Programming Languages — C++ Extensions for Concepts
Langages de programmation — Extensions C++ pour les concepts

Warning

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

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

This ISO document is a working draft or committee draft and is copyright-protected by ISO. While the reproduction of working drafts or committee drafts in any form for use by participants in the ISO standards development process is permitted without prior permission from ISO, neither this document nor any extract from it may be reproduced, stored or transmitted in any form for any other purpose without prior written permission from ISO.

Requests for permission to reproduce this document for the purpose of selling it should be addressed as shown below or to ISO’s member body in the country of the requester.

ISO copyright office
Case postale 56, CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Reproduction for sales purposes may be subject to royalty payments or a licensing agreement.

Violators may be prosecuted.
Contents

Contents iii
List of Tables iv
List of Figures v

1 General 1
  1.1 Scope . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
  1.2 Normative references . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
  1.3 Terms and definitions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
  1.4 Implementation compliance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
  1.5 Feature-testing recommendations (Informative) . . . . . . . . . . . . . . . . . . . . . . 2
  1.6 Acknowledgments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

2 Lexical conventions 3
  2.1 Keywords . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

5 Expressions 4
  5.1 Primary expressions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

7 Declarations 11
  7.1 Specifiers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

8 Declarators 18
  8.3 Meaning of declarators . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

10 Derived classes 22
  10.3 Virtual functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

13 Overloading 23
  13.1 Overloadable declarations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
  13.3 Overload resolution . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
  13.4 Address of overloaded function . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

14 Templates 25
  14.1 Template parameters . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
  14.2 Introduction of template parameters . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
  14.3 Names of template specializations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
  14.4 Template arguments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
  14.6 Template declarations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
  14.7 Name resolution . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
  14.8 Template instantiation and specialization . . . . . . . . . . . . . . . . . . . . . . . . . . 37
  14.9 Function template specializations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
  14.10 Template constraints . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

A Compatibility 50
  A.1 C++ extensions for Concepts and ISO C++ 2014 . . . . . . . . . . . . . . . . . . . . . . . 50
## List of Tables

1. Feature-test macros for concepts .................................................. 2
2. *simple-type-specifiers* and the types they specify ........................ 11
3. Value of folding empty sequences ............................................. 33
List of Figures
1 General [intro]

1.1 Scope [intro.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.

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

3 WG21 paper N4191 defines “fold expressions”, which are used to define constraint expressions resulting from the use of constrained-parameters that declare template parameter packs. This feature is not present in ISO/IEC 14882:2014, but it is present in ISO/IEC 14882:2017. The specification of that feature is included in this document.

1.2 Normative references [intro.refs]

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

1 ISO/IEC 14882:2014, Programming Languages – C++

ISO/IEC 14882:2014 is hereafter called the C++ Standard. The numbering of Clauses, sections, and paragraphs in this document reflects the numbering in the C++ Standard. References to Clauses and sections not appearing in this Technical Specification refer to the original, unmodified text in the C++ Standard.

1.3 Terms and definitions [intro.defs]

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

1.3.1 signature [defns.signature]

<function> name, parameter type list (8.3.5), and enclosing namespace (if any), and any associated constraints (14.10.2)
[Note: Signatures are used as a basis for name mangling and linking. — end note]

1.3.2 signature [defns.signature.templ]

<function template> name, parameter type list (8.3.5), enclosing namespace (if any), return type, and template parameter list, and any associated constraints (14.10.2)
1.3.3 [defns.signature.member]
signature
<class member function> name, parameter type list (8.3.5), class of which the function is a member, cv-qualifiers (if any), and ref-qualifier (if any), and any associated constraints (14.10.2)

1.3.4 [defns.signature.membertempl]
signature
<class member function template> name, parameter type list (8.3.5), class of which the function is a member, cv-qualifiers (if any), ref-qualifier (if any), return type, and template parameter list, and any associated constraints (14.10.2)

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

1.5 Feature-testing recommendations (Informative) [intro.features]
1 For the sake of improved portability between partial implementations of various C++ standards, WG21 (the ISO Technical Committee for the C++ Programming Language) recommends that implementers and programmers follow the guidelines in this section concerning feature-test macros. [Note: WG21’s SD-6 makes similar recommendations for the C++ Standard. — end note]

2 Implementers who provide a new standard feature should define a macro with the recommended name, in the same circumstances under which the feature is available (for example, taking into account relevant command-line options), to indicate the presence of support for that feature. Implementers should define that macro with the value specified in the most recent version of this technical specification that they have implemented. The recommended macro name is __cpp_experimental__ followed by the string in the “Macro name suffix” column in Table 1.

3 No header files should be required to test macros describing the presence of support for language features.

Table 1 — Feature-test macros for concepts

<table>
<thead>
<tr>
<th>Macro name suffix</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>concepts</td>
<td>201501</td>
</tr>
</tbody>
</table>

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

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

2.1 Keywords

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

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

In some contexts, unevaluated operands appear (5.1.4, 5.2.8, 5.3.3, 5.3.7).

5.1 Primary expressions

5.1.1 General

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

primary-expression:
fold-expression
requires-expression

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

nested-name-specifier:

auto ::
constrained-type-name ::

Add a new paragraph after paragraph 11:

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

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

auto::* p1 = &S1::n; // auto deduced as S1

template<typename T> concept bool C = sizeof(T) == sizeof(int);
C::* p2 = &S1::n; // OK: C deduced as S1
C::* p3 = &S1::c; // error: deduction fails because constraints are not satisfied

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

template<int N> concept bool D = true;
D::* p4 = &S1::n; // error: D does not designate a placeholder type

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

Add a new paragraph after paragraph 13:
A program that refers explicitly or implicitly to a function with associated constraints that are not satisfied (14.10), other than to declare it, is ill-formed. [Example:

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

In each case the associated constraints of `f` are not satisfied. In the declaration of `p2`, those constraints are required to be satisfied even though `f` is an unevaluated operand (Clause 5). — end example]

5.1.2 Lambda expressions

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

A generic lambda is a lambda-expression where one or more placeholders (7.1.6.4) appear in the parameter-type-list of the lambda-declarator.

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

The closure type for a non-generic lambda-expression has a public inline function call operator (13.5.4) whose parameters and return type are described by the lambda-expression's parameter-declaration-clause and trailing-return-type, respectively. For a generic lambda, the closure type has a public inline function call operator member template (14.5.2) whose template-parameter-list consists of one invented type template parameter for each occurrence of auto in the lambda's parameter-declaration-clause, in order of appearance. The invented type template-parameter is a parameter pack if the corresponding parameter declaration declares a function parameter pack (8.3.5). The return type and function parameters of the function-call operator template are derived from the lambda-expression’s trailing-return-type and parameter-declaration-clause by replacing each occurrence of auto in the decl specifiers of the parameter declaration clause with the name of the corresponding invented template parameter. — The closure type for a generic lambda has a public inline function call operator template that is an abbreviated function template whose parameters and return type are derived from the lambda-expression's parameter-declaration-clause and trailing-return-type according to the rules in (8.3.5).

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

[Example:

```c
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 { return a = *b; }
} fun;
```

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

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

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

fold-operator: one of
   - +  -  *  /  %  &  |  <<  >>
   - += -= *= /= %= ^= &= |= <<= >>= =
   - == !== < > <= >= && || , .* ->*

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

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

Example:

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

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

—end example

5.1.4 Requires expressions

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

requirement-parameter-list:
  ( parameter-declaration-clause_opt )

requirement-body:
  { requirement-seq }

requirement-seq:
  requirement
  requirement-seq requirement

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

A requires-expression defines a constraint (14.10) based on its parameters (if any) and its nested requirements.

A requires-expression has type bool and is an unevaluated expression (5). [Note: A requires-expression is transformed into a constraint (14.10.2) to in order determine if it is satisfied (14.10).—end note]

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

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

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

```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] [Note: Such requirements can also be written by defining them within a concept.

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

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

—end note]

A requires-expression may introduce local parameters using a parameter-declaration-clause (8.3.5). A local parameter of a requires-expression shall not have a default argument. Each name introduced by a local parameter is in scope from the point of its declaration until the closing brace of the requirement-body. These parameters have no linkage, storage, or lifetime; they are only
used as notation for the purpose of defining requirements. The parameter-declaration-clause of a
requirement-parameter-list shall not terminate with an ellipsis. [Example:

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

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

The requirement-body is comprised of a sequence of requirements. These requirements may refer
to local parameters, template parameters, and any other declarations visible from the enclosing
context. Each requirement appends a constraint (14.10) to the conjunction of constraints defined
by the requires-expression. Constraints are appended in the order in which they are written.

The substitution of template arguments into a requires-expression may result in the formation
of invalid types or expressions in its requirements. In such cases, the constraints corresponding to
those requirements are not satisfied; it does not cause the program to be ill-formed. [Note: But
if the substitution of template arguments into a requirement would always result in a substitution
failure, the program is ill-formed; no diagnostic required (14.7). — end note] [Example:

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

5.1.4.1 Simple requirements

```
simple-requirement: expression ;
```

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

[Example:

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

5.1.4.2 Type requirements

```
type-requirement:
    typename nested-name-specifier opt type-name ;
```

A type-requirement introduces a type constraint (14.10.1.4) for the type named by its optional
nested-name-specifier and type-name. [Note: A type requirement asserts the validity of an

associated type, either as a member type, a class template specialization, or an alias template. It is not used to specify requirements for arbitrary type-specifiers. —end note] [Example:

```cpp
template<typename T> struct S { }

template<typename T> using Ref = T&;

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

—end example]

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

A _compound-requirement_ introduces a conjunction of one or more constraints for the _expression_ \( E \). The order in which those constraints are introduced is:

1. The _compound-requirement_ introduces an expression constraint for \( E \) (14.10.1.3);
2. If the _noexcept_ specifier is present, the _compound-requirement_ appends an exception constraint for \( E \) (14.10.1.7);
3. If the _trailing-return-type_ is present, the _compound-requirement_ appends one or more constraints derived from the type \( T \) named by the _trailing-return-type_:
   1. If \( T \) contains one or more placeholders (7.1.6.4), the requirement appends a deduction constraint (14.10.1.6) of \( E \) against the type \( T \).
   2. Otherwise, the requirement appends two constraints: a type constraint on the formation of \( T \) (14.10.1.4) and an implicit conversion constraint from \( E \) to \( T \) (14.10.1.5).

[Example:

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

The _compound-requirement_ in \( C1 \) introduces an expression constraint for \( x++ \). It is equivalent to a _simple-requirement_ with the same _expression_.

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

The _compound-requirement_ in \( C2 \) introduces three constraints: an expression constraint for \( *x \), a type constraint for _typename_ \( T::inner \), and a conversion constraint requiring \( *x \) to be implicitly convertible to _typename_ \( T::inner \).

```cpp
template<typename T> concept bool C3 =
    requires(T x) {
    {g(x)} noexcept;
    };
```
The \textit{compound-requirement} in C3 introduces two constraints: an expression constraint for \( g(x) \) and an exception constraint for \( g(x) \).

\begin{verbatim}
    template<typename T> concept bool C() { return true; }

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

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

\begin{verbatim}
    void g(const C&);
\end{verbatim}

--- end example

\subsection*{5.1.4.4 Nested requirements \[expr.prim.req.nested\]}

\begin{verbatim}
    nested-requirement:
        requires-clause ;
\end{verbatim}

A \textit{nested-requirement} can be used to specify additional constraints in terms of local parameters. A \textit{nested-requirement} appends a predicate constraint (14.10.1.2) for its \textit{constraint-expression} to the conjunction of constraints introduced by its enclosing \textit{requires-expression}. [\textit{Example:}

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

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

The \textit{nested-requirement} appends the predicate constraint \texttt{sizeof(decltype (+t)) == 1} (14.10.1.2). --- end example]
7 Declarations

7.1 Specifiers

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

```
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`.

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

Modify paragraph 2 to begin:

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

Add `constrained-type-specifiers` to the table of `simple-type-specifiers` in Table 10.

<table>
<thead>
<tr>
<th>Specifier(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>constrained-type-specifier</code></td>
<td>placeholder for value (type, non-type, kind) to be deduced</td>
</tr>
</tbody>
</table>

7.1.6.4 auto specifier

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

Replace paragraph 1 with the text below.

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

Modify paragraph two to allow `constrained-type-specifiers` with function declarators, except in the declared return type.

§ 7.1.6.4
The placeholder type `Placeholders` can appear with a function declarator in the `decl-specifier-seq`, `type-specifier-seq`, `conversion-function-id`, or `trailing-return-type`, in any context where such a declarator is valid. If the function declarator includes a `trailing-return-type (8.3.5)`, that specifies the declared return type of the function. If the declared return type of the function contains a placeholder type, the return type of the function is deduced from return statements in the body of the function, if any. The declared return type of a function shall not include a `constrained-type-specifier`. In a function declarator of the form `auto D -> T where T contains placeholders, the initial auto does not designate a placeholder.

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

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

```c
auto glambda = [](int i, auto a) { return i; }; // OK: a generic lambda
```
—end example] Similarly, if a placeholder appears in a parameter type of a function declaration, the function declaration declares an abbreviated function template (8.3.5). [Example:

```c
void f(const auto& k, int); // OK: an abbreviated function template
```
—end example]

Add the following after paragraph 3 to allow the use of `auto` in the `trailing-return-type` of a `compound-requirement`. Also, disallow the use of `decltype(auto)` with function parameters and deduction constraints.

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

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

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

Modify paragraph 4 (paragraph 6, here) to allow multiple placeholders within a variable declaration, but disallowing `constrained-type-specifiers`.

The type of a variable declared using a placeholder `auto` or `decltype(auto)` is deduced from its initializer. This use is allowed when declaring variables in a block (6.3), in namespace scope (3.3.6), and in a `for-init-statement (6.5.3)`. `auto` or `decltype(auto)` shall appear as one of the `decl-specifiers` in the `decl-specifier-seq`. A placeholder can appear anywhere in the declared type of the variable, but `decltype(auto)` shall appear only as one of the `decl-specifiers` of the `decl-specifier-seq`. A `constrained-type-specifier` shall not appear in the declared type of a variable, and the `decl-specifier-seq` of such a variable shall be followed by one or more `init-declarators`, each of which shall have a non-empty initializer. In an initializer of the form

```c
(expression-list )
```
the `expression-list` shall be a single `assignment-expression`. [Example:
auto x = 5;          // OK: x has type int
const auto *v = &x, u = 6; // OK: v has type const int*, u has type const int
static auto y = 0.0;    // OK: y has type double
auto int r;            // error: auto is not a storage-class-specifier
auto f() -> int;       // OK: f returns int
auto g() { return 0.0; } // OK: g returns double
auto h();              // OK: h’s return type will be deduced when it is defined

— end example

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

struct N {
    template<typename T> struct Wrap;
    template<typename T> static Wrap<T> make_wrap(T);
};

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

template<int N> struct Size { void f(int) { } };

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

template<typename T> concept bool C = true;
C z = 0;     // error: constrained-type-specifier in declaration of z
auto cf() -> C; // error: constrained-type-specifier declared in return type of cf

Update paragraph 5 (paragraph 7, here) to disallow constrained-type-specifiers in other statements and expressions.

7 A placeholder type can also be used in declaring a variable in the condition of a selection statement (6.4) or an iteration statement (6.5), in the type-specifier-seq in the new-type-id or type-id of a new-expression (5.3.4), in a for-range-declaration, and in declaring a static data member with a brace-or-equal-initializer that appears within the member-specification of a class definition (9.4.2). A constrained-type-specifier shall not appear in those contexts.

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

Factor the deduction rules for auto into a new subsection.

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

Otherwise, obtain P from T as follows:
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;

otherwise, replace each occurrence of `auto` in the variable type with a new invented type template parameter according to the rules for inventing template parameters for placeholders in 8.3.5, or

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

\[
\begin{align*}
\text{auto } x1 &= \{ 1, 2 \}; & \text{OK: } & \text{decltype(x1) is std::initializer_list<int>} \\
\text{auto } x2 &= \{ 1, 2.0 \}; & \text{error: } & \text{cannot deduce element type}
\end{align*}
\]

— end example]

Add the following to the first example in paragraph 7 in the C++ Standard.

[Example:

\[
\begin{align*}
\text{template<typename T> struct Vec { }}; \\
\text{template<typename T> Vec<T> make_vec(std::initializer_list<T>) } & \{ \text{return Vec<T>{};} \} \\
\text{template<typename... Ts> struct Tuple { }}; \\
\text{template<typename... Ts> auto make_tup(Ts... args) } & \{ \text{return Tuple<Ts...>{};} \} \\
\text{auto } x3 &= *x1.begin(); & \text{OK: } & \text{decltype(x3) is int} \\
\text{const auto* } p &= &x3; & \text{OK: } & \text{decltype(p) is const int} \\
\text{Vec<auto> v1 } &= \text{ make_vec(\{1, 2, 3\})}; & \text{OK: } & \text{decltype(v1) is Vec<int>} \\
\text{Vec<auto> v2 } &= \text{ \{1, 2, 3\}; } & \text{error: type deduction fails} \\
\text{Tuple<auto...> v3 } &= \text{ make_tup(0, 'a'); } & \text{OK: } & \text{decltype(v3) is Tuple<int, char}}
\end{align*}
\]

— end example]

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

[Example:

\[
\begin{align*}
\text{template<typename F, typename S> struct Pair; } \\
\text{template<typename T, typename U> Pair<T, U> make_pair(T, U);} \\
\text{struct S } & \{ \text{void mfn(bool);} \} \ s; \\
\text{int fn(char, double);} \\
\text{Pair<auto (*)(auto, auto), auto (auto::*)(auto)> p = make_pair(fn, &S::mfn);} \\
\end{align*}
\]

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

\[
\begin{align*}
\text{template<class T1, class T2, class T3, class T4, class T5, class T6> } & \text{void } g(Pair< T1(*)(T2, T3), T4 (T5::*)(T6)> x); \\
\end{align*}
\]

— end example]

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

\[ \text{§ 7.1.6.4.1} \]
An explicit instantiation declaration (14.8.2) does not cause the instantiation of an entity declared using a placeholder type, but it also does not prevent that entity from being instantiated as needed to determine its type.

7.1.6.4.2 Constrained type specifiers

Add this section to 7.1.6.4.

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

constrained-type-specifier:
  qualified-concept-name

qualified-concept-name:
  nested-name-specifier
    constrained-type-name

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

concept-name:
  identifier

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

[Example:

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

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

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

  — end example]

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

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

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

  — end example]

A partial-concept-id is a concept-name followed by a sequence of template-arguments. [Example:
template<typename T, int N = 0> concept bool Seq = true;

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

—end example]  

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

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

7.1.7 concept specifier [dcl.spec.concept]

Add this section to 7.1.

The concept specifier shall be applied only to the definition of a function template or variable template, declared in namespace scope (3.3.6). A function template definition having the concept specifier is called a function concept. A function concept shall have no exception-specification and is treated as if it were specified with noexcept(true) (15.4). When a function is declared to be a concept, it shall be the only declaration of that function. A variable template definition having the concept specifier is called a variable concept. A concept definition refers to either a function concept and its definition or a variable concept and its initializer. [Example:

```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>
constexpr bool F3();                // error: function concept is not a definition

template<typename T>
concept bool F3() { return true; } // error: redeclaration of a function as a concept

template<typename T>
concept bool V1 = true;              // OK: declares a variable concept

template<typename T>
concept bool V2;                     // error: variable concept with no initializer

struct S {
  template<typename T>
  static concept bool C = true;       // error: concept declared in class scope
};

—end example]

Every concept definition is implicitly defined to be a constexpr declaration (7.1.5). A concept definition shall not be declared with the thread_local, inline, friend, or constexpr specifiers, nor shall a concept definition have associated constraints (14.10.2).

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

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

template<typename T>
concept bool V = V<T*>;                               // error
```

§ 7.1.7
The first declared template parameter of a concept definition is its \textit{prototype parameter}. A \textit{variadic concept} is a concept whose prototype parameter is a template parameter pack.

A function concept has the following restrictions:

\begin{enumerate}
\item No \textit{function-specifiers} shall appear in its declaration (7.1.2).
\item The declared return type shall have the type \texttt{bool}.
\item The declaration’s parameter list shall be equivalent to an empty parameter list.
\item The declaration shall have a \textit{function-body} equivalent to \{ \texttt{return E;} \} where \( E \) is a \textit{constraint-expression} (14.10.1.3).
\end{enumerate}

[\textit{Note:} Return type deduction requires the instantiation of the function definition, but concept definitions are not instantiated; they are normalized (14.10.2).  \textit{— end note}] [\textit{Example:}
\begin{verbatim}
template<typename T>
concept int F1() { return 0; } // error: return type is not bool
template<typename T>
concept auto F2() { return true; } // error: return type is deduced
template<typename T>
concept bool F3(T) { return true; } // error: not an empty parameter list
\end{verbatim}
\textit{— end example}\]

A variable concept has the following restrictions:

\begin{enumerate}
\item The declared type shall have the type \texttt{bool}.
\item The declaration shall have an initializer.
\item The initializer shall be a \textit{constraint-expression}.
\end{enumerate}

[\textit{Example:}
\begin{verbatim}
template<typename T>
concept bool V1 = 3 + 4; // error: initializer is not a constraint-expression
concept bool V2 = 0; // error: not a template

template<typename T> concept bool C = true;

template<C T>
concept bool V3 = true; // error: constrained template declared as a concept
\end{verbatim}
\textit{— end example}\]

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

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

```plaintext
declarator:
  ptr-declarator
  noptr-declarator parameters-and-qualifiers trailing-return-type requires-clause_opt
  parameters-and-qualifiers:
    ( parameter-declaration-clause ) cv-qualifier-seq_opt
    ref-qualifier_opt exception-specification_opt attribute-specifier-seq_opt requires-clause_opt
```

Add the following paragraphs at the end of this section.

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

```plaintext
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 (*)(*)() requires true; // error: constraint on a variable
void g(int (*)(()) requires true); // error: constraint on a parameter-declaration
auto* p = new void(*)(char) requires true; // error: not a function declaration
— end example]
```

8.3 Meaning of declarators

8.3.5 Functions

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

```plaintext
D1 ( parameter-declaration-clause ) cv-qualifier-seq_opt
    ref-qualifier_opt exception-specification_opt attribute-specifier-seq_opt requires-clause_opt
```

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

```plaintext
D1 ( parameter-declaration-clause ) cv-qualifier-seq_opt
    ref-qualifier_opt exception-specification_opt attribute-specifier-seq_opt trailing-return-type requires-clause_opt
```

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

5. A single name can be used for several different functions in a single scope; this is function overloading (Clause 13). All declarations for a function shall agree exactly in both the return type, and the parameter-type-list, and associated constraints, if any (14.10.2).

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

6. The return type, the parameter-type-list, the `ref-qualifier`, and the `cv-qualifier-seq`, but not the default arguments (8.3.6), associated constraints (14.10.2), or the exception specification (15.4), are part of the function type.
Modify paragraph 15. Note that the footnote reference has been omitted.

15 There is a syntactic ambiguity when an ellipsis occurs at the end of a parameter-declaration-clause without a preceding comma. In this case, the ellipsis is parsed as part of the abstract-declarator if the type of the parameter either names a template parameter pack that has not been expanded or contains auto a placeholder (7.1.6.4); otherwise, it is parsed as part of the parameter-declaration-clause.

Add the following paragraphs after paragraph 15.

16 An abbreviated function template is a function declaration whose parameter-type-list includes one or more placeholders (7.1.6.4). An abbreviated function template is equivalent to a function template (14.6.6) whose template-parameter-list includes one invented template-parameter for each occurrence of a placeholder in the parameter-declaration-clause, in order of appearance, according to the rules below.

17 Each template parameter is invented as follows.

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

(17.2) — Otherwise, the placeholder is designated by a constrained-type-specifier, and the corresponding invented parameter is a constrained-parameter (14.1) whose nested-name-specifier and constrained-type-name are those of the constrained-type-specifier.

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

[ Note: Template parameters are also invented to deduce the type of a variable whose declared type contains placeholders (7.1.6.4.1). — end note ]

The adjusted function parameters of an abbreviated function template are derived from the parameter-declaration-clause by replacing each occurrence of a placeholder with the name of the corresponding invented template-parameter. If the replacement of a placeholder with the name of a template parameter results in an invalid parameter declaration, the program is ill-formed.

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

Example:

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

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

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

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

§ 8.3.5
void g1(const C1*, C2&);
void g2(Vec<C1>&);
void g3(C1...);
void g4(Vec<C3<int>>);
void g5(C4...);
void g6(Tuple<C4...>);  
void g7(C4 p);
void g8(Tuple<C4>);

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

— end example  

Example:

  template<int N> concept bool Num = true;

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

The equivalent declaration would have this form:

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

— end example  

19

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

[Example:

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

  template<int N> void f(Array<auto, N>*);
  template<int N, typename T> void f(Array<T, N>*);  // OK: redeclaration of f(Array<auto, N>*)

— end example  

20

Two constrained-type-specifiers are said to be equivalent if the constraints associated by their corresponding invented constrained-parameters are equivalent according to the rules for comparing constraints described in 14.10.1.

21

All placeholders introduced by equivalent constrained-type-specifiers have the same invented template parameter. [Example:

  namespace N {
    template<typename T> concept bool C = true;
  }

  template<typename T> concept bool C = true;
  template<typename T, int> concept bool D = true;
  template<typename, int = 0> concept bool E = true;

  void f0(C a, C b);

The types of a and b are the same invented template type parameter.
void f1(C& a, C* b);

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

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

In both functions, the parameters \( a \) and \( b \) have different invented template type parameters.

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

The types of \( a \), \( b \), and \( c \) are the same because the constrained-type-specifiers \( E \), \( E<> \), and \( E<0> \) all associate the predicate constraint \( E<T, 0> \), where \( T \) is an invented template type parameter.

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

The types of \( \text{head} \) and \( \text{tail} \) have different types. Their respective associated constraints are \( C<T> \) and \( C<\text{U}>... \), where \( T \) is the template parameter invented for \( \text{head} \) and \( \text{U} \) is the template parameter invented for \( \text{U} \) (14.1).
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.

If a virtual function has associated constraints (14.10.2), the program is ill-formed. [Example:

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

— end example]
13 Overloading

Modify paragraph 1 to allow overloading based on constraints.

When two or more different declarations are specified for a single name in the same scope, that name is said to be overloaded. By extension, two declarations in the same scope that declare the same name but with different types or different associated constraints (14.10.2) are called overloaded declarations. Only function and function template declarations can be overloaded; variable and type declarations cannot be overloaded.

13.1 Overloadable declarations

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

13.1.1 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.

Two function declarations of the same name refer to the same function if they are in the same scope and have equivalent parameter declarations (13.1) and equivalent associated constraints, if any (14.10.2).

13.3 Overload resolution

13.3.2 Viable functions

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

From the set of candidate functions constructed for a given context (13.3.1), a set of viable functions is chosen, from which the best function will be selected by comparing argument conversion sequences and associated constraints for the best fit (13.3.3). The selection of viable functions considers associated constraints, if any (14.10.2), and relationships between arguments and function parameters other than the ranking of conversion sequences.

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

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

Second Third, for F to be a viable function...
13.3.3 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 C++ Standard are elided in this document.

— …
— F1 and F2 are function template specializations, and the function template for F1 is more specialized than the template for F2 according to the partial ordering rules described in 14.6.6.2, or, if not that,
— F1 and F2 are non-template functions with the same parameter-type-lists, and F1 is more constrained than F2 according to the partial ordering of constraints described in 14.10.3.

13.4 Address of overloaded function

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

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

Add the following example at the end of paragraph 5.

[Example:

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

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

— end example]

§ 13.4
14 Templates [temp]

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_opt 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 optional requires-clause following a template-parameter-list allows the specification of constraints (14.10.2) on template arguments (14.4).

14.1 Template parameters [temp.param]

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

```
template-parameter:
  constrained-parameter

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

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

Insert a new paragraph after paragraph 1.

There is an ambiguity in the syntax of a template parameter between the declaration of a constrained-parameter and a parameter-declaration. If the type-specifier-seq of a parameter-declaration is a constrained-type-specifier (7.1.6.4.2), then the template-parameter is a constrained-parameter.

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

A constrained-parameter declares a template parameter whose kind (type, non-type, template) and type match that of the prototype parameter of the concept designated by the qualified-concept-name (7.1.6.4.2) in the constrained-parameter. The designated concept is selected by the rules for concept resolution described in 14.10.4. Let X be the prototype parameter of the designated concept. The declared template parameter is determined by the kind of X (type, non-type, template) and the optional ellipsis in the constrained-parameter as follows.

(9.1) — If X is a type template-parameter, the declared parameter is a type template-parameter.
— If $X$ is a non-type template-parameter, the declared parameter is a non-type template-parameter having the same type as $X$.

— 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 (14.6.3).

Example:

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

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

A constrained-parameter associates a predicate constraint (14.10.1.2) with its template-declaration. The constraint is derived from the qualified-concept-name $Q$ in the constrained-parameter, its designated concept $C$, and the declared template parameter $P$.

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

2. Then, form a template-id $TT$ based on the qualified-concept-name $Q$. If $Q$ is a concept-name, then $TT$ is $C<A>$. Otherwise, $Q$ is a partial-concept-id of the form $C<A1, A2, ..., AN>$, and $TT$ is $C<A, A1, A2, ..., AN>$.

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

4. Finally, If $P$ declares a template parameter pack and $C$ is not a variadic concept, $E$ is adjusted to be the fold-expression $E && ...$ (5.1.3).

$E$ is the expression of the associated predicate constraint. [Example:

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

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

— end example]

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

§ 14.1
The default template-argument of a constrained-parameter shall match the kind (type, non-type, template) of the declared template parameter. [Example:

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

```
template<typename T> struct S0;
```

```
template<C1 T = int> struct S1; // OK
```

```
template<C2 N = 0> struct S2; // OK
```

```
template<C3 X = S0> struct S3; // OK
```

```
template<C1 T = 0> struct S4; // error: default argument is not a type
```

— end example]

14.2 Introduction of template parameters [temp.intro]

Add this section after 14.1.

A template-introduction provides a concise way of declaring templates.

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

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

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

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

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

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

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

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

(2.4) — If I includes an ellipsis, then the declared template parameter is a template parameter pack.

[Example:

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

```
C1{A, B, ...C}; // OK: A is declared as typename A,
struct S1;     // B is declared as int B, and
```
// C is declared as typename ...
C
C2{T} void f(); // OK: T is declared as template<typename> class T
C2{Ts} void g(); // error: the template parameter corresponding to Ts
// is not a template parameter pack
C3{T} struct S2; // OK: T is declared as typename T
C3{Ts} struct S2; // OK: Ts is declared as typename ...

— end example

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

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

— end example

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

template<typename T, int N = -1> concept bool P() { return true; }
P{T, N} struct Array { }; Array<double, 0> s1; // OK
Array<double> s2; // error: Array takes two template arguments

— end example

5 A template-introduction associates a predicate constraint with its template-declaration. This constraint is derived from the qualified-concept-name C in the template-introduction and the sequence of introduced-parameters.

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

(5.2) — Then, form an expression E as follows. If C designates a variable concept (7.1.7), then E is the id-expression C<A1, ..., AN>. Otherwise, C designates a function concept and E is the function call C<A1, ..., AN>(). E is the expression of the associated predicate constraint. [Example:

template<typename T, typename U> concept bool C1 = true;
template<typename T, typename U> concept bool C2() { return true; }
template<typename... Ts> concept bool C3 = true;
C1{A, B} struct s1; // associates C1{A, B>
C2{A, B} struct s2; // associates C2<A, B>()
C3{Ts} struct s3; // associates C3<Ts...>
C3{X, ...Y} struct s4; // associates C3<X, Y...>

§ 14.2
A template declared by a template-introduction can also be an abbreviated function template (8.3.5). The invented template parameters introduced by the placeholders in the abbreviated function template are appended to the list of template parameters declared by the template-introduction. [Example:

```cpp
template<typename T> concept bool C1 = true;
C1{int} void f(T, auto);
template<C1 T, typename U> void f(T, U); // OK: redeclaration of f(T, auto)
```

— end example]

14.3 Names of template specializations [temp.names]

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

When the template-name of a simple-template-id names a constrained non-function template or a constrained template template-parameter, but not a member template that is a member of an unknown specialization (14.7), and all template-arguments in the simple-template-id are non-dependent 14.6.2.4, the associated constraints of the constrained template shall be satisfied. (14.10). [Example:

```cpp
template<typename T> concept bool C1 = false;
template<C1 T> struct S1 { };
template<C1 T> using Ptr = T*;
S1<int>* p; // error: constraints not satisfied
Ptr<int> p; // error: constraints not satisfied

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

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

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

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

template<typename T> concept bool C2 = sizeof(T) == 1;
template<C2 T> struct S { };

template struct S<char[2]>; // error: constraints not satisfied
```
In #1, the error is caused by the substitution of `int` into the associated constraints of the template parameter `X`. In #2, no constraints can be checked for `typename T::template MT<char>` because `MT` is a member of an unknown specialization. — end example

### 14.4 Template arguments

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 C++ Standard 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 at least as constrained as `A` according to the rules in 14.10.3. Two template parameters match if they are of the same kind (type, non-type, template), for non-type `template-parameters`, their types are equivalent (14.6.6.1), and for template `template-parameters`, each of their corresponding `template-parameters` matches, recursively. When `P`’s `template-parameter-list` contains a template parameter pack (14.6.3), the template parameter pack will match zero or more template parameters or template parameter packs in the `template-parameter-list` of `A` with the same kind (type, non-type, template) and type as the template parameter pack in `P` (ignoring whether those template parameters are template parameter packs).

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

**Example:**

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

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

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

S<X> s1; // OK: X and P have equivalent constraints
S[Y> s2; // error: P is not at least as constrained as Y (Y is more constrained than P)
S[Z> s3; // OK: P is at least as constrained as Z

— end example
```

### 14.6 Template declarations

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

For purposes of name lookup and instantiation, default arguments, associated constraints (14.10.2), and exception-specifications of function templates and default arguments, associated constraints, and exception-specifications of member functions of class templates are considered definitions; each default argument, associated constraint, or exception-specification is a separate definition which is unrelated to the function template definition or to any other default arguments, associated constraints, or exception-specifications.
14.6.1 Class templates

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

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 (14.10.2) are those of the class template. The names of the template parameters used in the definition of the member may be different from the template parameter names used in the class template definition. The template argument list following the class template name in the member definition shall name the parameters in the same order as the one used in the template parameter list of the member. Each template parameter pack shall be expanded with an ellipsis in the template argument list.

Add the following example at the end of paragraph 3.

[Example:

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

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

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

template<C T> D{U} struct S<T>::Inner { };  // OK
@end example]

14.6.1.1 Member functions of class templates

Add the following example to the end of paragraph 1.

[Example:

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

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

14.6.2 Member templates

Modify paragraph 1 in order to account for constrained member templates of (possibly) constrained class templates.
A template can be declared within a class or class template; such a template is called a member template. A member template can be defined within or outside its class definition or class template definition. A member template of a class template that is defined outside of its class template definition shall be specified with the template-parameters and associated constraints (14.10.2) of the class template followed by the template-parameters and associated constraints of the member template.

Add the following example at the end of paragraph 1.

```cpp
[Example:

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

template<C1 T>
struct S {
    template<C2 U> void f(U);
    template<C2 U> void g(U);
};

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

— end example]
```

**14.6.3 Variadic templates**

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

— ...

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

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

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

Add the following paragraphs at the end of this section.

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

(9.1) \((E_1 \ op \ E_2 \ op \cdots \ op \ E_N)\) for a unary left fold,
(9.2) \(E_1 \ op (\cdots (E_{N-1} \ op \ E_N))\) for a unary right fold,
(9.3) \(((E \ op \ E_1) \ op \ E_2) \ op \cdots \ op \ E_N)\) for a binary left fold, and
(9.4) \(E_1 \ op (\cdots (E_{N-1} \ op \ (E_N \ op \ E))\)) for a binary right fold.

In each case, \(\text{op}\) is the *fold-operator*, \(N\) is the number of elements in the pack expansion parameters, and each \(E_i\) is generated by instantiating the pattern and replacing each pack expansion parameter with its \(i\)th element. For a binary *fold-expression*, \(E\) is generated by instantiating the *cast-expression* that did not contain an unexpanded parameter pack. [Example:
template<typename ...Args>
bool all(Args ...args) { return (... && args); }

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

Within the instantiation of `all`, the returned expression expands to `((true && true) && true) && false`, which evaluates to `false`. —end example

If \( N \) is zero for a unary fold-expression, the value of the expression is shown in Table 3; if the operator is not listed in Table 3, the instantiation is ill-formed.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Value when parameter pack is empty</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>+</td>
<td>int()</td>
</tr>
<tr>
<td>&amp;</td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>void()</td>
</tr>
</tbody>
</table>

14.6.4 Friends

Modify paragraph 9 to restrict constrained friend declarations.

When a friend declaration refers to a specialization of a function template, the function parameter declarations shall not include default arguments, the declaration shall not have associated constraints (14.10.2), nor shall the inline specifier be used in such a declaration.

Add examples following that paragraph.

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

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

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

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

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

§ 14.6.4
The friend declaration of g2 is ill-formed, no diagnostic required, because no valid specialization of S can be generated: the constraint on g2 can never be satisfied, so template argument deduction (14.9.2.6) will always fail. —end example]

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

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

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

—end example] In the instantiation of such a class template (14.8), the template arguments are substituted into the constraints but not evaluated. Constraints are checked (14.10) only when that function is considered as a viable candidate for overload resolution (13.3.2). If substitution fails, the program is ill-formed. —end note]

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

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

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

```
template<typename T> concept bool C = requires (T t) { t.f(); }
```
```
template<int I> concept bool N = I > 0;
```
```
template<C T1, C T2, N I> class A<T1, T2, I>;
```
```
// #6
```
```
template<C T, N I> class A<int, T*, I>;
```
```
// #7
```
```
—end example]
```

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

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

(8.1) — ...
(8.2) — The argument list of the specialization shall not be identical to the implicit argument list of the primary template.
(8.3) — The specialization shall be more specialized than the primary template (14.6.5.2).
(8.4) — ...

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

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

```
2 A partial specialization matches a given actual template argument list if the template arguments of the partial specialization can be deduced from the actual template argument list (14.9.2) and the deduced template arguments satisfy the constraints of the partial specialization, if any (14.10).
```

Add the following example to the end of paragraph 2.

[Example:
struct S { void f(); ;
A<S, S, 1> a6; // uses #6
A<int, S*, 3> a8; // uses #7
— end example]

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

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

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

(1.1) — the first function template has the same template parameters and associated constraints (14.10.2) as the first partial specialization, and has a single function parameter whose type is a class template specialization with the template arguments of the first partial specialization, and

(1.2) — the second function template has the same template parameters and associated constraints (14.10.2) as the second partial specialization, and has a single function parameter whose type is a class template specialization with the template arguments of the second partial specialization.

Add the following example to the end of paragraph 1.

[Example:

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

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

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

14.6.6 Function templates [temp.fct]

14.6.6.1 Function template overloading [temp.over.link]

Modify paragraph 6 to account for constraints on function templates.

Two function templates are equivalent if they are declared in the same scope, have the same name, have identical template parameter lists, and have return types and parameter lists that are equivalent using the rules described above to compare expressions involving template parameters.

Two function templates are equivalent if they:

(6.1) — are declared in the same scope,
(6.2) — have the same name,
(6.3) — have identical template parameter lists,
(6.4) — have return types and parameter lists that are equivalent using the rules described above to compare expressions involving template parameters, and
(6.5) — have associated constraints that are equivalent using the rules described in 14.10.2 to compare constraints.

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.

(6.6) — 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
(6.7) — the associated constraints are functionally equivalent using the rules described in 14.10.2 to compare constraints.

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

14.6.6.2 Partial ordering of function templates

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 14.10.3.

14.7 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 instantiation of the associated constraints (14.10.2) of a template would result in a valid expression, the template is ill-formed, no diagnostic required. If a hypothetical instantiation of a template immediately following its definition would be ill-formed due to a construct that does not depend on a template parameter, the program is ill-formed; no diagnostic is required. If the interpretation of such a construct in the hypothetical instantiation is different from the interpretation of the corresponding construct in any actual instantiation of the template, the program is ill-formed; no diagnostic is required.
14.7.2 Dependent names

14.7.2.2 Type-dependent expressions

Add the following paragraph to this section.

A fold-expression is type-dependent.

14.7.2.3 Value-dependent expressions

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

Expressions of the following form are value-dependent:

\texttt{sizeof ... ( identifier )}

fold-expression

14.7.4 Dependent name resolution

14.7.4.1 Point of instantiation

Add a new paragraph after paragraph 4.

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

14.8 Template instantiation and specialization

14.8.1 Implicit Instantiation

Change paragraph 1 to include associated constraints.

Unless a class template specialization has been explicitly instantiated 14.8.2 or explicitly specialized 14.8.3, the class template specialization is implicitly instantiated when the specialization is referenced in a context that requires a completely-defined object type or when the completeness of the class type affects the semantics of the program. [Note: Within a template declaration, a local class or enumeration and the members of a local class are never considered to be entities that can be separately instantiated (this includes their default arguments, exception-specifications, and non-static data member initializers, if any). As a result, the dependent names are looked up, the semantic constraints are checked, and any templates used are instantiated as part of the instantiation of the entity within which the local class or enumeration is declared. —end note]

The implicit instantiation of a class template specialization causes the implicit instantiation of the declarations, but not of the definitions, default arguments, associated constraints (14.10.2), or exception-specifications of the class member functions, member classes, scoped member enumerations, static data members and member templates; and it causes the implicit instantiation of the definitions of unscoped member enumerations and member anonymous unions.

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

The associated constraints of a template specialization are not instantiated along with the specialization itself; they are instantiated only to determine if they are satisfied (14.10). [Note: The satisfaction of constraints is determined during name lookup or overload resolution (13.3). Preserving the spelling of the substituted constraint also allows constrained member function to be partially ordered by those constraints according to the rules in 14.10.3. —end note] [Example:]
```cpp
template<typename T> concept bool C = sizeof(T) > 2;
template<typename T> concept bool D = C<T> && sizeof(T) > 4;

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

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

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

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

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

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

### 14.8.2 Explicit instantiation [temp.explicit]

Add the following note after paragraph 7.

8 [Note: An explicit instantiation of a constrained template shall satisfy that template’s associated constraints (14.10). The satisfaction of constraints is determined during name lookup for explicit instantiations in which all template arguments are specified (14.3), or for explicit instantiations of function templates, during template argument deduction (14.9.2.6) when one or more trailing template arguments are left unspecified. — end note]

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

An explicit instantiation that names a class template specialization is also an explicit instantiation of the same kind (declaration or definition) of each of its members (not including members inherited from base classes and members that are templates) that has not been previously explicitly specialized in the translation unit containing the explicit instantiation, and provided that the associated constraints (14.10.2), if any, of that member are satisfied (14.10) by the template arguments of the explicit instantiation, except as described below.

### 14.8.3 Explicit specialization [temp.expl.spec]

Add the following note after paragraph 10.

11 [Note: An explicit specialization of a constrained template shall satisfy that template’s associated constraints (14.10). The satisfaction of constraints is determined during name lookup for explicit specializations in which all template arguments are specified (14.3), or for explicit specializations

§ 14.8.3 38
of function templates, during template argument deduction (14.9.2.6) when one or more trailing template arguments are left unspecified. — end note]

14.9 Function template specializations [temp.fct.spec]

14.9.2 Template argument deduction [temp.deduct]

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

When all template arguments have been deduced or obtained from default template arguments, all uses of template parameters in the template parameter list of the template and the function type are replaced with the corresponding deduced or default argument values. If the substitution results in an invalid type, as described above, type deduction fails. If the function template has associated constraints (14.10.2), the template arguments are substituted into the associated constraints without evaluating the resulting expression. If this substitution results in an invalid type or expression, type deduction fails. [Note: The satisfaction of constraints (14.10) associated with the function template specialization is determined during overload resolution (13.3), and not at the point of substitution. — end note]

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

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

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

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

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

Add the following example to paragraph 3.

[Example:

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

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

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

— end example]

14.10 Template constraints [temp.constr]

Add this section after 14.9.
1 [Note: This section defines the meaning of constraints on template arguments. The abstract syntax, satisfaction rules, and equivalence rules are defined in 14.10.1. Constraints are associated with declarations in 14.10.2. Declarations are partially ordered by their associated constraints (14.10.3). — end note]

14.10.1 Constraints [temp.constr.constr]

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

There are several different kinds of constraints:

1. Conjunctions (14.10.1.1),
2. Disjunctions (14.10.1.1),
3. Predicate constraints (14.10.1.2),
4. Expression constraints (14.10.1.3),
5. Type constraints (14.10.1.4),
6. Implicit conversion constraints (14.10.1.5),
7. Argument deduction constraints (14.10.1.6),
8. Exception constraints (14.10.1.7), and
9. Parameterized constraints (14.10.1.8)

In order for a constrained template to be instantiated (14.8), its associated constraints (14.10.2) shall be satisfied. [Note: The satisfaction of constraints on class templates, alias templates, and variable templates is required when referring a template specialization (14.3). The satisfaction of constraints on functions and function templates is required during overload resolution (13.3.2). — end note] Determining if a constraint is satisfied entails the substitution of template arguments into that constraint. The rules for determining the satisfaction of different kinds of constraints are defined in the following subsections.

In certain contexts, it is necessary to know when two constraints are equivalent (14.10.2). The rules for determining the equivalence of different kinds of constraints are defined in the following subsections. Two constraints that are not equivalent are functionally equivalent if, for any given set of template arguments, either both constraints are satisfied or both constraints are unsatisfied.

14.10.1.1 Logical operations [temp.constr.op]

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 $P \land Q$, $P$ is the left operand and $Q$ is the right operand. Grouping of constraints is shown using parentheses. — end note]

A conjunction is a constraint taking two operands. A conjunction of constraints is satisfied if and only if both operands are satisfied. The satisfaction of a conjunction’s operands are evaluated left-to-right; if the left operand is not satisfied, template arguments are not substituted into the right operand, and the constraint is not satisfied. If the left and right operands of a conjunction are predicate constraints (14.10.1.2), let $P$ and $Q$ be the expressions of those constraints resulting from substitution. If the expression $P \&\& Q$ results in a call to a user-declared operator$\&\&$, the program is ill-formed. [Example:
template<typename T>
    constexpr bool fail() { return T::value; }

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

void f(int);

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

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

A conjunction P is equivalent to another conjunction Q if and only if the left operands of P and Q are equivalent and the right operands of P and Q are equivalent.

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

A disjunction P is equivalent to another disjunction Q if and only if the left operands of P and Q are equivalent and the right operands of P and Q are equivalent.

[Note: The prohibition against user-declared logical operators disallows constraint-expressions (14.10.2) whose evaluation disagrees with the satisfaction of its derived constraint. That is, for any atomic predicate constraints P and Q, the conjunction P \&\& Q is satisfied if and only if the constraint-expression P \&\& Q evaluates to true. Likewise, the disjunction P \lor Q is satisfied if and only if the constraint-expression P \lor Q evaluates to true. — end note]

14.10.1.2 Predicate constraints

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

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

```cpp
template<typename T>

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

```cpp
template<typename T>
```


requires S<T>{
void f(T);

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

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

A predicate constraint \( P \) is equivalent to another predicate \( Q \) if and only if the expressions of \( P \) and \( Q \) are equivalent using the rules described in 14.6.6.1 to compare expressions.

14.10.1.3 Expression constraints

An expression constraint is a constraint that specifies a requirement on the formation of an expression \( E \) through substitution of template arguments. An expression constraint is satisfied if substitution yielding \( E \) did not fail. Within an expression constraint, \( E \) is an unevaluated operand (Clause 5). [Note: An expression constraint is introduced by the expression in either a simple-requirement (5.1.4.1) or compound-requirement (5.1.4.3) of a requires-expression. — end note] [Example:

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

The concept \( C \) introduces an expression constraint for the expression \( ++t \). The type argument \( \text{int} \) satisfies this constraint because the the expression \( ++t \) is valid after substituting \( \text{int} \) for \( T \). — end example]

An expression constraint \( P \) is equivalent to another expression constraint \( Q \) if and only if the expressions of \( P \) and \( Q \) are equivalent using the rules described in 14.6.6.1 to compare expressions.

14.10.1.4 Type constraints

A type constraint is a constraint that specifies a requirement on the formation of a type \( T \) through the substitution of template arguments. A type constraint is satisfied if and only \( T \) is not ill-formed, meaning that the substitution yielding \( T \) did not fail. [Note: A type constraint is introduced by the typename-specifier in a type-requirement of a requires-expression (5.1.4.2). — end note] [Example:

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

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

A type constraint \( P \) is equivalent to another type constraint \( Q \) if and only if the types in \( P \) and \( Q \) are equivalent according to the rules in 14.4.

14.10.1.5 Implicit conversion constraints

An implicit conversion constraint is a constraint that specifies a requirement on the implicit conversion of an expression \( E \) to a type \( T \). The constraint is satisfied if and only if \( E \) is implicitly convertible to \( T \) (Clause 4). [Note: A conversion constraint is introduced by a trailing-return-type in a compound-requirement when the trailing-return-type contains no placeholders (5.1.4.3). — end note] [Example:
template<typename T> concept bool C =
  requires (T a, T b) {
    { a == b } -> bool;
  };

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

An implicit conversion constraint P is equivalent to another implicit conversion constraint Q if and only if the expressions of P and Q are equivalent using the rules in 14.6.6.1 to compare expressions, and the the types of P and Q are equivalent according to the rules in 14.4.

14.10.1.6 Argument deduction constraints [temp.constr.deduct]

An argument deduction constraint is a constraint that specifies a requirement that the type of an expression E can be deduced from a type T, when T includes one or more placeholders (7.1.6.4).

[Note: An argument deduction constraint is introduced by a compound-requirement (5.1.4.3) having a trailing-return-type that contains one ore more placeholders. In such a constraint, E is the expression of the compound-requirement, and T is the type specified by the trailing-return-type. — end note]

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

[Example:

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

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

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

template<typename T>
void g(T);

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

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

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

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

An argument deduction constraint P is equivalent to another argument deduction constraint Q if and only if the expressions of P and Q are equivalent using the rules in 14.6.6.1 to compare expressions, and the types of P and Q are equivalent (14.4).
14.10.1.7 Exception constraints

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

An exception constraint \( P \) is equivalent to another predicate \( Q \) if and only if the expressions of \( P \) and \( Q \) are equivalent using the rules described in 14.6.6.1 to compare expressions.

14.10.1.8 Parameterized constraints

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

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

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

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

Two parameterized constraints \( P \) and \( Q \) are equivalent if and only if their operands are equivalent. Two expressions involving constraint variables are equivalent if they are equivalent according to the rules for expressions described in 14.6.6.1, except that any identifiers referring to constraint variables are equivalent if and only if the types of their corresponding declarations are equivalent (14.4).

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

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

— end example]

14.10.2 Constrained declarations

A template declaration (Clause 14) or function declaration (8.3.5) can be constrained by the use of a requires-clause. This allows the specification of constraints for that declaration as an expression:
A constraint-expression introduces a predicate constraint (14.10.1.2) for its logical-or-expression (5.15).

Constraints can also be associated through the use of template-introductions, constrained-parameters in a template-parameter-list, or constrained-type-specifiers in the parameter-type-list of a function template. A template’s associated constraints are the conjunction of constraints introduced in its template-declaration. The ordering of operands in that conjunction is:

- a template-introduction (14.2), and
- all constrained-parameters (14.1) in the declaration’s template-parameter-list, in order of appearance, and
- the predicate constraint (14.10.1.2) of a requires-clause following a template-parameter-list (Clause 14), and
- all constrained-type-specifiers (7.1.6.4.2) in the type of a parameter-declaration in a function declaration (8.3.5), in order of appearance, and
- the predicate constraint of a trailing requires-clause (Clause 8) of a function declaration (8.3.5).

The formation of the associated constraints for a template declaration defines the order in which constraints are compared for equivalence (to determine when one template redeclares another), and the order in which template arguments are substituted when checking for satisfaction (14.10.1). A program containing two declarations whose associated constraints are functionally equivalent but not equivalent (14.10.1) is ill-formed, no diagnostic required. [Example:

```c
template<typename T> concept bool C = true;
void f1(C);
template<T> void f1(T);
C{T} void f1(T);
template<typename T> requires C<T> void f1(T);
template<typename T> void f1(T) requires C<T>;
```

All declarations of `f1` declare the same function.

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

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

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

```c
template<C1 T> requires C2<T> void f3();
template<C2 T> requires C1<T> void f3(); // error: constraints are functionally equivalent but not equivalent
```

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

Determining if a declaration’s associated constraints are satisfied and partially ordering declarations by their associated constraints requires the **normalization** of the associated constraints.
Normalization transforms a constraint into a sequence of conjunctions and disjunctions of atomic constraints. An atomic constraint (as defined below) is one that cannot be expressed as a conjunction or disjunction of its operands. The normal form of a constraint is defined as follows:

1. The normal form of the disjunction $P \lor Q$ is the disjunction of the normal form of $P$ and the normal form $Q$.

2. The normal form of the conjunction $P \land Q$ is the conjunction of the normal form of $P$ and the normal form $Q$.

3. The normal form of a predicate constraint $P$ is formed by first creating a new expression $E$ from the expression of $P$ by replacing all subexpressions in $P$ that refer to concepts with their corresponding definitions. In particular,
   - replace all function calls of the form $C<A_1, A_2, \ldots, A_N>()$, where $C<A_1, A_2, \ldots, A_N>$ names the specialization of a function concept $D$ (7.1.7), with the result of substituting $A_1, A_2, \ldots, A_N$ into the expression returned by $D$, and
   - replace all id-expressions of the form $C<A_1, A_2, \ldots, A_N>$, where $C<A_1, A_2, \ldots, A_N>$ names the specialization of a variable concept $D$ (7.1.7), with the result of substituting $A_1, A_2, \ldots, A_N$ into the initializer of $D$.

   If any such substitution fails, the program is ill-formed. Second, transform the expression $E$ into a constraint as follows:
   - An expression $(P)$ is transformed into the normalized predicate constraint $P$.
   - An expression $P || Q$ is transformed into the normalized disjunction of the predicate constraints $P$ and $Q$.
   - An expression $P && Q$ is transformed into the normalized conjunction of the predicate constraints $P$ and $Q$.
   - A requires-expression, (5.1.4) having the form
     \[
     \text{requires ( parameter-declaration-clause ) requirement-body}
     \]
     where the parameter-declaration-clause is not equivalent to an empty parameter list, is transformed into a parameterized constraint (14.10.1.8) with the same parameters as those in the parameter-declaration-clause and whose operand is the normal form of conjunction of constraints introduced by requirements in the requirement-body.
   - A requires-expression having one of the following forms
     \[
     \text{requires ( void ) requirement-body}
     \]
     \[
     \text{requires () requirement-body}
     \]
     \[
     \text{requires requirement-body}
     \]

     is transformed into the normal form of conjunction of constraints introduced by requirements in the requirement-body.
   - Otherwise, the expression $E$ is an atomic predicate constraint.

4. Expression constraints (14.10.1.3), type constraints (14.10.1.4), implicit conversion constraints (14.10.1.5), argument deduction constraints (14.10.1.6), and exception constraints (14.10.1.3) are all atomic constraints and in normal form.

[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; };
template<typename T> concept bool C4 = requires (T x) { ++x; };

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

template<typename T>
    requires (bool)3 + 4 void f4(T); // error: invalid constraints (#4)

The normalized associated constraints of #1 are sizeof(T) == 1 ∧ 1 == 2, those of #2 are
the type constraint for T::type, those of #3 are the parameterized constraint λ(T x) the
expression constraint ++x. Note that the normalized constraints of #2 includes two atomic
constraints: sizeof(char) == 1 and 1 == 2. In #4, the constraint-expression (bool)3 + 4 is
not a valid predicate constraint because it does not have type bool. —end example]
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
template<C1 T> void g(T); // #4

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

— end example —

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

14.10.4 Resolution of qualified-concept-names [temp.constr.resolve]

Concept resolution is the process of selecting a concept from a set of concept definitions referred to by a qualified-concept-name. Concept resolution is performed when a qualified-concept-name appears

1. as a constrained-type-specifier (7.1.6.4.2),
2. in a constrained-parameter (14.1), or
3. in a template-introduction (14.2).

Concept resolution selects a concept from a set referred to by an optional nested-name-specifier and the concept-name of a qualified-concept-name by matching the template parameters of each concept in that set to a sequence of template arguments and wildcards. This sequence is called the concept argument list, and its elements are called concept arguments. For the purpose of this matching, a wildcard can match a template parameter of any kind (type, non-type, template) as described below.

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

1. If C is part of a constrained-type-specifier or constrained-parameter, then
   1. if C is a constrained-type-name, the concept argument list is comprised of a single wildcard, or
   2. if C is the concept-name of a partial-concept-id, the concept argument list is comprised of a single wildcard followed by the template-arguments of that partial-concept-id.
2. If C is the concept-name in a template-introduction, the concept argument list is a sequence of wildcards of the same length as the introduction-list of the template-introduction.
3. If C appears as a template-name of a template-id, the concept argument list is the sequence of template-arguments of the template-id.

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

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

- a wildcard matches a template parameter of any kind;

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

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

```c++
// Example:

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

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

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

Q{...Ts} void q1(); // OK: selects Q
Q<T> void q2(); // OK: selects Q

— end example ]

§ 14.10.4
Annex A  (informative)
Compatibility

A.1  C++ extensions for Concepts and ISO C++ 2014

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

A.1.1  Clause 2: lexical conventions

2.1  Change: New Keywords

New keywords are added to C++ extensions for Concepts; see 2.1.

Rationale: These keywords were added in order to implement the semantics of the new features. In particular, the requires keyword is added to introduce constraints through a requires-clause or a requires-expression. The concept keyword is added to enable the definition of concepts (7.1.7), the normalization of constraints (14.10.2), and the semantic differentiation of concept-names from other identifiers.

Effect on original feature: Change to semantics of well-defined feature. Any ISO C++ programs that used any of these keywords as identifiers are not valid C++ programs with Concepts.

Difficulty of converting: Syntactic transformation. Converting one specific program is easy. Converting a large collection of related programs takes more work.

How widely used: Seldom.