Wording Paper, C++ extensions for Concepts
# Contents

**List of Tables**

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
</tr>
</tbody>
</table>

**Contents**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Terms and definitions</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5 Lexical conventions</td>
<td>2</td>
</tr>
<tr>
<td>6 Basic concepts</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8 Expressions</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11 Declarators</td>
<td>9</td>
</tr>
<tr>
<td>12 Classes</td>
<td>11</td>
</tr>
<tr>
<td>13 Derived classes</td>
<td>12</td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>16 Overloading</td>
<td>13</td>
</tr>
<tr>
<td>17 Templates</td>
<td>16</td>
</tr>
<tr>
<td>18</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
</tr>
<tr>
<td>A Compatibility</td>
<td>33</td>
</tr>
<tr>
<td>A.1</td>
<td></td>
</tr>
</tbody>
</table>

© ISO/IEC P0734R0
List of Tables
1 Terms and definitions

Modify the definitions of “signature” to include constraints. This allows different translation units to contain definitions of functions with the same signature, excluding constraints, without violating the one definition rule (6.2). That is, without incorporating the constraints in the signature, such functions would have the same mangled name, thus appearing as multiple definitions of the same function.

1.0.1 [defns.signature]
signature
<function> name, parameter type list (11.3.5), and enclosing namespace (if any), and requires-clause (17.10.2) (if any).

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

1.0.2 [defns.signature.templ]
signature
<function template> name, parameter type list (11.3.5), enclosing namespace (if any), return type, and template parameter list template-head, and requires-clause (17.10.2) (if any).

1.0.3 [defns.signature.member]
signature
<class member function> name, parameter type list (11.3.5), class of which the function is a member, cv-qualifiers (if any), and ref-qualifier (if any), and requires-clause (17.10.2) (if any).

1.0.4 [defns.signature.member.templ]
signature
<class member function template> name, parameter type list (11.3.5), class of which the function is a member, cv-qualifiers (if any), ref-qualifier (if any), return type, and template parameter list template-head, and requires-clause (17.10.2) (if any).
5 Lexical conventions

5.11 Keywords

In 5.11, add the keywords concept and requires to Table 5 in the C++ Standard.
6 Basic concepts

6.1 Basic concepts
Add concepts to the list of entities.

3 An entity is a value, object, reference, function, enumerator, type, class member, bit-field, template, concept, template specialization, namespace, or parameter pack.

6.2 One-definition rule
Modify paragraph 1.

1 No translation unit shall contain more than one definition of any variable, function, class type, enumeration type, \textit{or} template, \textit{or} concept.

Modify paragraph 6. The full requirements allowing multiple definitions in different translation units are unchanged and therefore omitted in this text.

6 There can be more than one definition of a class type (Clause 12), enumeration type (10.2), inline function with external linkage (10.1.6), inline variable with external linkage (10.1.6), class template (Clause 17), non-static function template (17.5.5), static data member of a class template (17.5.1.3), member function of a class template (17.5.1.1), \textit{or} template specialization for which some template parameters are not specified (17.7, 17.5.4), \textit{or} concept in a program provided that each definition appears in a different translation unit, and provided the definitions satisfy the following requirements.
8 Expressions

 Modify paragraph 8 to include a reference to requires-expressions.

1 In some contexts, unevaluated operands appear (8.1.7, 8.2.8, 8.3.3, 8.3.7, 10.1.7.2, Clause 17).

8.1 Primary expressions

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

primary-expression:
  literal
  this
  ( expression )
  id-expression
  lambda-expression
  fold-expression
  requires-expression

8.1.4 Names

Add new paragraphs to the end of this section.

3 An id-expression that denotes the specialization of a concept (17.5.6) results in a prvalue of type bool. The expression is true if the concept’s normalized (17.10.2) constraint-expression is satisfied according to the rules in 17.10.1 and false otherwise. [Example:

    template<typename T> concept C = true;
    static_assert(C<int>); // OK

— end example]  [Note: A concept’s constraints are also considered when using a template name (17.2) and overload resolution (16), and they are compared during the the partial ordering of constraints (17.10.4). — end note]

4 A program that refers explicitly or implicitly to a function with a requires-clause whose constraint-expression is not satisfied (17.10.2), other than to declare it, is ill-formed. [Example:

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

In each case the constraints of f are not satisfied. In the declaration of p2, those constraints are required to be satisfied even though f is an unevaluated operand (Clause 8). — end example]

8.1.7 Requires expressions

Add this section to 8.1.

1 A requires-expression provides a concise way to express requirements on template arguments. A requirement is one that can be checked by name lookup (6.4) or by checking properties of types and expressions.
requires-expression:

  requires requirement-parameter-list opt requirement-body

requirement-parameter-list:

  ( parameter-declaration-clause opt )

requirement-body:

  { requirement-seq }

requirement-seq:

  requirement

  requirement-seq requirement

requirement:

  simple-requirement

  type-requirement

  compound-requirement

  nested-requirement

2 A requires-expression is a prvalue of type bool whose value is described below. Expressions appearing within a requirement-body are unevaluated operands (Clause 8).

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

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

   A requires-expression can also be used in a requires-clause (Clause 17) as a way of writing ad hoc constraints on template arguments such as the one below:

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

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

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

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

   —end note]

4 A requires-expression may introduce local parameters using a parameter-declaration-clause (11.3.5). A local parameter of a requires-expression shall not have a default argument. Each name introduced by a local parameter is in scope from the point of its declaration until the closing brace of the requirement-body. These parameters have no linkage, storage, or lifetime; they are only used as notation for the purpose of defining requirements. The parameter-declaration-clause of a requirement-parameter-list shall not terminate with an ellipsis. [Example:

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

§ 8.1.7
The requirement-body is comprised of a sequence of requirements. These requirements may refer to local parameters, template parameters, and any other declarations visible from the enclosing context.

The substitution of template arguments into a requires-expression may result in the formation of invalid types or expressions in its requirements or the violation of the semantic constraints of those requirements. In such cases, the requires-expression evaluates to false; it does not cause the program to be ill-formed. The substitution and semantic constraint checking proceeds in lexical order and stops when a condition that determines the result of the requires-expression is encountered. If substitution (if any) and semantic constraint checking succeed, the requires-expression evaluates to true. [Note: If a requires-expression contains invalid types or expressions in its requirements, and it does not appear within the declaration of a templated entity, then the program is ill-formed. —end note] If the substitution of template arguments into a requirement would always result in a substitution failure, the program is ill-formed; no diagnostic required.

Example:

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

§ 8.1.7.1 Simple requirements

A simple requirement asserts the validity of an expression. [Note: The enclosing requires-expression will evaluate to false if substitution of template arguments into the expression fails. The expression is an unevaluated operand (Clause 8). —end note]

Example:

```cpp
template<typename T> concept C =
requires (T a, T b) {
    a + b; // C<T> is true if a + b is a valid expression
};
```

§ 8.1.7.2 Type requirements

A type requirement asserts the validity of a type. [Note: The enclosing requires-expression will evaluate to false if substitution of template arguments fails. —end note] Example:

```cpp
template<typename T, typename T::type = 0> struct S;
template<typename T> using Ref = T&;
template<typename T> concept C =
requires {
    typename T::inner; // required nested member name
    typename S<T>; // required class template specialization
    typename Ref<T>; // required alias template substitution, fails if T is void
};
```
A type requirement that names a class template specialization does not require that type to be complete (6.9).

### 8.1.7.3 Compound requirements

A compound-requirement asserts properties of the expression \( E \). Substitution of template arguments (if any) and verification of semantic properties proceed in the following order:

1. **Substitution of template arguments (if any) into the expression** is performed.
2. **If the noexcept specifier** is present, \( E \) **shall not be a potentially-throwing expression** (18.4).
3. **If the return-type-requirement** is present, then:
   1. **Substitution of template arguments (if any) into the return-type-requirement** is performed.
   2. **If the return-type-requirement is a trailing-return-type**, \( E \) is implicitly convertible to the type named by the trailing-return-type. If conversion fails, the enclosing requires-expression is false.
   3. **If the return-type-requirement starts with a constrained-parameter (17.1)**, the expression is deduced against an invented function template \( F \) using the rules in 17.8.2.1. \( F \) is a void function template with a single type template parameter \( T \) declared with the constrained-parameter. Form a new cv-qualifier-seq \( cv \) by taking the union of const and volatile specifiers around the constrained-parameter. \( F \) has a single parameter whose type-specifier is \( cv \ T \) followed by the abstract-declarator. If deduction fails, the enclosing requires-expression is false.

**Example:**

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

The compound-requirement in \( \text{C1} \) requires that the expression \( x++ \) is valid. It is equivalent to a simple-requirement with the same expression.

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

The compound-requirement in \( \text{C2} \) requires that \( *x \) is a valid expression, that \( \text{typename T::inner} \) is a valid type, and that \( *x \) is implicitly convertible to \( \text{typename T::inner} \).

```cpp
template<typename T, typename U> concept C3 = false;
template<typename T> concept C4 =
    requires(T x) {
        {*x} -> C3<int> const&;
    };
```

[Example:]

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

The compound-requirement in \( \text{C1} \) requires that the expression \( x++ \) is valid. It is equivalent to a simple-requirement with the same expression.

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

The compound-requirement in \( \text{C2} \) requires that \( *x \) is a valid expression, that \( \text{typename T::inner} \) is a valid type, and that \( *x \) is implicitly convertible to \( \text{typename T::inner} \).
The *compound-requirement* requires that \( x \) be deduced as an argument for the invented function:

```cpp
template<
    C3<int> X>
void f(X const&);
```

In this case, deduction always fails since \( C3 \) is *false*.

```cpp
template<
typename T>
concept C5 =
    requires(T x) {
        {g(x)} noexcept;
    };
```

The *compound-requirement* in \( C5 \) requires that \( g(x) \) is a valid expression and that \( g(x) \) is non-throwing. —end example

### 8.1.7.4 Nested requirements

A nested-requirement can be used to specify additional constraints in terms of local parameters. The constraint-expression shall be satisfied (17.10.2) by the substituted template arguments, if any. Substitution of template arguments into a nested-requirement does not result in substitution into the constraint-expression other than as specified in 17.10.2. [Example:

```cpp
template<typename U>
concept C = sizeof(U) == 1;

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

\( D<T> \) is satisfied if \( sizeof(decltype (+t)) == 1 \) (17.10.1.2). —end example]

[Note: Normalization of constraints appends a separate constraint for each nested-requirement within a requires-expression for the purpose of determining partial ordering (17.10.4). —end note]

A local parameter shall only appear as an unevaluated operand (Clause 8) within the constraint-expression. [Example:

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

—end example]
11 Declarators

In paragraph 1, modify the grammar of init-declarator to allow the specification of constraints on function declarations.

\[\text{init-declarator:}\]
\[\quad\text{declarator initizer}_{\text{opt}}\]
\[\quad\text{declarator requires-clause}\]

Add the following paragraph after paragraph 3.

3 The optional requires-clause (Clause 17) in an init-declarator or member-declarator shall be present only when the declarator declares a function (11.3.5). When present after a declarator, the requires-clause is called the trailing requires-clause. The trailing requires-clause introduces the constraint-expression that results from interpreting its constraint-logical-or-expression as a constraint-expression. [Example:
\begin{verbatim}
void f1(int a) requires true;  // OK
auto f2(int a) -> bool requires true;  // OK
auto f3(int a) requires true -> bool;  // error: requires-clause precedes trailing-return-type
void (*pf)() requires true;      // error: constraint on a variable
void g(int (*)(char) requires true);
// error: constraint on a parameter-declaration
auto* p = new void(*)(char) requires true;  // error: not a function declaration
\end{verbatim}
—end example]

11.3 Meaning of declarators

11.3.5 Functions

Modify the first part of paragraph 5. The unchanged remainder of the paragraph is omitted.

5 A single name can be used for several different functions in a single scope; this is function overloading (Clause 16). All declarations for a function shall agree exactly in both the return type and the parameter-type-list. All declarations for a function shall have equivalent return types, parameter-type-lists, and requires-clauses (17.5.5.1).

Modify paragraph 8 to exclude constraints from the type of a function. Note that the change occurs in the sentence following the example in the C++ Standard.

8 The return type, the parameter-type-list, the ref-qualifier, the cv-qualifier-seq, and the exception specification, but not the default arguments (11.3.6) or requires-clauses (Clause 17) are part of the function type.

11.5 Function definitions

11.5.1 In general

Change in paragraph 1:

\[\text{function-definition:}\]
\[\quad\text{attribute-specifier-seq}_{\text{opt}}\]
\[\quad\text{decl-specifier-seq}_{\text{opt}}\]
\[\quad\text{declarator virt-specifier-seq}_{\text{opt}}\]
\[\quad\text{function-body}\]
\[\quad\text{attribute-specifier-seq}_{\text{opt}}\]
\[\quad\text{decl-specifier-seq}_{\text{opt}}\]
\[\quad\text{declarator requires-clause function-body}\]
11.5.2 Explicitly-defaulted functions

Change in paragraph 1:

A function definition of the form \[
\text{...} \text{ with the function-body = \text{default} ;}
\]

is called an \textit{explicitly-defaulted} definition.

11.5.3 In general

Change in paragraph 1:

A function definition of the form \[
\text{...} \text{ with the function-body = \text{delete} ;}
\]

is called a \textit{deleted} function.
12 Classes

12.2 Class members

Change grammar in paragraph 1:

```
member-declarator:
  declarator virt-specifier-seq_opt pure-specifier_opt
  declarator requires-clause
  declarator brace-or-equal-initializer_opt
  identifier_opt attribute-specifier-seq_opt : constant-expression
```

§ 12.2
13 Derived classes [class.derived]

13.3 Virtual functions [class.virtual]

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

A virtual function shall not have a requires-clause. [Example:

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

—end example]
16 Overloading

Modify paragraph 1 to allow overloading based on constraints, by removing the repeated wording.

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

16.1 Overloadable declarations

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

[Note: As specified in 11.3.5, function declarations that have equivalent parameter declarations and requires-clauses, if any (17.10.2), declare the same function and therefore cannot be overloaded: ... — end note]

16.2 Declaration matching

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

Two function declarations of the same name refer to the same function if they are in the same scope and have equivalent parameter declarations (16.1) and equivalent requires-clauses, if any (17.10.2).

16.3 Overload resolution

16.3.2 Viable functions

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

From the set of candidate functions constructed for a given context (16.3.1), 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 (16.3.3). The selection of viable functions considers associated constraints, if any (17.10.2), and relationships between arguments and function parameters other than the ranking of conversion sequences.

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

Second, for a function to be viable, if it has associated constraints, those constraints shall be satisfied (17.10.2).

Second, Third, for F to be a viable function...

16.3.3 Best viable function

Modify paragraph 1 by adding a rule to the the criteria that determines when one function is better than another.
Given these definitions, a viable function $F_1$ is defined to be a better function than another viable function $F_2$ if for all arguments $i$, $\text{ICS}_i(F_1)$ is not a worse conversion sequence than $\text{ICS}_i(F_2)$, and then

- for some argument $j$, $\text{ICS}_j(F_1)$ is a better conversion sequence than $\text{ICS}_j(F_2)$, or, if not that,
- the context is an initialization by user-defined conversion (see 11.6, 16.3.1.5, and 16.3.1.6) and the standard conversion sequence from the return type of $F_1$ to the destination type (i.e., the type of the entity being initialized) is a better conversion sequence than the standard conversion sequence from the return type of $F_2$ to the destination type [Example:

```c
struct A {
    A();
    operator int();
    operator double();
} a;
int i = a; // a.operator int() followed by no conversion is better than
// a.operator double() followed by a conversion to int
float x = a; // ambiguous: both possibilities require conversions,
// and neither is better than the other
```
— end example] or, if not that,
- the context is an initialization by conversion function for direct reference binding (16.3.1.6) of a reference to function type, the return type of $F_1$ is the same kind of reference (i.e. lvalue or rvalue) as the reference being initialized, and the return type of $F_2$ is not [Example:

```c
template <class T> struct A {
    operator T&(); // #1
    operator T&&(); // #2
};
typedef int Fn();
A<Fn> a;
Fn& lf = a; // calls #1
Fn&& rf = a; // calls #2
```
— end example] or, if not that,
- $F_1$ is not a function template specialization and $F_2$ is a function template specialization, or, if not that,
- $F_1$ and $F_2$ are function template specializations, and the function template for $F_1$ is more specialized than the template for $F_2$ according to the partial ordering rules described in 17.5.5.2, or, if not that,
- $F_1$ and $F_2$ are non-template functions with the same parameter-type-lists, and $F_1$ is more constrained than $F_2$ according to the partial ordering of constraints described in 17.10.4, or if not that,
- $F_1$ is generated from a deduction-guide (16.3.1.8) and $F_2$ is not, or, if not that,
- $F_1$ is the copy deduction candidate (16.3.1.8) and $F_2$ is not, or, if not that,
- $F_1$ is generated from a non-template constructor and $F_2$ is generated from a constructor template. [Example:

```c
template <class T> struct A {
    using value_type = T;
    A(value_type); // #1
    A(const A&); // #2
    A(T, T, int); // #3
```
template<class U>
    A(int, T, U);  // #4
    // #5 is the copy deduction candidate, A(A)
};

A x(1, 2, 3);  // uses #3, generated from a non-template constructor

template <class T>
    A(T) -> A<T>;  // #6, less specialized than #5

A a(42);      // uses #6 to deduce A<int> and #1 to initialize
A b = a;      // uses #5 to deduce A<int> and #2 to initialize

template <class T>
    A(A<T>) -> A<A<T>>;  // #7, as specialized as #5

A b2 = a;      // uses #7 to deduce A<A<int>> and #1 to initialize

— end example 

16.4 Address of overloaded function

Modify paragraph 4 to incorporate constraints in the selection of an overloaded function when its address is taken.

4 Eliminate from the set of selected functions all those whose associated constraints are not satisfied (17.10.2). If more than one function is selected If more than one function in the set remains, any function template specializations in the set are eliminated if the set also contains a function that is not a function template specialization, and. Any given non-template function F0 is eliminated if the set contains a second non-template function that is more constrained than F0 according to the partial ordering rules of 17.10.4. Additionally, any given function template specialization F1 is eliminated if the set contains a second function template specialization whose function template is more specialized than the function template of F1 according to the partial ordering rules of 17.5.5.2. After such eliminations, if any, there shall remain exactly one selected function.
17 Templates

Modify the template-declaration grammar in paragraph 1.

A template defines a family of classes, functions, or variables, or a concept, or an alias for a family of types.

Template-declaration:
  template-head template < template-parameter-list > declaration
  template-head concept-definition

Template-head:
  template < template-parameter-list > requires-clause_opt

Concept-definition:
  concept concept-name = constraint-expression

Concept-name:
  identifier

Requires-clause:
  requires constraint-logical-or-expression

Constraint-logical-and-expression:
  primary-expression
  constraint-logical-and-expression && primary-expression

Constraint-logical-or-expression:
  constraint-logical-and-expression
  constraint-logical-or-expression || constraint-logical-and-expression

Allow concepts as templates, and make concept definitions also a definition in paragraph 1.

The declaration in a template-declaration shall

(1.1) — declare or define a function, a class, or a variable, or
(1.2) — define a member function, a member class, a member enumeration, or a static data member
  of a class template or of a class nested within a class template, or
(1.3) — define a member template of a class or class template, or
(1.4) — be a deduction-guide, or
(1.5) — be an alias-declaration, or
(1.6) — be a concept-definition.

A template-declaration is a declaration. A template-declaration is also a definition if its declaration
either is a concept-definition or defines a function, a class, a variable, or a static data member. A
declaration introduced by a template declaration of a variable is a variable template. A variable
template at class scope is a static data member template.

Add the following paragraphs after paragraph 6.

A template-declaration is written in terms of its template parameters. The optional requires-
clause following a template-parameter-list allows the specification of constraints (17.10.2) on
template arguments (17.3). The requires-clause introduces the constraint-expression that results
from interpreting the constraint-logical-or-expression as a constraint-expression. The constraint-
logical-or-expression of a requires-clause is an unevaluated operand (Clause 8).
```cpp
template<int N> requires N == sizeof unsigned short
int f();  // error: parentheses required around == expression
```

— end example ]

## 17.1 Template parameters

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

The syntax for template-parameters is:

```plaintext
template-parameter:
  type-parameter
  parameter-declaration
  constrained-parameter

type-parameter:
  type-parameter-key ...opt identifieropt
  type-parameter-key identifieropt = type-id

template < template-parameter-list > type-parameter-key ...opt identifieropt

template < template-parameter-list > type-parameter-key identifieropt = id-expression

type-parameter-key:
  class
  typename

constrained-parameter:
  qualified-concept-name ...opt identifieropt
  qualified-concept-name identifieropt default-template-argumentopt

qualified-concept-name:
  nested-name-specifieropt concept-name
  nested-name-specifieropt partial-concept-id

partial-concept-id:
  concept-name < template-argument-listopt>

default-template-argument:
  = type-id
  = id-expression
  = initializer-clause
```

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

A partial-concept-id is a concept-name followed by a sequence of template-arguments. These template arguments are used to form a constraint-expression as described below.

A constrained-parameter declares a template parameter whose kind (type, non-type, template) and type match that of the prototype parameter (17.5.6) of the concept designated by the qualified-concept-name in the constrained-parameter. Let X be the prototype parameter of the designated concept. The declared template parameter is determined by the kind of X (type, non-type, template) and the optional ellipsis in the constrained-parameter as follows.

- (10.1) If X is a type template-parameter, the declared parameter is a type template-parameter.
- (10.2) If X is a non-type template-parameter, the declared parameter is a non-type template-parameter having the same type as X.
- (10.3) If X is a template template-parameter, the declared parameter is a template template-parameter having the same template-parameter-list as X, excluding default template arguments.
— If the *qualified-concept-name* is followed by an ellipsis, then the declared parameter is a template parameter pack (17.5.3).

**Example:**

```cpp
template<typename T> concept C1 = true;
template<typename... Ts> concept C4 = true;
template<char... Cs> concept C5 = true;
template<typename T> void f1(); // OK: T is a type template-parameter
template<typename... Ts> void f4(); // OK: Ts is a template parameter pack of types
```

— end example

A *constrained-parameter* introduces a constraint-expression (17.10.2). The expression is derived from the *qualified-concept-name* Q in the constrained-parameter, its designated concept C, and the declared template parameter P.

— First, form a template argument A from P. If P declares a template parameter pack (17.5.3) and C is a variadic concept (17.5.6), then A is the pack expansion P... Otherwise, A is the *id-expression* P.

— Then, form an *id-expression* E as follows. If Q is a *concept-name*, then E is C<A>. Otherwise, Q is a partial-concept-id of the form C<A1, A2, ..., AN>, and E is C<A, A1, A2, ..., AN>.

— Finally, if P declares a template parameter pack and C is not a variadic concept, E is adjusted to be the fold-expression (E && ...) (8.1.6).

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

```cpp
template<typename T> concept C1 = true;
template<typename... Ts> concept C4 = true;
template<typename T> void f1(); // OK: T is a type template-parameter
```

— end example

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

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

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

— end example]

17.2 Names of template specializations

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

8 When the template-name of a simple-template-id names a constrained non-function template or
a constrained template template-parameter, but not a member template that is a member of
an unknown specialization (17.6), and all template-arguments in the simple-template-id are non-
dependent (17.6.2.4), the associated constraints of the constrained template shall be satisfied.
(17.10.2). [Example:

```cpp
template<typename T> concept C1 = sizeof(T) != sizeof(int);

template<C1 T> struct S1 { }

template<C1 T> using Ptr = T*;

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

```cpp
template<typename T>
 struct S2 { Ptr<int> x; }; // error, no diagnostic required

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

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

```cpp
template<template<C1 T> class X>
 struct S4 {
     X<int> x; // error, no diagnostic required
 };
```

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

template<C2 T> struct S { }

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

17.3 Template arguments

Add the following example to the end of paragraph 3, after the examples given in the C++ Standard.

```cpp
[Example:

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

template<typename T> concept D = C<T> && requires (T t) { t.g(); };
```
template<template<C> class P>  
   struct S { }; 

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

S<X> s1; // OK: X and P have equivalent constraints  
S<Y> s2; // error: P is not at least as specialized as Y  
S<Z> s3; // OK: P is at least as specialized as Z  
— end example ]

17.5 Template declarations [temp.decls]

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

2 For purposes of name lookup and instantiation, default arguments, partial-concept-ids, requires-clauses (Clause 17), and noexcept-specifiers of function templates and default arguments, partial-concept-ids, requires-clauses, and noexcept-specifiers of member functions of class templates are considered definitions; each default argument, partial-concept-ids, requires-clause, or noexcept-specifier is a separate definition which is unrelated to the function template definition or to any other default arguments or noexcept-specifiers. For the purpose of instantiation, the substatements of a constexpr if statement (9.4.1) are considered definitions.

17.5.1 Class templates [temp.class]

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

3 When a member function, a member class, a member enumeration, a static data member or a member template of a class template is defined outside of the class template definition, the member definition is defined as a template definition in which the template parameters are those template-head is equivalent to that of the class template (17.5.5.1). The names of the template parameters used in the definition of the member may be different from the template parameter names used in the class template definition. The template argument list following the class template name in the member definition shall name the parameters in the same order as the one used in the template parameter list of the member. Each template parameter pack shall be expanded with an ellipsis in the template argument list.

Add the following example at the end of paragraph 3.

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

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

   template<C A> void S<A>::f() { } // OK: template-heads match  
   template<typename T> void S<T>::g() { } // error: no matching declaration for S<T>  
]
template<typename T> requires C<T>  // error (no diagnostic required): template-heads are
void S<T>::h() { }  // functionally equivalent but not equivalent

— end example

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

Add the following example to the end of paragraph 1.

[Example:

template<typename T> concept C = requires {
typename T::type;
};

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

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

— end example
]

17.5.2 Member templates [temp.mem]

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

A template can be declared within a class or class template; such a template is called a member template. A member template can be defined within or outside its class definition or class template definition. A member template of a class template that is defined outside of its class
template definition shall be specified with the template parameters a template-head equivalent
to that of the class template followed by the template parameters a template-head equivalent to
that of the member template.

Add the following example at the end of paragraph 1.

[Example:

template<typename T> concept C1 = true;
template<typename T> concept 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>

§ 17.5.2
17.5.3 Friends

Add a new paragraph to make non-template friends ill-formed.

A non-template friend declaration shall not have a requires-clause.

17.5.4 Class template partial specialization

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

A class template partial specialization may be constrained (Clause 17). [Example:

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

template<typename T> struct X { }; // #1
template<concept C T> struct X<T> { }; // #2
```

Both partial specializations are more specialized than the primary. #1 is more specialized because the deduction of its template arguments from the template argument list of the class template specialization succeeds, while the reverse does not. #2 is more specialized because the template arguments are equivalent, but the partial specialization is more constrained (17.10.4). —end example]

17.5.4.1 Matching of class template partial specializations

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

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 (17.8.2), and the deduced template arguments satisfy the associated constraints of the partial specialization, if any (17.10.2).

Add the following example to the end of paragraph 2.

[Example:

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

template<typename T> struct S { }; // #1
template<concept C T> struct S<T> { }; // #2

struct Arg { void f(); };

S<int> s1; // uses #1; the constraints of #2 are not satisfied
S<Arg> s2; // uses #2; both constraints are satisfied but #2 is more specialized
```

—end example]

17.5.4.2 Partial ordering of class template specializations

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

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 (17.5.5.2):
— the first function template has the same template parameters and associated constraints (17.10.2) as the first partial specialization, and has a single function parameter whose type is a class template specialization with the template arguments of the first partial specialization, and

— the second function template has the same template parameters and associated constraints (17.10.2) as the second partial specialization, and has a single function parameter whose type is a class template specialization with the template arguments of the second partial specialization.

Add the following example to the end of paragraph 1.

[Example:

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

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

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

The partial specialization #2 is more specialized than #1 because B is more specialized than A.
— end example]

17.5.5  Function templates  [temp.fct]

17.5.5.1  Function template overloading  [temp.over.link]

Add a new paragraph prior to paragraph 6:

6 Two template-heads are equivalent if their template-parameter-lists have the same length, corresponding template-parameters are equivalent, and if either has a requires-clause, they both have requires-clauses and the corresponding constraint-expressions are equivalent. Two template-parameters are equivalent under the following conditions:

(6.1) — they declare template parameters of the same kind,

(6.2) — if either declares a template parameter pack, they both do,

(6.3) — if they declare non-type template parameters, they have equivalent types,

(6.4) — if they declare template template parameters, their template parameters are equivalent, and

(6.5) — if either is declared with a qualified-concept-name, they both are, and the qualified-concept-names are equivalent.

When determining whether types or qualified-concept-names are equivalent, the rules above are used to compare expressions involving template parameters. Two template-heads are functionally equivalent if they accept and are satisfied by the same set of template argument lists.

Modify paragraph 6 (now paragraph 7) to expand the scope of rules governing the existing term "equivalent".

7 Two function templates are equivalent if they are declared in the same scope, have the same name, have identical template parameter lists equivalent template-heads, and have return types, parameter lists, and trailing requires-clauses (if any) that are equivalent using the rules
described above to compare expressions involving template parameters. Two function templates are functionally equivalent if they are declared in the same scope, have the same name, accept and are satisfied by the same set of template argument lists, and have return types and parameter lists that are equivalent except that one or more expressions that involve template parameters in the return types and parameter lists are functionally equivalent using the rules described above to compare expressions involving template parameters. If a program contains declarations of function templates that the validity or meaning of the program depends on whether two constructs are equivalent, and they are functionally equivalent but not equivalent, the program is ill-formed; no diagnostic is required.

17.5.5.2 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 17.10.4.

17.5.6 Concept definitions

Add the following paragraph.

A concept is a template that defines constraints on its template arguments.

A concept-definition declares its identifier to be a concept. The name of the concept is a concept-name. [Example:

```c
template<typename T>
concept C = requires(T x) {
    { x == x } -> bool;
};
template<typename T>
    requires C<T> // C constrains f1(T) in constraint-expression
T f1(T x) { return x; }
template<C T> // C constrains f2(T) as a constrained-parameter
T f2(T x) { return x; }
```

—end example]

A concept-definition shall appear in the global scope or in a namespace scope (6.3.6).

A concept shall not have associated constraints (17.10.2).

A concept is not instantiated (17.7). A program that explicitly instantiates (17.7.2), explicitly specializes (17.7.3), or partially specializes a concept is ill-formed. [Note: An id-expression that denotes a concept specialization is evaluated as an expression (8.1.4). —end note]

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.

17.6 Name resolution

Modify paragraph 8.

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

17.7 Template instantiation and specialization

17.7.1 Implicit Instantiation

Change paragraph 1 to include associated constraints.

1 Unless a class template specialization has been explicitly instantiated (17.7.2) or explicitly specialized (17.7.3), the class template specialization is implicitly instantiated when the specialization is referenced in a context that requires a completely-defined object type or when the completeness of the class type affects the semantics of the program. [Note: In particular, if the semantics of an expression depend on the member or base class lists of a class template specialization, the class template specialization is implicitly generated. For instance, deleting a pointer to class type depends on whether or not the class declares a destructor, and a conversion between pointers to class type depends on the inheritance relationship between the two classes involved. —end note] [Example:

```c
template<class T> class B { ... };  
template<class T> class D : public B<T> { ... };  

void f(void*);  
void f(B<int>*);  

void g(D<int>* p, D<char>* pp, D<double>* ppp) {  
    f(p);  // instantiation of D<int> required: call f(B<int>*)  
    B<char>* q = pp;  // instantiation of D<char> required: convert D<char>* to B<char>*  
    delete ppp;  // instantiation of D<double> required
}
```

—end example] If a class template has been declared, but not defined, at the point of instantiation (17.6.4.1), the instantiation yields an incomplete class type (6.9). [Example:

```c
template<class T> class X;  
X<char> ch;  // error: incomplete type X<char>
```

—end example] [Note: Within a template declaration, a local class (or enumeration and the members of a local class are never considered to be entities that can be separately instantiated (this includes their default arguments, noexcept-specifiers, and non-static data member initializers, if any, but not their partial-concept-ids or requires-clauses). As a result, the dependent names are looked up, the semantic constraints are checked, and any templates used are instantiated as part of the instantiation of the entity within which the local class or enumeration is declared. —end note]

Add a new paragraph at the end of this section to describe how associated constraints are instantiated.

16 The partial-concept-ids and requires-clause of a template specialization or member function are not instantiated along with the specialization or function itself, even for a member function of a

§ 17.7.1
local class; substitution into the the atomic constraints formed from them is instead performed as
specified in 17.10.2 and 17.10.1.2 when determining whether the constraints are satisfied. [Note: The
satisfaction of constraints is determined during name lookup or overload resolution (16.3).
—end note] [Example:

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

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

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

When S<char> is instantiated, both constructors are part of the specialization. However, their
constraints will never be satisfied. This also has the effect of suppressing the implicit generation
of a default constructor (15.1). —end example] [Example:

```cpp
template<typename T> struct S1 {
    template<typename U>
    requires false
    struct Inner1; // error: ill-formed, no diagnostic required
};

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

The class S1<T>::Inner1 is ill-formed, no diagnostic required, because it has no valid special-
izations. S2 is ill-formed, no diagnostic required, since no substitution into the constraints of its
Inner2 template would result in a valid expression. —end example]

### 17.7.2 Explicit instantiation

Add the following note after paragraph 7.

[Note: An explicit instantiation of a constrained template shall satisfy that template’s associated
constraints (17.10.2). The satisfaction of constraints is determined when forming the template
name of an explicit instantiation in which all template arguments are specified (17.2), or for
explicit instantiations of function templates, during template argument deduction (17.8.2.6) when
one or more trailing template arguments are left unspecified. —end note]

Modify paragraph 8 in the C++ standard (paragraph 9, here) to ensure that only members whose constraints
are satisfied are explicitly instantiated during class template specialization. The note in the C++ Standard
is omitted.

An explicit instantiation that names a class template specialization is also an explicit instantiation
of the same kind (declaration or definition) of each of its members (not including members
inherited from base classes and members that are templates) that has not been previously ex-
licitly specialized in the translation unit containing the explicit instantiation, and provided that
the associated constraints, if any, of that member are satisfied by the template arguments of the
explicit instantiation (17.10.2), except as described below.
17.7.3 Explicit specialization

Add the following note after paragraph 10.

11 [Note: An explicit specialization of a constrained template shall satisfy that template’s associated constraints (17.10.2). The satisfaction of constraints is determined when forming the template name of an explicit specialization in which all template arguments are specified (17.2), or for explicit specializations of function templates, during template argument deduction (17.8.2.6) when one or more trailing template arguments are left unspecified. —end note]

17.8 Function template specializations

17.8.2 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 (17.10.2), those constraints are checked for satisfaction (17.10). If the constraints are not satisfied, type deduction fails.
17.10 Template constraints

Add this section after 17.9 in the C++ standard.

[Note: This section defines the meaning of constraints on template arguments. The abstract syntax and satisfaction rules are defined in 17.10.1. Constraints are associated with declarations in 17.10.2. Declarations are partially ordered by their associated constraints (17.10.4). —end note]

17.10.1 Constraints

A constraint is a sequence of logical operations and operands that specifies requirements on template arguments. The operands of a logical operation are constraints. There are several different kinds of constraints:

1. 1 conjunctions (17.10.1.1),
2. 2 disjunctions (17.10.1.1), and
3. 3 atomic constraints (17.10.1.2)

In order for a constrained template to be instantiated (17.7), its associated constraints shall be satisfied (17.10.2). [Note: Forming a template specialization name (17.2) of a class template, variable template, or an alias template requires the satisfaction of its constraints. Overload resolution (16.3.2) requires the satisfaction of constraints on functions and function templates. —end note] The rules for determining the satisfaction of different kinds of constraints are defined in the following subsections.

17.10.1.1 Logical operations

There are two binary logical operations on constraints: conjunction and disjunction. [Note: These logical operations have no corresponding C++ syntax. For the purpose of exposition, conjunction is spelled using the symbol \( \land \) and disjunction is spelled using the symbol \( \lor \). The operands of these operations are called the left and right operands. In the constraint \( A \land B \), \( A \) is the left operand, and \( B \) is the right operand. —end note]

A conjunction is a constraint taking two operands. To determine if a conjunction is satisfied, the satisfaction of the first operand is checked. If that is not satisfied, the conjunction is not satisfied. Otherwise, the conjunction is satisfied if and only if the second operand is satisfied.

A disjunction is a constraint taking two operands. To determine if a disjunction is satisfied, the satisfaction of the first operand is checked. If that is satisfied, the disjunction is satisfied. Otherwise, the disjunction is satisfied if and only if the second operand is satisfied.

[Example:

```cpp
template<
type
> const expr bool get_value() { return T::value; }

template<
type
> requires (sizeof(T) > 1) && get_value<T>()
void f(T); // has associated constraint sizeof(T) > 1 \land get_value<T>()

void f(int);

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

In the satisfaction of the associated constraints (17.10.2) of \( f \), the constraint \( \text{sizeof(char)} > 1 \) is not satisfied; the second operand is not checked for satisfaction. —end example]
17.10.1.2 Atomic constraints

An atomic constraint is formed from an expression $E$ and a mapping from the template parameters that appear within $E$ to template arguments involving the template parameters of the constrained entity, called the parameter mapping (17.10.2). Two atomic constraints are identical if they are formed from the same expression and the targets of the parameter mappings are equivalent according to the rules for expressions described in 17.5.5.1. [Note: Atomic constraints are formed by constraint normalization (17.10.3). $E$ is never a logical AND expression (8.14) nor a logical OR expression (8.15). —end note] Determining if a constraint is satisfied entails the substitution of the parameter mapping and template arguments into that constraint. If substitution results in an invalid type or expression, the constraint is not satisfied. Otherwise, the lvalue-to-rvalue conversion (7.1) is performed if necessary, and $E$ shall be a constant expression of type $bool$. The constraint is satisfied if and only if evaluation of $E$ results in $true$. [Example:

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

template<typename T>
struct S {
    constexpr operator bool() const { return true; }
};

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

void g() {
    f(0); // error: expression S<int>{} does not have type bool
    // while checking satisfaction of deduced arguments of #1, // even though #2 is a better match
}

— end example]

17.10.2 Constrained declarations

A template declaration (Clause 17) or function declaration (11.3.5) can be constrained by the use of a requires-clause. This allows the specification of constraints for that declaration as an expression:

```
constraint-expression:
    logical-or-expression
```

Constraints can also be associated with a declaration through the use of constrained-parameters in a template-parameter-list. Each of these forms introduces additional constraint-expressions that are used to constrain the declaration.

A template’s associated constraints are defined as follows:

(3.1) — If there are no introduced constraint-expressions, the declaration has no associated constraints.

(3.2) — If there is a single introduced constraint-expression, the associated constraints are the normal form (17.10.3) of that expression.

(3.3) — Otherwise, the associated constraints are the normal form of a logical AND expression (8.14) whose operands are in the following order:
— the constraint-expression introduced by each constrained-parameter (17.1) in the declaration’s template-parameter-list, in order of appearance, and

— the constraint-expression introduced by a requires-clause following a template-parameter-list (Clause 17), and

— the constraint-expression introduced by a trailing requires-clause (Clause 11) of a function declaration (11.3.5).

The formation of the associated constraints establishes the order in which constraints are instantiated when checking for satisfaction (17.10.1). [Example:

```cpp
template<typename T> concept C = true;
template<C T> void f1(T);
template<typename T> requires C<T> void f2(T);
template<typename T> void f3(T) requires C<T>;
```

The functions f1, f2, and f3 have the associated constraint C<T>.

```cpp
template<typename T> concept C1 = true;
template<typename T> concept C2 = sizeof(T) > 0;
template<C1 T> void f4(T) requires C2<T>;
template<typename T> requires C1<T> && C2<T> void f5(T);
```

The associated constraints of f4 and f5 are C1<T> ∧ C2<T>.

```cpp
template<C1 T> requires C2<T> void f6();
template<C2 T> requires C1<T> void f7();
```

The associated constraints of f6 are C1<T> ∧ C2<T>, and those of f7 are C2<T> ∧ C1<T>. — end example]

**17.10.3 Constraint normalization** [temp.constr.normal]

Determining a declaration’s associated constraints (17.10.1) requires that their constraint-expressions are normalized. Normalization transforms a constraint-expression into a sequence of conjunctions and disjunctions (17.10.1.1) of atomic constraints (17.10.1.2). The normal form of an expression E is defined as follows:

1. The normal form of an expression (E) is the normal form of E.
2. The normal form of an expression E1 || E2 is the disjunction (17.10.1.1) of the normal forms of E1 and E2.
3. The normal form of an expression E1 && E2 is the conjunction of the normal forms of E1 and E2.
4. The normal form of an id-expression of the form C<A1, A2, ..., AN>, where C names a concept, is the normal form of the constraint-expression of C, after substituting A1, A2, ..., AN for C’s respective template parameters in the parameter mappings in each atomic constraint. If any such substitution results in an invalid type or expression, the program is ill-formed; no diagnostic is required. [Example:

```cpp
template<typename T> concept A = T::value || true;
template<typename U> concept B = A<U*>;
template<typename V> concept C = B<V*>;
```

§ 17.10.3
Normalization of B’s constraint-expression is valid and results in T::value (with the mapping T↦→U*) ∧ true (with an empty mapping), despite the expression T::value being ill-formed for a pointer type T. Normalization of C’s constraint-expression results in the program being ill-formed, because it would form the invalid type T&* in the parameter mapping.

— end example —

The normal form of any other expression E is the atomic constraint whose expression is E and whose parameter mapping is the identity mapping.

[Example:

```cpp
template<typename T> concept C1 = sizeof(T) == 1;
template<typename T> concept C2 = C1<T>() && 1 == 2;
template<typename T> concept C3 = requires { typename T::type; };
template<typename T> concept C4 = requires (T x) { ++x; }

template<C2 U> void f1(U);  // #1
template<C3 U> void f2(U);  // #2
template<C4 U> void f3(U);  // #3
```

The associated constraints of #1 are sizeof(T) == 1 (with mapping T↦→U) ∧ 1 == 2, those of #2 are requires { typename T::type; } (with mapping T↦→U), those of #3 are requires (T x) { ++x; } (with mapping T↦→U). — end example]

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

A constraint P is said to subsume another constraint Q if it can be determined that P implies Q, up to the identity of atomic constraints in P and Q (17.10.1.2), as described below. [Example: Subsumption does not determine if the atomic constraint N >= 0 (17.10.1.2) subsumes N > 0 for some integral template argument N. — end example]

In order to determine if a constraint P subsumes a constraint Q, P is transformed into disjunctive normal form, and Q is transformed into conjunctive normal form1. Then, P subsumes Q if and only if

(1) for every disjunctive clause Pi in the disjunctive normal form of P, Pi subsumes every conjunctive clause Qi in the conjunctive normal form of Q, where

(2) a disjunctive clause Pi subsumes a conjunctive clause Qj if and only if there exists an atomic constraint Pa in Pi for which there exists an atomic constraint, Qjb, in Qj such that Pa subsumes Qjb.

(3) an atomic constraint A subsumes another atomic constraint B if and only if the A and B are identical using the rules described in 17.10.1.2.

[Example: Let A and B be atomic constraints (17.10.1.2). The constraint A ∧ B subsumes A, but A does not subsume A ∧ B. The constraint A subsumes A ∨ B, but A ∨ B does not subsume A. Also note that every constraint subsumes itself. — end example]

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

(3) the best viable candidate of non-template functions (16.3.3),

1 A constraint is in disjunctive normal form when it is a disjunction of clauses where each clause is a conjunction of atomic constraints. Similarly, a constraint is in conjunctive normal form when it is a conjunction of clauses where each clause is a disjunction of atomic constraints. [Example: Let A, B, and C be atomic constraints, which can be grouped using parentheses. The constraint A ∧ (B ∨ C) is in conjunctive normal form. Its conjunctive clauses are A and (B ∨ C). The disjunctive normal form of the constraint A ∧ (B ∨ C) is (A ∧ B) ∨ (A ∧ C). Its disjunctive clauses are (A ∧ B) and (A ∧ C). — end example]
— the address of a non-template function (16.4),
— the matching of template template arguments (17.3.3),
— the partial ordering of class template specializations (17.5.4.2), and
— the partial ordering of function templates (17.5.5.2).

— end note]

4 When two declarations \( D_1 \) and \( D_2 \) are partially ordered by their associated constraints (17.10.2) \( D_1 \) is at least as constrained as \( D_2 \) if

(4.1) \( D_1 \) and \( D_2 \) are both constrained declarations and \( D_1 \)'s associated constraints subsume those of \( D_2 \); or

(4.2) \( D_2 \) has no associated constraints.

5 A declaration \( D_1 \) is more constrained than another declaration \( D_2 \) when \( D_1 \) is at least as constrained as \( D_2 \), and \( D_2 \) is not at least as constrained as \( D_1 \).

[ Example:

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

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

\[ f(0); \quad \text{// selects #1} \]
\[ f((\text{int}*)0); \quad \text{// selects #2} \]
\[ g(true); \quad \text{// selects #3 because C1<bool> is not satisfied} \]
\[ g(0); \quad \text{// selects #4} \]

— end example ]
Annex A  (informative)
Compatibility

Add the following to Appendix C in the C++ Standard.

A.1  C++ and ISO C++ 2017  [diff.iso]
This subclause lists the differences between C++ and ISO C++ 2017, by the chapters of this document.

A.1.1 Clause 5: lexical conventions  [diff.lex]
5.11
Change: New keywords.

Rationale: Required for new features. The requires keyword is added to introduce constraints through a requires-clause or a requires-expression. The concept keyword is added to enable the definition of concepts (17.5.6).
Effect on original feature: Valid ISO C++ 2017 code using concept or requires as an identifier is not valid in this International Standard.