Note: An argument deduction constraint is introduced by a trailing-return-type in a compound-requirement when the trailing-type-specifier-seq contains at least one placeholder (5.1.3.3). — end note ]

Example: The concept \( C \) introduces an argument deduction constraint for the expression \( *t \).

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

When determining if \( C \) is satisfied, overload resolution is performed for the call \( g(*t) \) against the following invented function:

```cpp
template<typename T> requires C<T> void g(const T&);
```

— end example 

2 An argument deduction constraint \( P \) subsumes another argument deduction constraint \( Q \) if and only if the expressions of \( P \) and \( Q \) are equivalent (14.5.6.1), and the types of \( P \) and \( Q \) are equivalent (C++ §14.4).

14.9.2.6 Constant expression constraints

1 A constant expression constraint is an atomic constraint that specifies a requirement that an expression (call it \( E \)) can be evaluated during translation. The constraint is satisfied if and only if \( E \) is a prvalue constant expression (C++ §5.19). [ Note: A constant expression constraint determines if an expression can be evaluated during translation for specific constexpr arguments. It cannot be used to determine if an expression is evaluated during translation for all possible arguments. — end note ] [ Note: Constant expression constraints are introduced by a compound-requirement that includes the constexpr specifier (5.1.3.3). — end note ] Example:

```cpp
template<typename T, int N> concept bool C =
requires() {
    constexpr { T(N) }; // determines if T(N) is a constant expression
};
```

— end example 

2 A constant expression constraint \( P \) subsumes another constant expression constraint \( Q \) if and only if the expressions of \( P \) and \( Q \) are equivalent (14.5.6.1).

14.9.2.7 Exception constraints

1 An exception constraint is an atomic constraint that specifies a requirement that an expression (call it \( E \)) does not throw an exception. It is satisfied if and only if the expression noexcept\((E)\) is true (C++ §5.3.7). [ Note: Constant expression constraints are introduced by a compound-requirement that includes the noexcept specifier (5.1.3.3). — end note ]
An exception constraint \( P \) subsumes another exception constraint \( Q \) if and only if the expressions of \( P \) and \( Q \) are equivalent (14.5.6.1).

### 14.9.3 Partial ordering by constraints

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

- the best viable candidate of non-template functions (13.3.2),
- the address of a non-template function (13.4),
- the matching of template template arguments (14.4.1),
- the partial ordering of class template specializations (14.5.5.2),
- the partial ordering of function templates (14.5.6.2), and

When two declarations \( D_1 \) and \( D_2 \) are partially ordered by their constraints, \( D_1 \) is more constrained than \( D_2 \) if:

- \( D_1 \) and \( D_2 \) are both constrained declarations and \( D_1 \)'s associated constraints subsume but are not subsumed by those of \( D_2 \), or if not that,
- \( D_1 \) is constrained and \( D_2 \) is unconstrained.

**Example:**

```cpp
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
```

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**

### 14.9.4 Constraint expressions

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

**constraint-expression:**

- **logical-or-expression**

A constraint-expression \( \epsilon \) is normalized by forming a constraint (14.9) from \( \epsilon \) and its subexpressions. That transformation is defined as follows:

- The normalization of an expression of the form \( P \) is the normalization of \( P \).
- The normalization of an expression of the form \( P \mid\mid Q \) is the disjunction of the normalization of \( P \) and the normalization of \( Q \). If, in the expression \( P \mid\mid Q \), lookup of \texttt{operator||} finds a user-defined function, the program is ill-formed.
- The normalization of an expression of the form \( P \&\& Q \) is the conjunction of the normalization of \( P \) and the normalization of \( Q \). If, in the expression \( P \&\& Q \), lookup of \texttt{operator&&} finds a user-defined function, the program is ill-formed.
- The normalization of a function call of the form \( \text{C}<A_1, A_2, \ldots, A_n>() \) where \( A_1, A_2, \ldots, A_n \) is a sequence of template arguments and \( \text{C} \) names a function concept (7.1.7) is defined as follows: Let \( \epsilon \) be the expression returned by the function concept \( \text{C} \), and let \( S \) be the result of substituting the template arguments into that expression. The resulting constraint is the normalization of \( S \).
- The normalization of an id-expression of the form \( \text{C}<A_1, A_2, \ldots, A_n> \) where \( A_1, A_2, \ldots, A_n \) is a sequence of template arguments and \( \text{C} \) names a variable concept (7.1.7) is defined as follows: Let \( \epsilon \) be the initializer of the variable concept \( \text{C} \), and let \( S \) be the result of substituting the template arguments into that expression. The resulting constraint is the normalization of \( S \).
The normalization of a requires-expression is defined to be the conjunction of constraints introduced by the body of that expression (§14.9.3).

Otherwise, \( E \) shall be a prvalue constant expression of type bool, and it denotes a predicate constraint (§14.9.2.1).

NOTE: A constraint-expression defines a subset of constant expressions over which certain logical implications can be deduced during translation. The prohibition against user-defined logical operators is intended to prevent the subversion of the underlying logic necessary for partially order constraints (§14.9.3).

EXAMPLE:

```cpp
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; };
```

In the normalized constraints, the expressions sizeof(char) == 1, 1 == 2, and (bool)(3 + 4) are predicate constraints (§14.9.2.1). The concept C4 is normalized to a single type constraint (§14.9.2.3) for the ill-formed type int::type. The expression 3 + 4 is not a constraint-expression because it does not satisfy the requirements for being normalized into a predicate constraint. — end example

Two constraint-expressions are considered equivalent if two function definitions containing the expressions would satisfy the one definition rule (C++ §3.2), except that the tokens used to name template parameters may differ as long as a token used to name a template parameter in one expression is replaced by another token that names the same template parameter in the other expression, and the tokens used to name names of other entities may differ only if those names refer to the same set of declarations. [Example:]

```cpp
template<typename T> concept bool C = sizeof(T) == 1;
template<typename T> requires C<T> void f(); // #1
template<typename U> requires C<U> void f(); // OK: redeclaration of #1

namespace N1 { template<typename T> concept bool C1 = true; }
namespace N2 { template<typename T> concept bool C2 = true; }

template<typename T> requires N1::C1<T> void g(); // #2
template<typename T> requires N1::C1<T> void h(); // #3
using N1::C1;
template<typename T> requires C1<T> void g(); // OK: redeclaration of #2
using namespace N2;
template<typename T> requires C2<T> void h(); // OK: redeclaration of #3
```

— end example ] Two constraint-expressions that are not equivalent are functionally equivalent if, for any given set of template arguments, the satisfaction of their normalized constraints yields the same result. A constraint-expression or subexpression thereof of the form \((E)\) is not equivalent to \(E\), but the two expressions are functionally equivalent.

[ Example:

```cpp
void f() requires true || 1 == 2; // #1
void f() requires true || (1 == 2); // not equivalent but functionally equivalent
```

— end example ]