Working Draft, C++ Extensions for Concepts

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

1 General ................................................................. 4
  1.1 Scope ............................................................... 4
  1.2 Normative references ............................................ 4
  1.3 Terms and definitions ............................................ 4
  1.4 Implementation compliance ...................................... 4
  1.5 Acknowledgments ............................................... 4

2 Lexical conventions .................................................. 6
  2.1 Keywords ........................................................ 6

3 Expressions .......................................................... 7
  3.1 Primary expressions .......................................... 7
    3.1.1 Lambda expressions .................................... 7
    3.1.2 Requires expressions .................................. 8
      3.1.2.1 Simple requirements ............................... 9
      3.1.2.2 Type requirements ............................... 9
      3.1.2.3 Nested requirements ............................. 10
      3.1.2.4 Compound requirements ......................... 10

4 Declarations ........................................................ 12
  4.1 Specifiers ...................................................... 12
    4.1.1 Simple type specifiers ................................. 12
    4.1.2 auto specifier .......................................... 12
    4.1.3 Constrained type specifiers .......................... 12
    4.1.4 concept specifier ...................................... 14

5 Declarators .......................................................... 16
  5.1 Meaning of declarators ....................................... 16
    5.1.1 Functions ............................................... 16
  5.2 Function definitions ......................................... 17
    5.2.1 In general ............................................... 17

6 Classes .............................................................. 18
  6.1 Class members ................................................ 18

7 Templates ............................................................. 19
  7.1 Template parameters .......................................... 20
  7.2 Template names ................................................ 22
  7.3 Template arguments .......................................... 22
    7.3.1 Template template arguments .......................... 22
  7.4 Template declarations ........................................ 23
    7.4.1 Class templates ......................................... 23
      7.4.1.1 Member functions of class templates ............ 23
    7.4.2 Member templates ...................................... 24
    7.4.3 Friends .................................................. 24
    7.4.4 Class template partial specialization ................. 25
      7.4.4.1 Matching of class template partial specializations .. 25
      7.4.4.2 Partial ordering of class template specializations ... 25
    7.4.5 Function templates ..................................... 26
      7.4.5.1 Template argument deduction ........................ 26
      7.4.5.2 Function template overloading ........................ 26
      7.4.5.3 Partial ordering of function templates ................ 26
  7.5 Template instantiation and specialization ....................... 27
    7.5.1 Implicit instantiation ................................. 27
7.5.2 Explicit instantiation ......................................................... 27
7.6 Template constraints .......................................................... 27
1 General

1.1 Scope

1 This technical specification describes extensions to the C++ Programming language (1.2) that enable the specification and checking of constraints on template arguments, and the ability to overload functions and specialize templates based on those constraints. These extensions include new syntactic forms and modifications to existing language semantics.

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

1.2 Normative references

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

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

2 ISO/IEC XXXX:2011 is herein after called the C++ Standard. References to clauses within the C++ Standard are written as "C++14 §3.2".

1.3 Terms and definitions

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

   1.3.1 atomic constraint
   A subexpression of a constraint that is not a logical-and-expression, logical-or-expression, or a subexpression of an atomic constraint.

1.4 Implementation compliance

1 Conformance requirements for this specification are the same as those defined in C++14 §1.4. [ Note: Conformance is defined in terms of the behavior of programs. — end note ]

1.5 Acknowledgments

1 The design of this specification is based, in part, on a concept specification of the algorithms part of the C++ standard library, known as "The Palo Alto" TR (WG21 N3351), which was developed by a large group of experts as a test of the expressive power of the idea of concepts.
Despite syntactic differences between the notation of the Palo Alto TR and this TS, the TR can be seen as a large-scale test of the expressiveness of this TS.

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

2.1 Keywords

1 In C++14 §2.12, Table 4, add the keywords concept and requires.
3 Expressions

3.1 Primary expressions

1 In C++14 §5.1.1, add requires-expression to the rule, primary-expression.

    primary-expression:
        requires-expression

3.1.1 Lambda expressions

1 Modify C++14 §5.1.2/5.

2 The closure type for a non-generic lambda-expression has a public inline function call operator (C++14 §13.4) 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 (C++14 §14.5.2) whose template-parameter-list consists of one invented type template-parameter for each occurrence of auto or each unique occurrence of a constrained-type-name 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 (C++14 §8.3.5). If the The invented type template-parameter is a constrained parameter if the decl-specifier-seq of the corresponding parameter-declaration includes a constrained-type-specifier, the invented type parameter is a constrained-parameter, whose constrained-type-specifier matches that of the parameter-declaration. (7.1). 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. [Example: Consider the following:

    template<typename T> concept bool C = ...;
    auto fun = []{const C& arg} { }    

The parameter arg has a constrained-type-specifier C, signifying that it is a generic lambda whose closure type is like the following function object:

    struct Fun {
        template<__T> auto operator()(const __T& x) const { }
    };

    — end example ]

3 Also insert the following paragraph after C++14 §5.1.2/5.

4 All placeholder types introduced using the same concept-name have the same invented template parameter. [Example:

    auto f = [](C a, C b) { };
    f(0, 0); // Ok
    f(0, 'a'); // Error: template argument deduction failure

The second call to f results in a compiler error because the types of the deduced arguments cannot be unified. — end example ]
3.1.2 Requires expressions

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

```
requires-expression:
  requires requirement-parameter-list requirement-body
requirement-parameter-list:
  ( parameter-declaration-clauseopt )
requirement-body:
  { requirement-list }
requirement-list:
  requirement
  requirement-list requirement
requirement:
  simple-requirement
  compound-requirement
  type-requirement
  nested-requirement
simple-requirement:
  expression ;
compound-requirement:
  constexpr opt { expression } noexcept opt trailing-return-typeopt ;
type-requirement:
  typename-specifier ;
nested-requirement:
  requires-clause ;
```

A requires-expression has type bool.

A requires-expression shall not appear outside of a concept definition (4.1.4) or a requires-clause.

[ Example: The most common use of requires-expressions is to define syntactic requirements in concepts (4.1.4) such as the one below:

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

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

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

— end example ]

The requires-expression may introduce local arguments via a parameter-declaration-clause. These parameters have no linkage, storage, or lifetime. They are used only to write constraints within the requirement-body and are not visible outside the closing } of the requirement-body. The requirement-parameter-list shall not include an ellipsis.
The requirement-body is a sequence of requirements separated by semicolons. These requirements may refer to local arguments, template parameters, and any other declarations visible from the enclosing context. Each requirement introduces a conjunction of one or more atomic constraints (7.6). The kinds of atomic constraints introduced by a requirement are:

- A valid expression constraint is a predicate on an expression. The constraint is satisfied if and only if the substitution of template arguments into that expression does not result in substitution failure. The result of successfully substituting template arguments into the dependent expression produces a valid expression.
- A valid type constraint is a predicate on a type. The constraint is satisfied if and only if the substitution of template arguments into that type does not result in substitution failure. The result of successfully substituting template arguments into the dependent type produces an associated type.
- A result type constraint is a predicate on the result type of a valid expression. Let \( E \) be a valid expression and \( X \) be a trailing-return-type. The constraint is satisfied if and only if \( E \) can be used as an argument to an invented function \( f \), which has a single function parameter of type \( X \) and returning void. That is, the function call \( f(E) \) must be a valid expression. [Note: Each template parameter referred to by \( X \) is a template parameter of the invented function \( f \). If \( X \) contains a constrained-type-specifier or auto specifier, then \( f \) is a generic function (5.1.1). — end note]
- A constant expression constraint is satisfied if and only if a valid expression \( E \) is a constant expression (C++14 §5.19).
- An exception constraint is satisfied if and only if, for a valid expression \( E \), the expression \( \text{noexcept}(E) \) evaluates to true (C++14 §5.3.7).

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

### 3.1.2.1 Simple requirements

A simple-requirement introduces a valid expression constraint for its expression. The expression is an unevaluated operand (C++14 §3.2). [Example: The following is requirement evaluates to true for all arithmetic types (C++14 §3.9.1), and false for pointer types (C++14 §3.9.2).

```cpp
requires (T a, T b) {
    a + b;  // A simple requirement
}
— end example ]
```

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

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

### 3.1.2.2 Type requirements

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

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

— end example ]

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

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

— end example ]

3.1.2.3 Nested requirements

A nested-requirement introduces an additional constraint expression 7.6 to be evaluated as part of the satisfaction of the requires-expression. The requirement is satisfied if and only if the constraint evaluates to value true. [Example: Nested requirements are generally used to provide additional constraints on associated types within a requires-expression.

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

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

3.1.2.4 Compound requirements

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

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

A compound-requirement is satisfied if and only if every constraint in the set is satisfied. The required valid expression is an unevaluated operand (C++14 §3.2) except in the case when the constexpr specifier is present. These other requirements are described in the following paragraphs.

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

3 The presence of a trailing-return-type introduces a result type constraint on ε.

4 If the constexpr specifier is present then a constant expression constraint is introduced for the valid expression ε.
If the `noexcept` specifier is present, then an exception constraint is introduced for the valid expression \( E \).

[Example:]

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

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

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

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

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

```cpp
void z2(const Inscrutable&)
```

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

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

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

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

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

4.1 Specifiers

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

4.1.1 Simple type specifiers

1 Extend the simple-type-specifier production to include constrained-type-specifier.
   simple-type-specifier:
   constrained-type-specifier
   constrained-type-specifier:
   nested-name-specifier opt constrained-type-name
   constrained-type-name:
   concept-name
   partial-concept-id
   concept-name:
   identifier
   partial-concept-id:
   concept-name < template-argument-list >

4.1.2 auto specifier

1 Modify C++14 §7.1.6.4/1 so that the last sentence reads:
2 The auto type-specifier is also used to signify that a lambda is a generic lambda or that a function is a generic function.

4.1.3 Constrained type specifiers

1 A constrained-type-specifier designates a placeholder type that will be replaced later by deduction from a required valid expression in a compound-requirement. A constrained-type-specifier is also used to signify that a lambda is a generic lambda or that a function is a generic function.

2 A constrained-type-specifier can appear in the trailing-return-type of a compound-requirement or in any context in which the auto type-specifier appears, except:
   — in the decl-specifier-seq of a variable declaration,
   — in the return type of a function declaration,
   — in the decltype(auto) type-specifier, or
   — a conversion-function-id.

3 If the constrained-type-name appears as one of the decl-specifiers of a parameter-declaration in a template-parameter-list, then the declared parameter is a constrained-parameter, and its meaning is defined in section 7.1. Otherwise, the meaning of constrained-type-specifiers is defined in this section. [ Note: A constrained template parameter can introduce type parameters as well as designate the type of a non-type template parameter. The meaning of those declarations are specified separately. — end note ]
If the constrained-type-specifier appears as one of the decl-specifiers of a parameter-declaration in either a lambda-expression or function declaration then the lambda is a generic lambda 3.1.1 and the function is a generic function 3.1.1.

A constrained-type-specifier designates a placeholder type that will be replaced later, and it introduces an associated constraint on deduced type, called the constrained type within the enclosing declaration or requires-expression.

If the constrained-type-specifier appears in the trailing-return-type of a compound-requirement, then the constrained type is deduced from the required valid expression. Otherwise, the constrained type is deduced using the rules for deducing auto (4.1.2).

The introduced constraint is a constraint expression (7.6) synthesized from the concept-name or partial-concept-id in the constrained-type-name.

When an identifier is a concept-name, it refers to one or more function concepts or a single variable concept. At least one concept referred to by the constrained-type-name shall be a type concept (4.1.4). [Example: Function concepts can be overloaded to accept different numbers and kinds of template arguments. This is sometimes done to generalize a single concept for different kinds of arguments.]

```cpp
template<typename T>
concept bool C() { ... }

template<typename T, typename U>
concept bool C() { ... }
```

— end example ] The concept-name C refers to both concept definitions.

A partial-concept-id is a concept-name followed by a sequence of template arguments. A partial-concept-id does not refer to template specialization; the template argument list must be adjusted by adding a template argument before the first of the initial template arguments before the name refers to a template specialization. [ Example:

```cpp
template<typename T, typename U>
concept bool C = ...;

C<int>       // A partial-concept-id
C<char, int> // A template-id
```

The first name is a partial-concept-id and can be used as part of constrained type name as part the type specifier of a parameter declaration or a template parameter. The second name is a template-id and determines whether the concept is satisfied for the given arguments. — end example ]

A partial-concept-id shall not have an empty list of template arguments.

An introduced constraint is formed by applying the following rules to each concept referenced by the concept-name in the constrained-type-name. Let C be a concept referred to by the concept-name, T be the constrained type, and Args be a sequence of template arguments. If the constrained-type-name is a partial-concept-id, then Args is its template-argument-list, otherwise Args is an empty sequence. The candidate constraint is a template-id having the form C<T, Args>.

[ Note: If Args is empty, the resulting template-id is of the form C<T>. — end note ] If C<T, Args> does not refer to a template specialization, the candidate constraint is rejected. [ Note: The expression C<T, Args> may not refer to a valid template specialization if Args contains too many or to few template arguments for C, or if Args do not match C’s template parameters. — end note ]

If, after constructing candidate constraints for each concept named by the concept-name, there are no candidates or more than one candidate, the program is ill-formed.
The introduced constraint is constructed from the remaining candidate. If \( c \) is a function concept, then the resulting constraint is a function call of the form \( c<T, \text{Args}()> \). Otherwise, the introduced constraint is the same as the remaining candidate.

**Example: The following unary and binary concepts are defined as variables and functions.**

```cpp
template<typename T>
class bool V1 = ...;

template<typename T, typename U>
class bool V2 = ...;

template<typename T>
class bool F1() { return ...; }

template<typename T, typename T2>
class bool F2() { return ...; }
```

Suppose \( X \) is a template parameter being declared, either explicitly or as an invented template parameter of a *parameter-declaration* in a generic function or generic lambda. The synthesized constraints corresponding to each declaration are:

- \( V1 X \)  // becomes \( V1<T> \)
- \( V2<Y> X \)  // becomes \( V2<X, Y> \)
- \( F1 X \)  // becomes \( F1<X>() \)
- \( F2<Y> X \)  // becomes \( F2<X, Y>() \)

— *end example*

The meaning of the introduced constraint depends on the context in which the *constrained-type-specifier* appears. If it appears in the *decl-specifiers* of a *parameter-declaration* of a generic lambda (3.1.1) or generic function (5.1.1), the introduced constraint is associated with the corresponding template declaration (7). If it appears in *trailing-return-type* of a *compound-requirement*, the introduced constraint is evaluated as part of the enclosing *requires-expression* (3.1.2).

### 4.1.4 concept specifier

1. The *concept specifier* shall be applied to only the definition of a function template or variable template. A function template definition having the *concept specifier* is called a *function concept*. A variable template definition having the *concept specifier* is called a *variable concept*. A *concept definition* refers to either a function concept and its definition or a variable concept and its initializer.

2. A *type concept* is a concept whose first template parameter is a *type-parameter*, but not a template template parameter. Otherwise, the concept is a *non-type concept*. A *variadic concept* is a concept whose first template parameter is a template parameter pack.

3. Every concept definition is also a *constexpr* declaration (C++14 §7.1.5).

4. A function concept has the following restrictions:
   - The template must be unconstrained.
   - The result type must be `bool`.
   - The declaration shall have a *parameter-declaration-clause* equivalent to `()`.  
   - The declaration shall be a definition.
   - The function shall not be recursive.
— The function body shall consist of a single `return` statement whose expression shall be a `constraint-expression`.

[Example:

```cpp
template<typename T>
concept bool C1() { return true; } // OK

template<typename T>
concept int C2() { return 0; } // error: must return bool

template<typename T>
concept bool C3(T) { return true; } // error: must have no parameters

custom bool p = 0; // error: not a template

— end example ]

A variable template has the following restrictions:
— The template must be unconstrained.
— The declared type must be `bool`.
— The declaration must have an initializer.
— The initializer shall be a `constraint-expression`.

[Example:

```cpp
template<typename T>
concept bool D1 = has_x<T>::value; // OK

template<typename T>
concept bool D2 = 3 + 4; // Error: initializer is not a constraint

template<Integral T>
concept bool D3 = has_x<T>::value; // Error: constrained concept definition

— end example ]

A program that declares an explicit or partial specialization of a concept definition is ill-formed.

[Example:

```cpp
template< typename T>
concept bool C = is_iterator<T>::value;

template< typename T>
concept bool C<T*> = true; // Error: partial specialization of a concept

— end example ]

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

1 Modify C++14 §8/1 as follows:

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

3 Insert the following paragraph after C++14 §8/1

4 A `requires-clause` (7) shall only be present if the `declarator` declares a generic function (5.1.1).

[ Example:

```cpp
template concept bool C = ...;
void f1(auto x) requires C<decltype(x)>;// Ok
void f2(int x) requires C<int>;         // Error: f2 is not a generic function
auto n requires C<decltype(n)> = g();   // Error: cannot constrain variable declaration
struct S {} requires C<S>;// Error: cannot constrain a class declaration
```

— end example ]

5 The `requires-clause` associates its constraint with its template declaration (7).

5.1 Meaning of declarators

5.1.1 Functions

1 Add the following paragraphs after C++14 §8.3.5/14.

2 A generic function is a function template whose `template-parameter-list` has a `parameter-declaration` whose `type-specifier` is either `auto` or a `constrained-type-name`. [ Example:

```cpp
auto f(auto x); // Ok
void sort(C& c); // Ok (assuming C names a concept)
```

— end example ]

3 The declaration of a generic function has a `template-parameter-list` that consists of one invented type `template-parameter` for each occurrence of `auto` or each unique occurrence of a `constrained-type-name` in the function's `parameter-declaration-clause`, in order of appearance. The invented type of `template-parameter` is a parameter pack if the corresponding `parameter-declaration` declares a function parameter pack (C++14 §8.3.5). If the `decl-specifier-seq` of the corresponding `parameter-declaration` includes a `constrained-type-specifier`, the invented type parameter is a `constrained-parameter`, whose `constrained-type-specifier` matches that of the `parameter-declaration`. (7.1). [ Example: The following generic function declarations are equivalent:

```cpp
template<typenaem T>
  constexpr bool C() { ... }

auto f(auto x, const C& y);
```
template<typename T1, C T2>
    auto f(T1 x, const T2& y);

The type of y is a type parameter constrained by C. — end example ]

4 All placeholder types introduced using the same constrained-type-name have the same invented template parameter. [ Example: The following generic function declarations are equivalent:

    auto g(C a, C* b);

    template<C T>
        auto g(T a, T* b);

    — end example ]

5 If an entity is declared by an abbreviated template declaration, then all its declarations must have the same form.

5.2 Function definitions

5.2.1 In general

1 Modify the function-definition syntax in C++14 §8.4.1 to include a requires-clause.

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

2 Add the following paragraph at the end of C++14 §8.4.1.

3 A requires-clause (7) shall only be present if the declarator declares a generic function (5.1.1) or a member function definition (6.1). [ Note: Constraints for a function template or member function template are written after the template-parameter-list. — end note ]

    [ Example:

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

        void f(C1 a, C2 b) requires D<decltype(a), decltype(b)> { } // Ok

        template<typename T>
            void f(const T& x) requires C<T>; // Error: f is declared as a template

        template<typename T>
            struct S1 {
                S1(T&) requires C1<T> { } // Defines a constrained constructor
                void f() requires C2<T> { } // Defines a constrained member function
            };

            struct S2 {
                void g(auto x) requires D< decltype(x) > { } // Ok
            };

            — end example ]

4 A function-definition shall not declare a destructor (C++14 §12.4) with a requires-clause.
6 Classes

6.1 Class members

1 In C++14 §9.2, modify the *member-declarator* syntax.

   \[
   \text{member-declarator:} \\
   \text{ declarator \ virt-specifier-seq}_{\text{opt}} \text{ pure-specifier-seq}_{\text{opt}} \text{ requires-clause}_{\text{opt}}
   \]

2 Insert the following paragraph after C++14 §9.2

3 A *requires-clause* (7) shall only be present if the *declarator* declares a constrained member function of a class template (7.4.1.1) or a generic function (5.1.1). [Example:

   \[
   \text{template<typename T> concept bool C = ...;}
   \text{template<typename T> concept bool D = ...;}
   \]

   \[
   \text{template<typename T>}
   \text{ struct A {}
   \text{ A(T*) requires C<T>; // Declares a constrained constructor}
   \text{ void f() requires D<T>; // Declares a constrained member function}
   \text{ }}
   \]

   \[
   \text{ struct B {}
   \text{ void g(int n) requires C<int>; // Error: cannot constrain a non-template}
   \text{ void h(C a, D b); // Declares a constrained generic member function}
   \text{ }}
   \]

   — end example ] [ Note: A constrained generic function declared at class scope is a member function template. — end note ]

4 A destructor (C++14 §12.4) shall not be declared with *requires-clauses.*
7 Templates

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

   *template-declaration*:
   
   \[
   \text{template} < \text{template-parameter-list} > \text{requires-clause opt}
   \]

   \[
   \text{concept} \{ \text{introduction-list} \}
   \]

   *requires-clause*:

   \[
   \text{requires} \text{constraint-expression}
   \]

   *concept*:

   \[
   \text{nested-name-specifier opt concept-name}
   \]

   *introduction-list*:

   \[
   \text{identifier}
   \]

   \[
   \text{introduction-list} , \text{identifier}
   \]

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

3 A *requires-clause* associates a *constraint-expression* with the template declaration.

4 A *template-declaration* is written in terms of its *template-parameters*. These parameters are declared explicitly in a *template-parameter-list* (7.1), or they are introduced by a *concept* and its *introduction-list*. In the latter case, each *identifier* in the *introduction-list* is declared to be a template parameter. Each template parameter shall match in kind the corresponding template parameter of the concept definition designated by the *concept*. If, after lookup, the *concept* designates an overloaded set of concept functions, the designated concept is the one with the same number of template parameters as the number of *identifier* in the *introduction-list*. The introduction of template parameters by a *concept* constrains the template declaration. This constraint is a *template-id* whose *template-name* is that of the introducing *concept* and whose template arguments are the introduced template parameters. If the introducing concept is a function concept, the constraint is a function call with no function arguments. The constraint is associated with the template declaration. [Example:

   \[
   \text{template<typename T, int N, typename... Xs>}
   \]

   \[
   \text{concept bool Inscrutable()} \{ \ldots \};
   \]

   Mergeable\{A, B, C\} struct s;

   This class template declaration of \( s \) is equivalent to:

   \[
   \text{template<typename A, int B, typename... C>}
   \]

   \[
   \text{requires Inscrutable<A, B, C>()}
   \]

   struct s;

   The introduced parameter \( C \) is a template parameter pack. — end example ]

5 [Note: A template parameter with a default argument does not introduce omitted identifiers. This would cause the introduction of an unnamed and unusable template parameter in the template declaration. [Example:

   \[
   \text{template<typename T1, typename T2, typename T3 = T2>}
   \]

   \[
   \text{concept bool Ineffable()} \{ \ldots \};
   \]

   Ineffable\{X, Y\} void f(); // Error: not all parameters are introduced
   — end example ] — end note ]
If a constrained declaration is introduced by a concept introduction, then all its declarations must have the same form.

A template declaration's associated constraints are the conjunction (logical and) of all constraint-expressions introduced by:

- constrained-parameters in a template-parameter-list,
- a requires-clause following a template-parameter-list,
- introduction by a concept (7),
- constrained-type-specifiers in a declarator, and
- a requires-clause in an init-declarator, member-declarator, or function-definition.

[ Note: A template's associated constraints form a constraint expression (7.6). — end note ]

### 7.1 Template parameters

1 Modify the template-parameter grammar in C++14 §14.1 to include constrained-parameter.

```
template-parameter:
  constrained-parameter

constrained-parameter:
  attribute-specifier-seqopt decl-specifier-seq declarator
  attribute-specifier-seqopt decl-specifier-seq declarator = constrained-initializer
  attribute-specifier-seqopt decl-specifier-seq abstract-declaratoropt
  attribute-specifier-seqopt decl-specifier-seq abstract-declaratoropt =
  constrained-initializer

constrained-initializer:
  type-id
  initializer-clause
```

[ Note: The constrained-parameter syntax is largely identical to the parameter-declaration syntax except syntax of default arguments, which is extended to also accept a type-id. — end note ]

2 Add the following paragraphs after C++14 §14.1/15.

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

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

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

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

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

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

— end example ]
```

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

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

— end example ]
```

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

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

— end example ]
```

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

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

template<typename T> concept bool U = ...;

template<P... Xs> void f4(); // requires P<Xs...>

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

Here, \( \text{Args}0, \text{Args}1, \text{etc.} \) denote elements of the template argument pack \( \text{Args} \) used as part of the introduced constraint. — end example ]
7.2 Template names

1 Insert the following paragraphs after C++14 §14.2/7.

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

   [ Example:
     
     template<typename T> concept bool True = true;
     template<typename T> concept bool False = false;
     
     template<False T> struct S;
     template<True T> using Ptr = T*;
     
     S<int>* x;   // Error: int does not satisfy the constraints of False.
     Ptr<int> z;  // Ok: z has type int*
     
     — end example ]

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

7.3 Template arguments

7.3.1 Template template arguments

1 Modify C++14 §14.3.3.

2 A template-argument matches a template template-parameter (call it \textit{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 \textit{A}) matches the corresponding template parameter in the template-parameter-list of \textit{P}, and the associated constraints of \textit{P} shall subsume the associated constraints of \textit{A} (7.6). [ Example:

     template<typename T>
     concept bool X = has_x<T>::value;
     template<typename T>
     concept bool Y = X<T> & has_y<T>::value;
     template<typename T>
     concept bool Z = Y<T> & has_z<T>::value;
     
     template<template<Y> class C> class temp { ... };
     class temp<X T> class x;
     template<Z T> class z;
     
     temp<X> s1; // Ok: X is subsumed by Y
     temp<Z> s2; // error: Z subsumes Y
     
     — end example ]
7.4 Template declarations

7.4.1 Class templates

1 Insert the following paragraph after C++14 §14.5.1/3.

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

[Example:

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

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

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

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

7.4.1.1 Member functions of class templates

1 Add the following paragraphs after C++14 §14.5.1.1.

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

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

—end example ]

3 Constraints on member functions are instantiated as needed during overload resolution, not when the class template is instantiated (C++14 §14.7.1). [Note: Constraints on member functions do not affect the declared interface of a class. That is, a constrained copy constructor is still a copy constructor, even if it will not be viable for a specialization of the class template. —end note ]

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

```cpp
template<typename T>
  void S<T>::f() { } // Error: no declaration of f() in S<T>.
```
template<typename T>
    void S<T>::f() requires C<T>() { } // Ok: defines S<T>::f

— *end example* ]

7.4.2 Member templates

1 Insert the following paragraph after C++14 §14.5.2/1.

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

[ *Example:* 

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

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

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

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

7.4.3 Friends

1 Add the following paragraphs after C++14 §14.5.4/9.

2 A *constrained friend* is a friend of a class template with associated constraints. A constrained friend can be a constrained class template, constrained function template, or an ordinary (non-template) function. Constraints on template friends are written using shorthand, introductions, or a requires clause following the *template-parameter-list*. Constraints on non-template friend functions are written after the result type. [ *Example:* When c is a type concept, all of the following are valid constrained friend declarations.

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

        template<C W>
            friend struct Z { };

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

— *end example* ]

§ 7.4.3
A non-template friend function shall not be constrained unless the function's parameter or result type depends on a template parameter. [Example:

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

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

```cpp
template<typename T>
struct S {
    friend void f<> (T x) requires C<T>(); // Error: declares a specialization
    friend void g(T x) requires C<T>() { } // OK: does not declare a specialization
};
```
—end example]

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

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

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

1 Modify C++14 §14.5.5.1/2.

2 A partial specialization matches a given actual template argument list if the template arguments of the partial specialization can be deduced from the actual template argument list (C++14 §14.8.2) and the deduced template arguments satisfy the constraints of the partial specialization, if any (7.6).

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

1 Modify C++14 §14.5.5.2/1.

2 For two class template partial specializations, the first is at least as specialized as the second if, given the following rewrite to two function templates, the first function template is at least as specialized as the second according to the ordering rules for function templates (C++14 §14.5.6.2):

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

[Example:

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

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

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

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

7.4.5 Function templates

7.4.5.1 Template argument deduction

1 Immediately after C++14 §14.8.2/5, add the following paragraph:

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

7.4.5.2 Function template overloading

1 Modify C++14 §14.5.6.1/6.

2 A function template can be overloaded either by (non-template) functions of its name or by (other) function templates of the same name. When a call to that name is written (explicitly, or implicitly using the operator notation), template argument deduction 7.4.5.1, and checking of any explicit template arguments C++14 §, and checking of associated constraints 7.6 are performed for each function template to find the template argument values (if any) that can be used with that function template to instantiate a function template specialization that can be invoked with the call arguments. For each function template, if the argument deduction and checking succeeds, the template-arguments (deduced and/or explicit) are used to synthesize the declaration of a single function template specialization which is added to the candidate functions set to be used in overload resolution. If, for a given function template, argument deduction fails, no such function is added to the set of candidate functions for that template. The complete set of candidate functions includes all the synthesized declarations and all of the non-template overloaded functions of the same name. The synthesized declarations are treated like any other functions in the remainder of overload resolution, except as explicitly noted in C++14 §.

3 Modify C++14 §14.5.6.1

4 Two function templates are equivalent if they are declared in the same scope, have the same name, have identical template parameter lists, and have return types, and parameter lists, and constraints 7.6 that are equivalent using the rules described above to compare expressions involving template parameters.

7.4.5.3 Partial ordering of function templates

1 Modify C++14 §14.5.6.2/2.
Partial ordering selects which of two function templates is more specialized than the other by transforming each template in turn (see next paragraph) and performing template argument deduction using the function type. The deduction process determines whether one of the templates is more specialized than the other. If so, the more specialized template is the one chosen by the partial ordering process. If the two templates have identical template parameter lists and equivalent return types and parameter lists, then partial ordering selects the template whose associated constraints subsume but are not equivalent to the associated constraints of the other. A constrained template is always selected over an unconstrained template.

### 7.5 Template instantiation and specialization

#### 7.5.1 Implicit instantiation

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

#### 7.5.2 Explicit instantiation

An explicit instantiation of constrained template declaration (7) or constrained member function declaration (7.4.1.1) shall satisfy the associated constraints of that declaration (7.6).

**Example:**

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

template<typename T>
requires C<T>
struct X { }

template struct X<int>; // Error: int does not satisfy C.
```

---

### 7.6 Template constraints

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

**constraint-expression:**

```
logical-or-expression
```

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

- is a constant expression,
- has type bool, and
- both operands $P$ and $Q$ in every subexpression of a constraint of the form $P \lor Q$ or $P \land Q$ have type bool.
A constraint-expression defines a subset of constant expressions over which certain logical implications can be proven during translation. The requirement that operands to logical operators have type `bool` prevents constraint expressions from finding user-defined overloads of those operators and possibly subverting the logical processing required by constraints.

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

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

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

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

A subexpression of a constraint-expression that calls a function concept or refers to a variable concept 4.1.4 is a concept check. A concept check is not evaluated; it is simplified according to the rules described in this.

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

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

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

Example:

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

A concept check is not an atomic expression.

Constraints are simplified by reducing them to expressions containing only logical operators and atomic constraints. Concept checks and requires-expressions are replaced by simplified expressions. A concept check is not an atomic expression.

Constraints are simplified by reducing them to expressions containing only logical operators and atomic constraints. Concept checks and requires-expressions are replaced by simplified expressions. A concept check is not an atomic expression.
A concept check that calls a function concept is simplified by substituting the explicit template arguments into the named function body’s return expression. A concept check that refers to a variable concepts is simplified by substituting the template arguments into the variable’s initializer.

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

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

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

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

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

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

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

template<typename T>
requires C<T> void h();

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

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

— end example ]

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

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

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