Concepts Lite Specification

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

Contents

1 General 1
   1.1 Introduction ................................................................. 1
   1.2 Scope .......................................................................... 1
   1.3 Normative References ...................................................... 2
   1.4 Terms and Definitions ..................................................... 2
   1.5 Conformance ................................................................. 2
   1.6 Acknowledgements .......................................................... 2

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

3 Expressions 2
   3.1 Primary expressions ......................................................... 2

4 Templates 5
   4.1 Template parameters ....................................................... 6
   4.2 Template names .............................................................. 6
   4.3 Template arguments ........................................................ 7
   4.4 Template declarations ...................................................... 7

5 Constrained Declarations 10
   5.1 Partial ordering of constrained declarations ......................... 10
   5.2 Equivalence of declaration constraints .................................. 10
   5.3 Constraint satisfaction ..................................................... 10
   5.4 Constraint ordering ......................................................... 10
   5.5 Constraint equivalence ..................................................... 12
   5.6 Concept introduction ....................................................... 12
1 General

1.1 Introduction

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.

A concept is a predicate that expresses a set of requirements on types. These requirements consist of syntactic requirements, which what related types, literals, operations, and expressions are available, and semantic requirements that give meaning to the required syntax and also provide complexity guarantees. Concepts are the basis of generic programming in C++ and allow us to write and reason about generic algorithms and data structures by constraining template arguments.

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 (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 in ISO/IEC 14882 gives the definition of the STL *Iterator* concept as a list valid expressions and their result types, operational semantics, and pre- and post-conditions.

- 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.

This specification describes a solution to the problem of constraining template arguments in the form “Concepts Lite.” Constraints are defined as constexpr functions and evaluate compile time properties of types and values. A small amount of additional syntax is provided to make the specification of constraints and the description of syntactic requirements easier.

The design of this specification is based in part of a specification of the algorithms part of the C++ standard library (known as “The Palo Alto” TR (WG21 N3350) 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

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. The Technical Specification also describes library requirements for a core set of constraints related to the C++ type system and closely related to the C++ standard type traits ??.

International Standard, ISO/IEC 14882, provides important context and specification for this Technical Specification. Clause 2 of this Technical Specification should be read as if merged into Clause 2 of ISO/IEC
Clause 3 of this specification should be read as if merged into Clause 5 of ISO/IEC 14882. Clause 4 of this Technical specification should be read as if merged into Clause 14 of ISO/IEC 14882. Clause 5 of this specification should be added as Section 14.9 in ISO/IEC 14882.

1.3 Normative References

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

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

1.4.1 constraint
A template declaration declared with the concept declaration specifier.

1.4.2 constraint
An atom in the constraints language, represented as a constant Boolean expression.

1.4.3 constraint
An expression denoting sets of constraints.

1.5 Conformance

Conformance is specified in terms of behavior.

1.6 Acknowledgements

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

2 Lexical conventions

2.1 Keywords

Add to table 4, the keywords concept and requires.

3 Expressions

3.1 Primary expressions

3.1.1 General

Modify the grammar of primary-expression

```
primary-expression::=
  literal
  .
  this
  ...
  requires-expression
```
Add the following sections.

### 3.1.2 Requires expressions

1 A requires expression provides a concise way to express syntactic requirements for 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:
     constraint-specifier-seq result-type-requirement_opt
     constraint-specifier-seq:
       constexpr_opt noexcept_opt
     result-type-requirement:
       -> type-id
   ```

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

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

   The return expression of the is a `requires` expression and the statements written within the enclosing braces denote specific interface 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 as notation for the purpose of writing 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.

§ 3.1.2
A requires-expression evaluates to true if and only if each 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

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

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

—end example]

3.1.2.2 Compound requirements

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

1 If a result-type-requirement is present then a) instantiation of the result type must succeed and b) the result type of the instantiated expression must be convertible to that type. [Example:

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

The concept evaluates to true iff the expression *p can be instantiated, the type T::reference can be instantiated to a type, and decltype(*p) is convertible to T::reference when instantiated. —end example]

2 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 instantiate expression. [Example:

```cpp
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 Iterator<decltype(begin(x))>() evaluates to true. —end example]

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

```cpp
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. Otherwise, the concept is not satisfied. —end example]
If the `noexcept` specifier is present, in the `constraint-specifier-seq` the instantiated expression must not propagate exceptions. [Example:

```cpp
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 not, 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 only if instantiation succeeds.

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`.

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

— end example]

4 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:
    constant-expression
```

Add the following paragraphs.

7 The `requires-clause` introduces the following `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` (4.1) and an expression introduced by a `requires-clause`. The associated constraints of a `concept-introduction` are those required by the reference concept declaration. [Example:

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

```cpp
template<Integral T>
requires Unsigned<T>()
    T binary_gcd(T a, T b);
```
A constrained template declaration’s associated constraints must be satisfied (5.3) before its corresponding definition is instantiated. Class template and alias template constraints are checked during name lookup (4.2), function template constraints and class template partial specialization constraints are checked during template argument deduction (??).

### 4.1 Template parameters

The syntax for template-parameters 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
```

A constrained-parameter is introduced by a constraint-id, which is either a concept-name or a partial-concept-id. The concept declaration referred to by the constraint-id determines the kind of template parameter. The template parameter introduced by the constraint-id has the same kind as the first template parameter of the concept declaration. If that template 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]

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]

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

### 4.2 Template names

Modify paragraph 6.
A simple-template-id that names a class template specialization is a class-name provided that the template-arguments satisfy the associated constraints (5.3) (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. The enforcement of this requirement is done by checking the primary template during name lookup rather than enforcing the requirement when the partial specialization is declared. — end note]

4.3 Template arguments [temp.arg]

4.3.1 Template template arguments [temp.arg.template]

Modify paragraph 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 more constrained (5.1) than 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]

4.4 Template declarations [temp.decls]

4.4.1 Class templates [temp.class]

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

Add the following paragraphs.

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 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.

The member function’s associated constraints are not not evaluated 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]
During overload resolution, if a member function candidate is an instantiation of a constrained member function template, then those constraints must be satisfied (5.3) 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.

### 4.4.2 Friends

Add the following paragraphs.

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:

```c
    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;
        } 
`]

11 A non-template friend function may not be constrained if the function’s parameter types or result type are not dependent on any template parameters. [Example:

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

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

```c
    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
    }
`]

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

### 4.4.3 Class template partial specializations

#### 4.4.3.1 Matching of class template partial specializations

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.

4.4.3.2 Partial ordering of class template specializations

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.

Example:

```cpp
template<typename T> class S { };  // #1
template<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 #2 will be more specialized than #1.

— end example

4.4.4 Function templates

4.4.4.1 Function template overloading

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, have return types and parameter lists that are equivalent using the rules described above to compare expressions involving template parameters, and have equivalent constraints.

4.4.4.2 Partial ordering of function templates

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 more specialized template is the one that is whose constraints are more strict.

4.4.5 Abbreviated template declaration

Add the following paragraphs.

The name of a unary concept may be used as a simple-type-specifier in a function or constructor declaration, either as part of parameter-declaration or as part of the return type. In that case, that declaration is considered a template declaration declaring the function or the constructor. This is called an abbreviated template declaration. The use of the concept name shall be interpreted as a use of a template parameter satisfying the kind and requirements of the concept. [Note: The exact mechanism for achieving this is unspecified.]

Example: The following declaration

§ 4.4.5
void sort(Sortable& c);

is equivalent to

```cpp
template<Sortable __Sortable>
void sort(__Sortable& c);
```

—end example —end note

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

5 Constrained Declarations [con.decl]

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.

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

One declaration $D_1$ is more constrained than another $D_2$ when both declarations are of the same kind and have equivalent type and the associated constraints of $D_1$ are more strict than those of $D_2$. A constrained declaration is more constrained than an unconstrained declaration of the same kind and equivalent type.

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

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

Before a constrained template is instantiated, its associated constraints must be satisfied [5.3].

The partial ordering of constrained function templates and the partial ordering of constrained class template specializations rely on the partial ordering those template declaration’s constraints. A template declaration is more constrained than another if its associated constraints are more strict.

5.3 Constraint satisfaction [con.sat]

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

5.4 Constraint ordering [con.order]

Partial ordering of constraints is used to choose among template specializations during the partial ordering of function templates, the partial ordering of class templates, and the use of template template arguments. This computation of this ordering is modeled as a propositional deduction by holding one constraint, $P$, as an assumption and determining if another constraint, $Q$, can be inferred from $P$. [Note: In formal logic, the ordering of constraints determines whether $P \vdash Q$ is valid inference, that is, if $P$ implies $Q$. — end note]

A constraint $P$ is more strict than another constraint $Q$ if and only if $P$ implies $Q$ and $Q$ does not imply $P$.

The mechanism of this computation depends on the kind of expression or proposition of $P$ and $Q$. There are four kinds of propositions in the constraints language.

— conjunctions – logical AND expressions

— disjunctions – logical OR expressions

— concept checks – function calls to `concept`-declared functions

— requirements – `requires` expressions

— atomic propositions – every other constant expression

The following sections describe semantics for the different kinds of propositions and their derivation rules.
5.4.1 Atomic Propositions  \[\text{con.prop.atomic}\]

An atomic proposition is an expression that has no deeper logical meaning in the constraints language; they represent a truth value when evaluated.  [Example: The following are all atomic propositions, assuming that \(M\) and \(N\) are constant expressions (possibly having literal type) and \text{is_prime} is declared constexpr.

\[
\begin{align*}
\text{true} \\
M \&\& N \\
M < N \\
\text{is_prime(N)} \\
\text{std::is_integral<T>::value} \\
\text{std::is_integral<T>()} \\
!\text{std::is_reference<T>::value}
\end{align*}
\]

—end example\]

[Note: The literal values \text{true} and \text{false} do not have meaning within the constraints language. A template constrained by only requires \text{true} is not equivalent to an unconstrained template (\text{true} is a constraint). A template constrained by requires \text{false} is allowed, even though it will never be selected by overload resolution. —end note]\n
[Note: The logical ! expression is not in the constraints language because it has multiple interpretations. It can mean both the negation of a proposition (it must not be the case that) and the negation of a requirement (it is not required that) depending on context and operand. By making it atomic, it can only have the former meaning. —end note]\n
An atomic proposition \(P\) implies an atomic proposition \(Q\) if and only if both propositions have the same spelling.

If \(Q\) is a conjunction, \(A \&\& B\), then \(P\) must imply \(A\) and \(P\) must imply \(B\).

If \(Q\) is a disjunction, \(A \|\| B\), then either \(P\) must imply \(A\) or \(P\) must imply \(B\).

5.4.2 Requirements  \[\text{temp.prop.req}\]

A requires expression denotes a conjunction of syntactic, type, and nested requirements. An the context of a constraint, a requires expression is replaced by its conjunction of requirements.  [Note: The syntactic representation of expressions checking syntactic and type requirements is implementation-specific. However, each such valid expression and valid type requirement is an atomic proposition. —end note]\n
5.4.3 Conjunctions  \[\text{con.prop.conj}\]

A conjunction is a logical AND expression. User-defined \&\& operators are not found in the resolution of disjunctions within a constraint.

A conjunction of the form \(P \&\& Q\) implies a proposition \(R\) if and only if \(P\) implies \(R\) or \(Q\) implies \(R\).

If \(R\) is a conjunction of the form \(A \&\& B\), then either \(P\) or \(Q\) must imply \(A\), and either \(P\) or \(Q\) must imply \(B\).

If \(R\) is a disjunction of the form \(A \|\| B\), then either \(P\) or \(Q\) must imply \(A\), or either \(P\) or \(Q\) must imply \(B\).

5.4.4 Disjunction  \[\text{con.con.disj}\]

A disjunction is a logical OR expression. User-defined \|\| operators are not found in the resolution of disjunctions within a constraint.

A disjunction of the form \(P \|\| Q\) implies a proposition \(R\) if and only if \(P\) matches \(R\) and \(Q\) matches \(R\).

If \(R\) is a conjunction of the form \(A \&\& B\), then \(P\) must imply either \(A\) or \(B\), and \(Q\) must imply either \(A\) or \(B\).

If \(R\) is a disjunction of the form \(A \|\| B\), then either \(P\) or \(Q\) must imply \(A\), or either \(P\) or \(Q\) must imply \(B\).

5.4.5 Concept Checks  \[\text{temp.con.prop.check}\]

A concept check is a function call expression to function template declared with the concept specifier. Argument dependent lookup is not used in the resolution of concept checks. After lookup, a concept check is replaced by the instantiated expression returned by the resolved concept definition.  [Note: Replacing

§ 5.4.5

11
a concept check with its definition means that there no “concept check propositions” in the associated constraints of a template. — end note]

5.5 Constraint equivalence

The equivalence of constraints is used to determine whether function declarations are redeclarations or overloads and whether constrained partial template specializations are redeclarations or new declarations. Two constraints \( P \) and \( Q \) are equivalent if and only if \( P \) implies \( Q \) and \( Q \) implies \( P \).

5.6 Concept introduction

A concept-introduction introduces a list of template parameters for template declaration or lambda expression. A concept-introduction:

\[
\text{concept-name} \{ \text{introduced-parameter-list} \}
\]

The concept name is matched, based on the number of introduced parameters to a corresponding concept definition. If no such concept can be found, the program is ill-formed.

The kind of each introduced parameter (type, non-type, template), is the same as the corresponding template parameter in the matched concept definition. The concept is applied introduced parameters as a constraint on the trailing declaration. [Example:

\[
\text{template<typename I1, typename I2, typename O>}
\text{concept bool Mergeable() \{ ... \}}
\]

Mergeable\{A, B, C\}

\(O\) merge\(\{A\ \text{first1, A last1, B first2, B last2 C out}\};

A, B, and C are introduced as type parameters. The constraint on the algorithm is \(\text{Mergeable}<A, B, C>()\). The declaration is equivalent to:

\[
\text{template<typename A, typename B, typename C>}
\text{requires Mergeable\{A, B, C\}()}
\text{O merge\{A first1, A last1, B first1, B first2, C out\};}
\]

— end example]