Document Number: N3929
Date: 2014-02-14
Revises: N3889
Editor: Andrew Sutton
University of Akron
asutton@uakron.edu

Working Draft, Information technology – Programming languages, their environments and system software interfaces – Information technology - Programming languages, their environments and system software interfaces - C++ Extensions for Concepts

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

Contents

1 General [intro]

1.1 Scope [general.scope]

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

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

1.2 Normative references [intro.refs]

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

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

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

2 ISO/IEC 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 [intro.defns]

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

1.3.1 **[atomic.defns.atomic-constraint]**
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 [intro.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 [intro.ack]

1 The design of this specification is based, in part, on a concept specification of the algorithms part of the C++ standard library, known as ``The Palo Alto" TR (WG21 N3351), which was developed by a large group of experts as a test of the expressive power of the idea of concepts. Despite

syntactic differences between the notation of the Palo Alto TR and this TS, the TR can be seen as a large-scale test of the expressiveness of this TS.

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

2 Lexical conventions [lex]

2.1 Keywords [lex.key]

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

3 Expressions [expr]

3.1 Primary expressions [expr.prim]

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

primary-expression:
requires-expression

3.1.1 Lambda expressions [expr.prim.lambda]

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). The associated constraints of the generic lambda are the conjunction of constraints introduced by the use of *constrained-type-name*s in the *parameter-declaration-clause*. 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*.

3 All placeholder types introduced using the same *concept-name* have the same invented template parameter.

3.1.2 Requires expressions [expr.req]

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

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

expression `;`

compound-requirement:

`constexpr`*opt* `{` *expression* `}` `noexcept`*opt* *trailing-return-type*$_{opt}$ `;`

type-requirement:

typename-specifier `;`

nested-requirement:

requires-clause `;`

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

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

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

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

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

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

— *end example*]

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

6 The *requirement-body* is a sequence of *requirement*s separated by semicolons. These *requirement*s 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 (6.1.5). The kinds of atomic constraints introduced by a *requirement* are:

— A *valid expression constraint* is a predicate on a dependent 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*. Valid expressions are the targets of other atomic constraints introduced by *requirement*s.
— A *valid type constraint* is a predicate on a dependent 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*. Associated types are the targets of conversion constraints, described below.
— A *conversion constraint* is satisfied if and only if a valid expression is implicitly convertible to a non-dependent or associated type (C++14 §4).
— A *constant expression constraint* is satisfied if and only if a valid expression is a constant expression (C++14 §5.19).
— An *exception constraint* is satisfied if and only if, for a valid expression `E`, the expression `noexcept(E)` evaluates to `true` (C++14 §5.3.7).

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

3.1.2.1 Simple requirements [expr.req.simple]

1 A *simple-requirement* introduces a valid expression constraint for its *expression* . The expression is an unevaluated operand (C++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).

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

— *end example*]

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

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

— *end example*]

3.1.2.2 Type requirements [expr.req.type]

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

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

— *end example*] — *end note*]

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

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

— *end example*]

3.1.2.3 Nested requirements [expr.req.nested]

1 A *nested-requirement* introduces an additional constraint expression 6.1.5 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 [expr.req.compound]

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

— a valid expression constraint,
— an optional associated type requirement
— an optional conversion requirement or nested requirement,
— an optional constant expression requirement, and
— an optional an exception requirement.

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

4 The presence of a *trailing-return-type* introduces additional requirements. These depend on whether or not the *trailing-return-type* contains a *constrained-type-specifier*.

— If so, two new constraints are introduced. Let `C` be the *constrained-type-specifier*. A valid type constraint is introduced for the type formed by substituting `decltype((E))` for `C` in the *trailing-return-type*. A new constraint is formed also according to the rules described in by substituting `decltype((E))` as the constraint argument.
— Otherwise, a valid type constraint is introduced for the trailing *type-specifier*. Let `T` be the associated type formed form successful substitution. A conversion constraint requiring that `E` be implicitly convertible to `T` is also introduced.

5 If the `constexpr` specifier is present then a constant expression constraint is introduced for the valid expression `E`.[*Note:* The constraint is satisfied only when `E` is a constant expression. *— end note*]

6 If the `noexcept` specifier is present, then an exception constraint is introduced for the valid expression `E`.[*Note:* The constraint is satisfied only when `noexcept(E)` evaluates to `true`. *— end note*]

7 [*Example:*

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

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

Requirement #1 introduces only a valid expression requirement and is equivalent to a *simple-requirement* containing the same expression. In requirement #2, `*p` is a required valid expression and `typename T::r` is a required associated types. After substitution, `*p` must be implicitly convertible to the type named by `typename T::r`. In #3, `const decltype((f(x)))&` is a required associated type, and `Inscrutible<decltype((f(x)))>` is a nested requirement. Requirement

#3 includes an exception requirement, that `g(x)` does not propagate requirements. It also requires an associated type requirement (`T&`) and a conversion requirement (`g(x)` must be implicitly convertible to `T&`). Requirement #5 includes a constant expression requirement, namely that `T::value` must be a constant expression, but no associated type, conversion, or nested requirements. #6 includes a constant expression requirement (`T() + T()`), an associated type requirement (`T`), and a conversion requirement (`T() + T()` must be implicitly convertible to `T`). — *end example*]

4 Declarations [dcl.dcl]

4.1 Specifiers [dcl.spec]

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

decl-specifier:
`concept`

4.1.1 Simple type specifiers [dcl.type.simple]

1 Extend the *simple-type-spcifier* 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 [dcl.spec.auto]

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 [dcl.spec.constr]

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 6.1. Otherwise, the meaning of *constrained-type-specifier*s 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*]

4 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 5.1.1.

5 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*.

6 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).

7 The *introduced constraint* is a constraint expression (6.1.5) synthesized from the *concept-name* or *partial-concept-id* in the *constrained-type-name*.

8 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.

```
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.

9 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:*

```
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*]

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

11 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*]

12 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.

13 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, Args>()`. Otherwise, the introduced constraint is the same as the remaining candidate.

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

```
template<typename T>
concept bool V1 = ...;

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

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

template<typename T, typename T2>
concept 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*]

15 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 the introduced constraint is an associated constraint of the corresponding template. 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 [dcl.concept]

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:*

```
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

      concept bool p = 0; // error: not a template
```

— *end example*]

5 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:*

```
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*]

6

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

```
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*]

8 [*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 [dcl.decl]

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.

init-declarator:
    declarator requires-clause$_{opt}$ *initializer*$_{opt}$

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

4 A *declarator* followed by a *requires-clause* declares a *constrained declaration*. The constrained declaration shall be a a template declaration or member function declaration of a class template. The *requires-clause* associates its constraint with the declaration (6.1.5).

5 A declarator that declares a constrained variable or type is ill-formed.

6 [*Example:* Let `C` be a variable concept constraining a single type parameter:

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

void f1(auto x) requires C<decltyp(x)>; // Ok
void f2(int x) requires C<int>;         // Error: f2 does not have dependent type

auto n requires C<decltype(n)> = g();   // Error: cannot declare a constrained variable.
```

— *end example*]

5.1 Meaning of declarators [dcl.meaning]

5.1.1 Functions [dcl.fct]

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:*

```
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 occurence 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). The associated constraints of a generic function are the conjunction of constraints introduced by the use of *constrained-type-name*s in the *parameter-declaration-clause* and the *requires-clause* in the function's declarator, if present. [*Example:* The following generic function declarations are equivalent:

```
auto f(auto x, const C& y);

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

— *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.

6 Templates [temp]

1 Modify the *template-declaration* grammar in .

2

> *template-declaration:*
> `template <` *template-parameter-list* `>` *requires-clause$_{opt}$* *declaration*
> *concept-introduction declaration*
> *requires-clause:*
> `requires` *constraint-expression*

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

4 A *constrained template declaration* is a *template-declaration* with associated constraints. The *associated constraint*s of a constrained template declaration are the conjunction of the associated constraints of all

— *constrained-parameter*s in template's *template-parameter-list* (6.1).
— and all *constraint-expression*s introduced by *requires-clause* in the *template-declaration*.
[*Note:* A function or member function may have a *requires-clause* in its declarator. These constraints are also part of the associated constraints of the template declaration. — *end note*]

5 The associated constraints of a *concept-introduction* are those required by the referenced concept definition. [*Example:*

```
template<typename T>
  concept bool Integral() { return is_integral<T>::value; }

template<Integral T>
  requires Unsigned<T>()
    T binary_gcd(T a, T b);
```

The associated constraints of `binary_gcd` are denoted by the conjunction `Integral<T>() && Unsigned<T>()`. — *end example*]

6 A constrained template declaration's associated constraints must be satisfied (6.1.5) to allow instantiation of the constrained template. The associated constraints are satisfied by substituting template arguments into the constraints and evaluating substituted expression. Constraints are satisfied when the result of that evaluation is `true`. Class template, alias template, and variable template constraints are checked during name lookup (6.1.1); function template constraints and class template partial specialization constraints are checked during template argument deduction (6.1.3.4.1).

7 Any usage of a constrained template in a template declaration is ill-formed unless the associated constraints of the constrained template are subsumed by the associated constraints of template parameter. No diagnostic is required.

6.1 Template parameters [temp.param]

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

> *template-parameter:*
> *constrained-parameter*
> *constrained-parameter:*
> *constrained-type-specifier* `...`$_{opt}$ *identifier*
> *constrained-type-specifier identifier* `=` *constrained-type-initializer*

<u>*constrained-type-initializer:*</u>
<u>*type-id*</u>
<u>*id-expression*</u>

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

3 A *constrained template parameter* is a *constrained-parameter* or a *parameter-declaration* whose *decl-specifier-seq* contains a *constrained-type-specifier*. A *constrained-parameter* defines its identifier to be a template parameter, and it introduces an associated constraint of the template declaration. In a constrained template parameter, the concept definition referred to by the *constrained-type-specifier* is called the *constraining concept*, and its first template parameter is called the *prototype parameter*.

4 The kind of the template parameter declared by a *constrained-parameter* matches that of the prototype parameter. If the template parameter is a template template parameter, then it has the same number and kind of template parameters as the prototype parameter. [*Example:*

```
template<typename T>
  concept bool C1 = ...;
template<template<typename> class X>
  concept bool C2() { ... }

template<C1 T> void f(); // T is a type parameter
template<C2 X> void g(); // X is a template with one type parameter
```

— end example]

5 If the constrained parameter is a non-type template parameter, then then type of the declared parameter is formed by substituting the type of prototype parameter for the *constrained-type-specifier*. [*Example:*

```
template<int N>
  concept bool P = ...;

template<P N> void x();        // N has type int
template<const P* N> void y(); // N has type const int*
```

— end example]

6 If the constraining concept is a variadic concept (4.1.4), then the constrained template parameter is declared as a parameter pack. The declaration of the constrained template parameter shall include an ellipses. [*Example:*

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

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

— end example]

7 The *constrained-type-initializer* shall match the constrained template parameter.
[*Example:*

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

— end example]

8 If the constrained template parameter is not a template parameter pack, then the constraint introduced by the *constrained-type-specifier* in a constrained template parameter is formed according to the rules in section 4.1.3. When forming the constraint, the declared parameter used as a template argument in place of the constrained type.
[*Example:*

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

[*Example: — end example*]
— *end example*]

9 If the constrained template parameter is a template parameter pack, the formation of the constraint depends on whether the constraining concept is a variadic concept. Let `T` be the declared parameter, `C` be the constraining concept, and `Args...` be a sequence of template arguments from a *partial-concept-id*, possibly empty. If `C` is a variadic concept, then the associated constraint is a *template-id* of the form `C<T..., Args...>`. Otherwise, if `C` is not a variadic concept, the associated constraint is a conjunction of sub-constraints `C<T`*i*`, 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, Args...>()` where `X` is either the template parameter pack `T` or an element `T`*i*. [*Example:*

```
template<X... Xs> void f4();    // requires X<Xs...>
template<C1... Args> void f5(); // requires C1<Args0> && C1<Args1> && ... && C1<Argsn>
```

— *end example*]

6.1.1 Template names [temp.names]

1 Modify C++14 §14.2/6.

2 A *simple-template-id* that names a class template specialization is a *class-name*. The template-arguments shall satisfy the associated constraints of the designated class, if any. [*Example:*

```
template<Object T, int N> // T must be an object type
  class array;

array<int&, 3>* p; // error: int& is not an object type
```

— *end example*] [*Note:* 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*]

6.1.2 Template arguments [temp.arg]

6.1.2.1 Template template arguments [temp.arg.template]

1 Modify C++14 §14.3.3.

2 A *template-argument* matches a template *template-parameter* (call it `P`) when each of the template parameters in the *template-parameter-list* of the *template-argument*'s corresponding class template or alias template (call it `A`) matches the corresponding template parameter in the *template-parameter-list* of `P`, and the associated constraints of `P` shall subsume the associated constraints of `A` (6.1.5). [*Example:*

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

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

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

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

— *end example*]

6.1.3 Template declarations [temp.decls]

6.1.3.1 Class templates [temp.class]

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

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:*

```
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*]

6.1.3.2 Friends [temp.friend]

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.

```
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*]

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

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

— *end example*]

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

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

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

— *end example*]

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

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

6.1.3.3.1 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 (6.1.5).

6.1.3.3.2 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:*

```
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*]

6.1.3.4 Function templates [temp.fct]

6.1.3.4.1 Template argument deduction [temp.deduct]

1 Modify C++14 §14.8.2/2.

2 When an explicit template argument list is specified, the template arguments must be compatible with the template parameter list and must result in a valid function type as described below; otherwise type deduction fails. Specifically, the following steps are performed when evaluating an explicitly specified template argument list with respect to a given function template:

— The specified template arguments must match the template parameters in kind (i.e., type, non-type, template). There must not be more arguments than there are parameters unless at least one parameter is a template parameter pack, and there shall be an argument for each non-pack parameter. Otherwise, type deduction fails.
— Non-type arguments must match the types of the corresponding non-type template parameters, or must be convertible to the types of the corresponding non-type parameters as specified in C++14 §14.3.2 otherwise type deduction fails.
— If the function template is constrained, the specified template arguments are substituted into the associated constraints and evaluated. If the result of the evaluation is `false`, type deduction fails.
— The specified template argument values are substituted for the corresponding template parameters as specified below.

6.1.3.4.2 Function template overloading [temp.over.link]

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 6.1.3.4.1~~, and~~ checking of any explicit template arguments C++14 § , and checking of associated constraints 6.1.5 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 6.1.5 that are equivalent using the rules described above to compare expressions involving template parameters.

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

1 Modify C++14 §14.5.6.2/2.

2 Partial ordering selects which of two function templates is more specialized than the other by transforming each template in turn (see next paragraph) and performing template argument deduction using the function type. The deduction process determines whether one of the templates is more specialized than the other. If so, the more specialized template is the one chosen by the partial ordering process. If the two templates have identical template parameter lists and equivalent return types and parameter lists, then partial ordering selects the template whose associated constraints subsume but are not equivalent to the associated constraints of the other 6.1.5. A constrained template is always selected over an unconstrained template.

6.1.4 Template instantiation and specialization [temp.spec]

6.1.4.1 Implicit instantiation [temp.inst]

1 Insert the following pargraph after C++14 §14.7.1/1.

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

6.1.4.2 Explicit instantiation [temp.explicit]

1 Insert the following paragraph under C++14 §14.7.2.

2 An explicit instantiation of constrained template declaration (6) or constrained member function declaration (6.1.3.1.1) shall satsify the associated constraints of that declaration (6.1.5). [*Example:*

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

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

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

— *end example*]

6.1.5 Template constraints [temp.constr]

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

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

constraint-expression:
    logical-or-expression

Constraint expressions are required after a *requires-clause* and in the body or initializer of a concept definition.

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

— is a constant expression,
— has type `bool`, and
— the operands to every subexpression that is either a *logical-or-expression* of the form `P || Q` or a *logical-and-expression* of the form `P && Q` has type `bool`.

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

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

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

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

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

6 A subexpression of a *constraint-expression* that calls a function concept or refers to a variable concept 4.1.4 is a *concept check*.

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

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

8 Constraints are *simplified* by reducing them to expressions containing only logical operators and atomic constraints. Concept checks and *requires-expression*s are replaced by simplified

expressions. [*Note:* An implementation is not required to normalize the constraint by rewriting it in e.g., disjunctive normal form. — *end note*]

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

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

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

12 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).

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

6.1.6 Concept introductions [concept.intro]

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

2 A *concept-introduction* allows the declaration of template and its associated constraints in a concise way.

concept-introduction:
  concept-name `{` *introduction-list* `}` *declaration*
introduction-list:
  identifier
  introduction-list , *identifier*

3 The *concept-introduction* names a concept and a list of identifiers to be used as template parameters, called the *introduced parameters* in the declaration. The number of *identifier*s in the *introduction-list* shall match the number of template parameters in the named concept. [*Example:*

```
template<typename I1, typename I2, typename O>
  concept bool Mergeable() { ... };

Mergeable{First, Second, Out} // OK
  Out merge(First, First, Second, Second, Out);

Mergeable{X, Y} // Error: not enough parameters
  void f(X, Y);
```

— *end example*]

4 The introduced parameters are the template parameters of the declaration, and they match the template parameters in the declaration of the named concept. The associated constraints of the declaration are formed by applying the introduced parameters as arguments to the named concept (). [*Example:* The following declaration

```
Mergeable{X, Y, Z}
  Z merge(X, X, Y, Y, Z);
```

is equivalent to the declaration below.

```
template<typename X, typename Y, typename Z>
  requires Mergeable<X, Y, Z>()
    Z merge(X, X, Y, Y, Z);
```

— *end example*]

5 If a constrained declaration is introduced by a concept introduction, then all its declarations must have the same form.

6 The sequence of introduced parameters in a *concept-introduction* shall have the same number of template parameters as the referenced concept. [*Example:*

```
template<typename T1, typename T2, typename T3 = T2>
  concept bool Ineffable() { ... };

Ineffable{X, Y} void f();    // Error: does not introduce all parameters
Ineffable{X, Y, Z} void g(); // Ok
```

— *end example*] [*Note:* Allowing default arguments to be deduced in a *concept-introduction* would cause the introduction of an unnamed and unusable template parameter in the template declaration. — *end note*]

Publish to new window