Document Number: N3889
Date: 2014-01-20
Reply to: Andrew Sutton
University of Akron
asutton@uakron.edu

Concepts Lite Specification

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

Contents

1 General [intro]

1.1 Introduction [intro.intro]

1 C++ has long provided language support for generic programming in the form of templates. However, these templates are unconstrained, allowing any type or value to be substituted for a template argument, often resulting in compiler errors. What is lacking is a specification of an interface for a template, separate from its implementation, so that a use of a template can be selected among alternative templates and checked in isolation.

2 A concept is a predicate that defines the syntactic and semantic requirements on template arguments. A type that satisfies these requirements is said to be a *model* of that concept, or that the type *models* the concept. Syntactic requirements specify the set of valid expressions that can be used with conforming types, and their associated types. Semantic requirements describe the required behavior of those syntactic requirements and also provide complexity guarantees. Concepts are the basis of generic programming in C++ and support the ability to reason about generic algorithms and data structures independently of their instantiation by concrete template arguments.

3 Concepts are not new to C++ or even to C (where Integral and Arithmetic are long-established concepts used to specify the language rules for types); the idea of stating and enforcing type requirements on template arguments has a long history, e.g., several methods are discussed in *The Design and Evolution of C++* (1994). Concepts were a part of documentation of the STL and are used to express requirements in the C++ standard, ISO/IEC 14882. For example, Table 106 gives the definition of the STL `Iterator` concept as a list of valid expressions and their result types, operational semantics, and pre- and post-conditions.

4 This specification describes a solution to the problem of constraining template arguments in the form "Concepts Lite." The goals of "Concepts Lite" are to

- allows programmers to directly state the requirements of a set of template arguments as part of a template's interface,
- supports function overloading and class template specialization based on constraints,
- seamlessly integrates a number of orthogonal features to provide uniform syntax and semantics for generic lambdas, `auto` declarations, and result type deduction,
- fundamentally improves diagnostics by checking template arguments in terms of stated intent at the point of use,
- do all of this without any runtime overhead or longer compilation times.

"Concepts Lite" does not provide facilities for checking template definitions separately from their instantiation, nor does it provide facilities for specifying or checking semantic requirements.

5 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.

1.2 Scope [intro.scope]

1 This Technical Specification specifies requirements for implementations of an extension to the C++ programming language concerning the application of constraints to template arguments, the use of constraints in function overloading and class template specialization, and the definition of those constraints.

2 International Standard, ISO/IEC 14882, provides important context and specification for this Technical Specification. Notes in this Technical Specification indicate how and where new text should be added to or removed from the International Standard.

1.3 Normative References [intro.refs]

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

— ISO/IEC 14882, Programming Language C++

1.4 Terms and Definitions [intro.defs]

For the purpose of this document, the following definitions apply.

1.4.1 [defns.concept]

concept

A template declared with the `concept` declaration specifier.

1.4.2 [defns.constraint]

constraint

An constant expression that evaluates properties of template arguments, determining whether or not they can be substituted into a template.

1.4.3 [defns.requirement]

requirement

Used interchangeably with "constraint". The term "requirement" is often used to refer to a single atomic propositions such as valid expressions and associated types (i.e., a syntactic requirement).

1.5 Conformance [intro.conform]

Conformance is specified in terms of behavior.

1.6 Acknowledgements [intro.ack]

The following people have contributed to writing and editing of this technical specification:

2 Lexical conventions [lex]

2.1 Keywords [lex.key]

Add to table 4, the keywords `concept` and `requires`.

3 Expressions [expr]

3.1 Primary expressions [expr.prim]

3.1.1 General [expr.prim.general]

Modify the grammar of *primary-expression*

primary-expression:
literal
`this`
...
requires-expression

.

Add the following sections.

3.1.2 Requires expressions [expr.requires]

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

requires-expression:
  `requires` *requirement-parameter-list requirement-body*

requirement-parameter-list:
  `(` *parameter-declaration-clause*$_{opt}$ `)`

requirement-body:
  `{` *requirement-list* `}`

requirement-list:
  requirement$_{opt}$
  requirement-list `;` *requirement*$_{opt}$

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

simple-requirement:
  expression

compound-requirement:
  `{` *expression* `}` *trailing-requirements*

type-requirement:
  type-id

nested-requirement:
  requires-clause

trailing-requirements:
  constexpr-specifier$_{opt}$ *result-type-requirement*$_{opt}$ *noexcept-specifier*$_{opt}$

result-type-requirement:
  `->` *type-id*

constexpr-specifier:
  `constexpr`

noexcept-specifier:
  `noexcept`

2 A *requires-expression* is a constant expression, and its result type is is `bool`. [*Example:*

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

The return expression is a `requires` expression and the statements written within the enclosing braces denote specific syntactic requirements on the template parameter `T`. — *end example*]

3 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 requirements within the *requirement-body* and are not visible outside the closing `}` of *requirement-body*. The *requirement-body* is a list of requirements written as statements. These statements may refer to local arguments, template parameters, and any other declarations visible from the concept definition.

4 A *requires-expression* evaluates to `true` if and only if every *requirement* in the *requirement-list* evaluates to `true`. The semantics of each requirement are described in the following sections.

3.1.2.1 Simple requirements [expr.req.simple]

1 A *simple-requirement* introduces a requirement that instantiation of the *expression* will not result in a substitution failure. The expression is not evaluated. A *simple-requirement* evaluates to `true` if and only if instantiation succeeds. [*Example:*

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

— *end example*]

3.1.2.2 Compound requirements [expr.req.compound]

1 A *compound-requirement* introduces a set of constraints pertaining to a single *expression*. The expression is not evaluated. A *compound-requirement* evaluates to `true` if and only if instantiation succeeds and every other associated constraint evaluates to `true`.

2 If a *result-type-requirement* is present then a) substitution into the result type shall succeed and b) the result type of the instantiated expression must be implicitly convertible to the instantiated type **??**. [*Example:*

```
template<typename T>
  concept bool Deref() {
    return requires(T p) {
      {*p} -> T::reference;
    }
  }
```

The concept evaluates to `true` only when the expression `*p` and the type name `T::reference` do not result in substitution failures when instantiate, and `decltype(*p)` is convertible to `T::reference` when instantiated. — *end example*]

3 If the *type-id* specified by the *result-type-requirement* refers to a concept, then that concept is applied to the result type of the instantiated expression. [*Example:*

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

template<typename T>
  concept bool Range() {
    return requires(T x) {
      {begin(x)} -> Iterator; // Iterator
    }
  }
```

The concept evaluates to `true` iff the expression `begin(x)` can be instantiated, and the expression `Iterator<decltype(begin(`
evaluates to `true`. — *end example*]

4 If the `constexpr` specifier is present in the *constraint-specifier-seq*, the instantiated expression must be `constexpr`-evaluable. [*Example:*

```
template<typename Trait>
  bool concept Boolean_metaprogram() {
    return requires (Trait t) {
      {Trait::value} constexpr -> bool;
      {t()} constexpr -> bool;
    }
  }
```

When instantiated, the resolved nested `value` member and function call operator must be `constexpr`-evaluable. If either instantiated expression is not `constexpr`-evaluable the concept is not satisfied. — *end example*]

5 If the `noexcept` specifier is present, in the *constraint-specifier-seq* the instantiated expression must not propagate exceptions. [*Example:*

```
template<typename T>
  bool concept Nothrow_movable() {
    return requires (T x) {
      {T(std::move(x))} noexcept;
      {x = std::move(x)} noexcept -> T&;
    }
  }
```

When instantiated, the resolved move constructor and move assignment operator must not propagate exceptions. If either of the instantiated expressions does propagate exceptions, the concept is not satisfied. — *end example*]

3.1.2.3 Type requirements [expr.req.type]

1 A *type-requirement* introduces a requirement that an associated *type-id* can be formed when instantiated. A *type-requirement* evaluates to `true` if and only if instantiation does not result in a substitution failure.

3.1.2.4 Nested requirements [expr.req.nested]

1 A *nested-requirement* introduces additional constraints to be evaluated as part of the *requires-expression* and evaluates to `true` only if the given expression evaluates to `true`. [*Example:* Nested requirements are generally used to provide additional constraints on associated types within a *requires-expression*.

```
template<typename T>
  concept bool Input_range() {
    return requires(T range) {
      typename Iterator_type<T>;
      requires Input_iterator<Iterator_type<T>>();
    };
  }
```

— *end example*]

3.2 Constraints [expr.constr]

Add the following section after 5.19.

1 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:
  or-expression

2 An *or-expression* is a *constraint* if and only if it is a *constant-expression* of type `bool` where operands to logical operators in its sub-expressions shall also be of type `bool`. [*Note:* A *constraint-expression* defines a subset of constant expressions over which certain logical implications can be proven during translation. — *end note*]

3 [*Example:*

```
template<typename T>
  requires is_integral<T>::value && is_signed<T>::value
void f(T x); // #2

constexpr int Fn() { return 1; }

template<typename T>
  requires Fn() // Error: Fn() is not a valid constraint
void g(T x);
```

— *end example*]

4 A subexpression of a *constraint-expression* that calls a function concept ?? or refers to a variable concept ?? is a *concept check*. When processing a constraint containing a concept check, that concept check is replace by the concept's definition. For function concepts, the definition is formed by substituting the explicit template arguments into the return expression. For variable concepts, the definition is formed by substituting the template arguments into the variable's initializer. [*Example:*

```
template<typename T>
  concept bool C() { return sizeof(T) >= 4; }

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

template<typename X>
  requires C<X>() // Processed as sizeof(X) >= 4
void f();

template<typename Q>
  requires D<Q>   // Processed as sizeof(Q) >= 4
void g();
```

— *end example*]

5 Certain subexpressions of a *constraint-expression* are considered *atomic constraints*. A constraint is atomic if it is not a *logical-and-expression*, a *logical-or-expression*, or a *concept check*. [*Note:* The partial ordering of constraints requires the decomposition of constraint expressions into lists of atoms. — *end note*] [*Example:* The expression `x == y && is_integral<T>::value` has two atoms: `x == y` and `is_integral<T>::value`. — *end example*]

6 Two atomic constraints are equivalent if and only if they have the same syntax. [*Example:* The expression `M == N` is equivalent to itself not equivalent to `N == M`. — *end example*]

7 Two *constraint-expressions* are equivalent if and only if they have the same atom constraints.

8 A *constraint-expression* `P` is *at least as strict* as `Q` if and only if the atomic constraints in `P` subsume those in `Q`. [*Note:* Determining if `P` subsumes `Q` is equivalent to deciding the validity of the logical argument that `P` implies `Q`. — *end note*]

9 A *constraint-expression* `P` is *stricter* than `Q` if and only if `P` is at least as strict as `Q` and `Q` is not at least as strict as `P`. In this case, `Q` is *weaker* than `P`.

4 Declarations [dcl.dcl]

4.1 Specifiers [dcl.spec]

Modify the grammar of *decl-specifier*.

1 The specifiers that can be used in a declaration are

decl-specifier:
  . . .
  `concept`

4.1.1 Simple type specifiers [dlt.type.simple]

Modify the grammar of *type-name*.

1 The simple type specifiers are

simple-type-specifier:
  . . .
  type-name

type-name:
  . . .
  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.spc.auto]

Insert a new paragraph after 3

.

1 If the `auto` *type-specifier* appears as one of the *decl-specifiers* in the *decl-specifier-seq* of a `parameter-declaration` of a function declarator, then the function is a *generic function* 5.1.1.

4.1.3 Constrained type specifiers [dcl.spec.constr]

1 A *constrained-type-name* introduces constraints for a *template-parameter* or placeholder type depending on the context in which it appears. A *constrained-type-name* can be used as the *type-specifier* of template parameters, a *result-type-requirement* in a *compound-requirement*, or wherever the `auto` specifier is used.

2 A *constrained-type-name* cannot be used

— as part of the type of a variable declaration,

— as part of a function's result type

— in the place of `auto` in `decltype(auto)`, or

— as part of a *conversion-function-id*.

3 A *constrained-type-name* that refers to a *non-type concept* cannot be used as part of the *type-specifier* that introduces a placeholder type. [*Note:* Non-type concepts can be used as type specifiers of non-type

template parameters and template template parameters. — *end note*] [*Example:* template<int N> concept bool Prime() ...
void f(Prime n) // Error
template<Prime P> // Ok void g(); — *end example*]

4.1.3.1 Constraint formation [dcl.constr.form]

1 The introduced constraints are formed by applying applying template arguments to the concept referred to by the *constrained-type-name*. The constraint corresponding to a *concept-name* is formed by applying a declared or invented *template-parameter* as an explicit template argument. The constraint corresponding a *partial-concept-id* is formed by applying a declared or invented *template-parameter* as the first template argument and the original template arguments following the first.

2 If the *concept-name* or if the *concept-name* of a *partial-concept-id* refers to a function concept, the new constraint is formed as a function call with no function arguments.

3 [*Example:* The formation of constraints depends on whether the concept is defined as a variable or function template and the number of arguments the concept can accept.

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

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

template<typename T>
  concept bool V1() { return ...; }

template<typename T, typename T2>
  concept bool V2() { return ...; }
```

The following constrained template parameter declarations result in the formation of the following constraints.

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

Note that `X` could also be the implementation defined name of an invented template parameter in generic function or generic lambda. — *end example*]

4.1.3.2 The meaning of constrained type specifiers [dcl.constr.meaning]

1 The meaning of a constrained type specifier depends on the context in which it is used. The different meanings of constrained type specifiers are enumerated in this clause.

2 If a *constrained-type-name* is used as the *type-specifier* of a *template-parameter*, the constraint is formed by applying the declared parameter to the *constrained-type-name*.

3 When a *constrained-type-name* is used as part of the *type-specifier* of a *parameter-declaration*, the parameter's type is formed by replacing the *constrained-type-name* with `auto`, creating a *generic function* or *generic lambda*. The introduced constraint is formed by applying the to the invented template parameter to the *constrained-type-name*.

4 When a *constrained-type-name* is used as part of a *type-specifier* in a *result-type-requirement*, the constraint is introduced as a *nested-requirement* that applies the *constrained-type-name* to the result type of the required expression.

4.1.4 The 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*.

2 Every concept definition is also a `constexpr` declaration ??.

3 A function concept has the following restrictions:

- The template must be unconstrained.
- The result type must be `bool`.
- The declaration may have no function parameters.
- The declaration must be defined.
- The function shall not be recursive.
- The function body shall consist of a single `return` statement whose expression is a *constraint-expr*.

[*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 function template
```

— *end example*]

4 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-expr*.

[*Example:* template<typename T> concept bool Integral = is_integral<T>::value; // OK
template<typename T> concept bool C = 3 + 4; // Error: initializer is not a constraint
template<Integral T> concept bool D = is_unsigned<T>::value; // Error: constrained concept definition — *end example*]

5 If a program declares a non-concept overload of a concept definition with the same template parameters and no function parameters, the program is ill-formed. [*Example:*

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

template<Graph G>
  constexpr bool Totally_ordered() // error: subverts concept definition
  { return true; }
```

— *end example*]

6 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*]

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

5 Declarators [dcl.decl]

5.1 Meaning of declarators [dcl.meaning]

5.1.1 Functions [dcl.fct]

Add the following paragraphs.

15 A *generic function* is denoted by function declarator having `auto` or a *concept-name* as part of the *type-specifier* in its *parameter-declaration-clause*. [*Example:*

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

— *end example*]

16 The use of `auto` or a *concept-name* in the *parameter-declaration-clause* shall be interpreted as the use of a *type-parameter* having the same constraints and the named concept. [*Note:* The exact mechanism for achieving this is unspecified. — *end note*] [*Example:* The generic function declared below

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

Is equivalent to the following declaration

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

— *end example*]

17 All placeholder types introduced using the same *concept-name* have the same invented template parameter. [*Example:* The generic function declared below

```
auto gcd(Integral a, Integral b);
```

Is equivalent to the following declaration:

```
template<Integral T>
  auto gcd(T a, T b);
```

— *end example*]

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

6 Templates [temp]

1 A *template* defines a family of classes or functions or an alias for a family of types.

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

requires-clause:
  `requires` *constraint-expression*

Add the following paragraphs.

7 The *requires-clause* introduces the following *constraint-expression* as an *associated constraint* of the *template-declaration*.

8 A *constrained template declaration* is a *template-declaration* with *associated constraints*. The associated constraints of a constrained template declaration are the conjunction of the associated constraints of all *constrained-parameter*s in the *template-parameter-list* (6.1) and a *constraint-expression* introduced by a

requires-clause, if present. 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*]

9 A constrained template declaration's associated constraints must be satisfied (7.3) to allow instantiation of the constrained template. Class template and alias template constraints are checked during name lookup (6.2); function template constraints and class template partial specialization constraints are checked during template argument deduction (**??**).

6.1 Template parameters [temp.param]

1 The syntax for *template-parameter*s is:

template-parameter:
  type-parameter
  parameter-declaration
  constrained-parameter

constrained-parameter:
  constraint-id ...$_{opt}$ *identifier*
  constraint-id ...$_{opt}$ *identifier* = *constrained-default-argument*

constraint-id:
  concept-name
  partial-concept-id

constrained-default-argument:
  type-id
  template-name
  expression

Add the following paragraphs.

16 A *constrained-parameter* is introduced by a *constraint-id*, which is either a *concept-name* or a *partial-concept-id*. The template parameter introduced by the *constraint-id* has the same kind as the first template parameter, called the *prototype parameter*, of the named concept.

17 If the prototype parameter is a *type-parameter* that is a `class` or `typename`, then introduced template parameter shall be `class` or `typename` parameter.

18 If the prototype parameter is a *template-declaration*, then the introduced parameter shall be a *template-declaration* having the same number of kinds of template parameters.

19 If the prototype parameter is a *parameter-declaration*, then the introduced parameter shall be a *parameter-declaration* with the same *type-specifier*.

20 If prototype parameter is a parameter pack, then the constrained parameter shall also be declared as a parameter pack. [*Example:*

```
template<typename... Ts>
  concept bool Same_types() { ... }

template<Same_types Args> // error: Must be Same_types...
  void f(Args... args);
```

— *end example*]

21 The associated constraints introduced by the *constraint-id* are formed by applying the *concept-name* to that parameter. If the *constraint-id* is a *partial-concept-id*, then the supplied *template-arguments* follow the declared parameter in the application. [*Example:*

```
template<Input_iterator I, Equality_comparable<Value_type<I>> T>
  I find(I first, I last, const T& value);
```

The constraints formed from these constrained template parameters are equivalent to the following declaration:

```
template<typename I, typename T>
  requires Input_iterator<I>() && Equality_comparable<T, Value_type<I>>()
    I find(I first, I last, const T& value);
```

— *end example*]

22 The kind of *constrained-default-argument* shall match the kind of parameter introduced by the *constrained-id*.

6.2 Template names [temp.names]

Modify paragraph 6.

6 A *simple-template-id* that names a class template specialization is a *class-name* provided that the *template-arguments* satisfy the associated constraints (7.3) (if any) of the referenced primary template. Otherwise the program is ill-formed. [*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*]

6.3 Template arguments [temp.arg]

6.3.1 Template template arguments [temp.arg.template]

Modify paragraph 3.

3 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` given that `Q` is at least as constrained (7.1) as `A`. [*Example:*

```
// ... from standard

template<template<Copyable>> class C>
  class stack { ... };

template<Regular T> class list1;
template<Object T> class list2;

stack<list1> s1; // OK: Regular is more strict than Copyable
stack<list2> s2; // error: Object is not more strict than Copyable
```

— *end example*]

6.4 Template declarations [temp.decls]

6.4.1 Class templates [temp.class]

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

Add the following paragraphs.

6 A member function of a class template can be constrained by writing a *requires-clause* after the member declarator. [*Example:*

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

— *end example*] The *requires-clause* introduces the following *expression* as an associated constraint of the member function. A member function of a class template with an associated constraint is a *constrained member function*.

7 A constrained member function's associated constraints are evaluated during overload resolution, not during class template instantiation. [*Note:* Member function constraints do not affect the declared interface of a class. This means that the rules for synthesizing default constructors are unaffected by the presence of constrained constructors and assignment operators. Constraints on member functions are evaluated during overload resolution. — *end note*]

8 During overload resolution, if a candidate member function is an instantiation of a constrained member function template, then those constraints must be satisfied (7.3) before it is considered viable. Constraints are checked by substituting the template arguments of member function's corresponding class template specialization into the associated constraints of the constrained member function template and evaluating the results.

6.4.2 Friends [temp.friend]

Add the following paragraphs.

10 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:* All of the following are valid constrained friend declarations:

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

    template<Object W>
      friend struct Z { };

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

— *end example*]

11 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*]

12 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) { } // OK: does not declare a specialization
  };
```

— *end example*]

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

6.4.3 Class template partial specializations [temp.class.spec]

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

Modify paragraph 2.

A partial specialization matches a given actual template argument list if the template arguments of the partial specialization can be deduced from the actual template argument list (14.8.2), and the deduced template arguments satisfy the constraints of partial specialization, if any (??).

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

Modify paragraph 1.

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

— 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.

New text.

[*Example:*

```
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 `A` will be more specialized than `B`. — *end example*]

6.4.4 Function templates [temp.fct]

6.4.4.1 Function template overloading [temp.over.link]

Modify paragraph 6.

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

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

Modify paragraph 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 both deductions succeed, the the more specialized template is the one that is whose constraints are more strict (??).

6.5 Concept introductions [con.intro]

Add this section as 14.9.

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

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

concept-introduction:
concept-name `{` *introduction-list* `}`

introduction-list:
identifier
introduction-list `,` *identifier*

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

```
template<typename I1, typename I2, typename O>
  concept 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*]

3 The *introduced parameters* are template parameters of the trailing declaration. The kind and type of those parameters are the same as the template parameters of the named concept. The declaration is constrained by applying the introduced arguments to the named concept to form a new constraint 4.1.3.1. [*Example:* The following declaration

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

is equivalent this the declaration below.

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

— *end example*]

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

7 Constrained Declarations [con.decl]

1 A *constrained declaration* is a *constrained-template-declaration*, a *constrained-parameter*, or a *constrained-member-function*. A declaration without associated constraints is an *unconstrained declaration*.

7.1 Partial ordering of constrained declarations [con.decl.order]

1 A declaration `D1` is *more constrained* than another declaration `D2` when both declarations are of the same

kind and have equivalent types, and the associated constraints of `D1` are *stricter* (3.2) than those of `D2`. A constrained declaration is more constrained than an unconstrained declaration of the same kind and equivalent type.

7.2 Equivalence of declaration constraints [con.decl.equiv]

1 Two declarations of the same kind and equivalent type are *equivalently constrained* when their constraints are equivalent (3.2), or when both declarations are unconstrained.

7.3 Constraint satisfaction [con.sat]

A template's constraints are satisfied if the `constexpr` evaluation of the associated constraints results in `true`.