Document Number: N3929
Date: 2014-02-14
Reply to: Andrew Sutton
University of Akron
asutton@uakron.edu

Technical Specification: 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 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 the types associated with those expressions. 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

— allow programmers to directly state the requirements of a set of template arguments as part of a template's interface,

— support 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, when comparing similar programs using `enable_if`.

"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 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.

6 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. Notes in this Technical Specification indicate how and where new text should be added to or removed from the International Standard. [*Note:* The proposal is written against the C++14 specification, whatever its final document number may be. — *end note*]

1.2 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++

1.3 Terms and Definitions [intro.defs]

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

1.3.1 [defns.atomic_constr]
atomic constraint
A subexpression of a constraint that is not a *logical-and-expression*, *logical-or-expression*, or a concept check.

1.3.2 [defns.assoc_constr]
associated constraints
A conjunction of all constraints on a constrained template declaration that includes constraints on template parameters, constraints on function parameters, and constraints specified explicitly in a *requires-clause*.

1.3.3 [defns.concept]
concept
A template declared with the `concept` declaration specifier.

1.3.4 [defns.concept_check]
concept check
A call to a function concept or a *template-id* that names a variable concept.

1.3.5 [defns.constraint]
constraint
A constant expression with type `bool` that evaluates properties of template arguments, determining whether or not they can be substituted into a template.

1.3.6 [defns.constrained_decl]
constrained declaration
A declaration with associated constraints.

1.3.7 [defns.constr_expr]
constraint expression
A constant expression that evaluates requirements of a template argument.

1.3.8 [defns.generic_fn]
generic function
A function declaration having `auto` or a *constrained-type-name* in the type specifier any of its parameters.

1.3.9 [defns.intro_param]
introduced parameters

A sequence of template parameters that are introduced into the scope of a template declaration by a *concept-introduction*.

1.4 Conformance [intro.conform]

Conformance requirements for this specification are the same as those of ISO/IEC 14882. [*Note:* Conformance is defined in terms of the behavior of programs. — *end note*]

1.5 Acknowledgements [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 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 Lambda expressions [expr.lambda]

Modify paragraph 5.

5 The closure type for a non-generic *lambda-expression* has a public `inline` function call operator (ISO/IEC 14882 13.5.4) whose parameters and return type are described by the *lambda-expression*'s *parameter-declaration-clause* and *trailing-return-type* respectively. For a generic lambda, the closure type has a public inline function call operator member template (ISO/IEC 14882 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 (ISO/IEC 14882 8.3.5). The associated constraints of the generic lambda are the conjunction of constraints introduced by the use of *constrained-type-names* 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*.

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

3.1.3 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 `;`
    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` *type-id*
    typename-specifier

nested-requirement:
    requires-clause

2 A *requires-expression* has type `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 A *requires-expression* shall not appear outside a template declaration.

4 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.

5 The *requirement-parameter-list* shall not include an ellipsis.

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

3.1.3.1 Simple requirements [expr.req.simple]

1 A *simple-requirement* introduces a requirement that the substitution of template arguments into the expression will not result in a substitution failure. A *simple-requirement* has the value `true` if and only if substitution succeeds. The expression is an unevaluated operand (ISO/IEC 14882 3.2). [*Example:*

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

— *end example*]

3.1.3.2 Compound requirements [expr.req.compound]

1 A *compound-requirement* introduces a set of constraints pertaining to a single *expression*. A *compound-requirement* is `true` if and only if the substitution of template arguments into the expression does not result in a substitution failure and all other associated requirements are `true`. The expression is an unevaluated operand except in the case when the `constexpr` specifier is present. These other requirements are described in the following sections.

2 The presence of a *trailing-return-type* denotes a result type requirement. The result type requirement is `true` if and only if the substitution of template arguments into the specified type, and `decltype((e))` is convertible to `T`, where `e` is the substituted expression and `T` is the substituted type (ISO/IEC 14882 4). [*Example:*

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

The concept is `true` when the expression `*p` and the type name `T::reference` do not result in substitution failures when template arguments are substituted, and `decltype((*p))` is convertible to `T::reference` after substitution. — *end example*]

3 If the *trailing-return-type* is a *constrained-type-name*, then that concept is applied to the `decltype((e))` where `e` is the substituted expression. The requirement is `true` if and only if the result of that application is `true`. [*Example:*

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

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

The concept is to `true` iff the expression `begin(x)` is a valid expression, and `Iterator<decltype((begin(x)))>()` is `true`. — *end example*]

4 If the `constexpr` specifier is present in the *compound-requirement*, the requirement is true if and only if the substituted expression is a constant expression. [*Example:*

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

When substituted into, the concept is `true` only when the nested `value` member and function call operator must be constant expressions. Otherwise, the concept is `false`. — *end example*]

5 If the `noexcept` specifier is present, in the *compound-requirement* the requirement is `true` when `noexcept(e))` is `true`, where `e` is the substituted expression. [*Example:*

```
template<typename T>
  concept bool Nothrow_movable() {
```

```
    return requires (T x) {
      {T(std::move(x))} noexcept;
      {x = std::move(x)} noexcept -> T&;
    };
  }
```

When template arguments are substituted into the requirement, the move constructor and move assignment operator selected by overload resolution must not propagate exceptions. If either of the instantiated expressions does propagate exceptions, the concept is not satisfied. *— end example*]

3.1.3.3 Type requirements [expr.req.type]

1 A *type-requirement* introduces a requirement that an associated *type-id* can be formed when template arguments are substituted into the type. The requirement is `true` if and only if substitution does not result in a substitution failure. [*Note:* The `typename` may be part of a *typename-specifier*. *— end note*]

```
template<typename T>
  concept bool Input_range() {
    return requires(T range) {
      typename T::value_type;     // Required typename-specifier
      typename Iterator_type<T>; // Required alias template
    };
  }
```

— end example]

3.1.3.4 Nested requirements [expr.req.nested]

1 A *nested-requirement* introduces additional constraints to be evaluated as part of the *requires-expression*. The requirement is `true` if and only if the required 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]

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

3 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 When an identifier is a *concept-name*, it refers to a function concept or variable concept (4.1.4).

2 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 in the *type-specifier* of template parameters, a *result-type-requirement* in a *compound-requirement*, or wherever the `auto` specifier is used, except

— as part of the type of a variable declaration,

— as part of a function's result type or *trailing-result-type*,

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

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

3 A *constrained-type-name* that refers to a non-type concept shall not be used as part of a *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 When a *template-parameter* or parameter-declaration is declared using a constrained-type-name in its type-specifier, a new constraint expression is synthesized and associated with the template declaration. The rules for forming that constraint depend on whether the type specifier is a *concept-name* or *partial-concept-id*. Both cases require the synthesis of a *template-id* referring that refers to a specialization of the named concept. The template argument list is formed using the following rules.

2 Letting `X` be the declared *template-parameter* or the invented type of a *parameter-declaration* in a generic function or generic lambda:

- If the *constrained-type-name* is a *concept-name*, the synthesized template argument list contains only `X`.
- If the *constrained-type-name* is a *partial-concept-id* whose template argument list contains the arguments `Y1`, `Y2`, ..., `Yn`, the synthesized template argument list contains the sequence `X`, `Y1`, `Y2`, ..., `Yn`.

3 If the *constrained-type-name* refers to a function concept, then the synthesized constraint is a call expression with no function arguments.

4 [*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*]

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.

- 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*.

— 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 declared parameter to the invented template parameter to the *constrained-type-name*.

— 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 (ISO/IEC 148827.1.5).

3 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 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 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 differs from the one computed by evaluating its constraints. — *end note*]

5 Declarators [dcl.decl]

Modify paragraph 4.

declarator:
ptr-declarator
noptr-declarator parameters-and-qualifiers trailing-return-type *requires-clause*$_{opt}$

Add the following paragraphs.

6 A declarator having a *requires-clause* is a *constrained declaration*. A declarator that declares a constrained variable or type is ill-formed.

5.1 Meaning of declarators [dcl.meaning]

5.1.1 Functions [dcl.fct]

Add the following paragraphs.

15 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(Sortable& c); // Ok (assuming Sortable names a concept)
```

— *end example*]

16 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 (ISO/IEC 14882 8.3.5). The associated constraints of the generic function are the conjunction of constraints introduced by the use of *constrained-type-name* s in the *parameter-declaration-clause*.
[*Example:* The generic function declared below

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

Is equivalent to the following declaration

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

— *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 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 all *constraint-expression*s introduced by *requires-clause* in the *template-declaration* and subsequent *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*]

8 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 (6.5) 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 (6.2); function template constraints and class template partial specialization constraints are checked during template argument deduction (6.4.4.1).

10 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 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* defines its identifier to be a template parameter. The declared template parameter matches that of the first template parameter, called the *prototype parameter*, of the concept referred to by the *constraint-id*. [*Note:* The rules for declaring the parameter are:

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

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

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

— *end note*]

17 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*]

18 The associated constraints of the constrained template parameter are synthesized according to the rules defined in 4.1.3.1.

19 The kind of *constrained-default-argument* shall match that of the declared *constrained-parameter*.

6.2 Template names [temp.names]

Modify paragraph 6.

6 A *simple-template-id* that names a class template specialization is a *class-name*. The template-arguments shall satisfy the associated constraints of the primary template, 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*]

6.3 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`. The associated constraints of `P` shall subsume the associated constraints of `A` (6.5). [*Example:*

```
template<typename T>
  concept bool Object = is_object<T>::value;
template<typename T>
  concept bool Copyable = is_copyable<T>::value;
template<typename T>
  concept bool Regular = Copyable<T> && ...;

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

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

stack<list1> s1; // error: Regular is more strict than Copyable
stack<list2> s2; // Ok: 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 Constraints on member functions are instantiated as needed during overload resolution (ISO/IEC 14882 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 all instantiations of the class. — *end note*]

8 A destructor shall not be constrained.

9 During overload resolution, if a candidate member function is an instantiation of a constrained member function template, then those constraints must be satisfied (6.5) 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 substituted expression. If the result of that evaluation is `bool`, then member function is a viable candidate.

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) requires C<T>() { } // 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.

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 (ISO/IEC 14882 14.8.2), and the deduced template arguments satisfy the constraints of the partial specialization, if any (6.5).

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

Modify paragraph 1.

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 (ISO/IEC 14882 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>
  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*]

6.4.4 Function templates [temp.fct]

6.4.4.1 Template argument deduction [temp.deduct]

Modify paragraph 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 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.

6.4.4.2 Function template overloading [temp.over.link]

Modify paragraph 1.

1 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 (14.8.2), ~~and~~ checking of any explicit template arguments (14.3), and checking of associated constraints 6.5 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 13.3.3.

Modify paragraph 6.

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, and constraints (6.5) that are equivalent using the rules described above to compare expressions involving template parameters.

6.4.4.3 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 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. (6.5). A constrained template is always selected over an unconstrained template.

6.5 Template constraints [temp.constr]

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:
    logical-or-expression

2 A *logical-or-expression* is a *constraint-expression* if it is a *constant-expression* of type `bool` whose operands to logical operators in its sub-expressions, when substituted have type `bool`. [*Note:* The required contextual conversion to `bool` in a *constraint-expression* prevents user-defined overloads of logical operators from being selected during overload resolution. — *end note*] [*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);

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 4.1.4. is a *concept check*. When processing a constraint containing a concept check, that concept check is replaced by the concept's definition, forming a new expression. For checks against function concepts, the replacement is done by substituting the explicit template arguments into the return expression. For checks against 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 Given two constraints `P` and `Q` depending on template parameters `T1`, ..., `Tn`, `P` is said to *subsume* `Q` if for any template arguments substituted for `T1`, ..., `Tn`, the constraint `Q` is true, then `P` is also true. [*Example:* Let `P` be the constraint `is_integral<T>::value && sizeof(T) == 4`, and let `Q` be the constraint `sizeof(T) == 4`. Then `P` subsumes `Q`, but `Q` does not subsume `P`. — *end example*]

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

8 A constraint is *satisfied* if it evaluates to `true`. Otherwise, the constraint is *unsatisfied*.

6.6 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 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 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*]

3 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 4.1.3.1. [*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*]

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

5 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, unusable template parameter in the template declaration. — *end note*]