TECHNICAL SPECIFICATION

ISO/IEC TS 19217

First edition
2015-11-15

Information technology — Programming languages — C++ Extensions for concepts

Technologie de l'information — Langages de programmation — Extensions C++ pour les concepts warning

Reference number
ISO/IEC TS 19217:2015(E)

Contents

List of Tables

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation on the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the WTO principles in the Technical Barriers to Trade (TBT) see the following URL: Foreword - Supplementary information

The committee responsible for this document is ISO/IEC JTC 1, *Information technology*, SC 22, *Programming languages, their environments and system software interfaces.*

1 General [intro]

1.1 Scope [intro.scope]

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

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

3 WG21 paper N4191 defines "fold expressions", which are used to define constraint expressions resulting from the use of *constrained-parameter*s that declare template parameter packs. This feature is not present in ISO/IEC 14882:2014, but it is planned to be included in the next revision of that International Standard. The specification of that feature is included in this document.

1.2 Normative references [intro.refs]

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

(1.1) — ISO/IEC 14882:2014, *Programming Languages – C++*

ISO/IEC 14882:2014 is hereafter called the *C++ Standard*. The numbering of Clauses, sections, and paragraphs in this document reflects the numbering in the C++ Standard. References to Clauses and sections not appearing in this Technical Specification refer to the original, unmodified text in the C++ Standard.

1.3 Terms and definitions [intro.defs]

Modify the definitions of "signature" to include associated constraints (14.10.2). This allows different translation units to contain definitions of functions with the same signature, excluding associated constraints, without violating the one definition rule (3.2). That is, without incorporating the constraints in the signature, such functions would have the same mangled name, thus appearing as multiple definitions of the same function.

1.3.1 [defns.signature]

signature
<function> name, parameter type list (8.3.5), ~~and~~ enclosing namespace (if any), and any associated constraints (14.10.2)
[*Note:* Signatures are used as a basis for name mangling and linking. — *end note*]

1.3.2 [defns.signature.templ]

signature
<function template> name, parameter type list (8.3.5), enclosing namespace (if any), return type, ~~and~~ template parameter list, and any associated constraints (14.10.2)

1.3.3 [defns.signature.member]

signature
<class member function> name, parameter type list (8.3.5), class of which the function is a member, *cv*-qualifiers (if any), ~~and~~ *ref-qualifier* (if any), and any associated constraints (14.10.2)

1.3.4 **[defns.signature.member.templ]**
signature
<class member function template> name, parameter type list (8.3.5), class of which the function is a member, *cv*-qualifiers (if any), *ref-qualifier* (if any), return type, ~~and~~ template parameter list, and any associated constraints (14.10.2)

1.4 Implementation compliance [intro.compliance]

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

1.5 Feature-testing recommendations [intro.features]

1 An implementation that provides support for this Technical Specification shall define the feature test macro(s) in Table A.

Table A — Feature-test macro(s)

Macro name	Value
`__cpp_concepts`	201507

1.6 Acknowledgments [intro.ack]

1 The design of this specification is based, in part, on a concept specification of the algorithms part of the C++ standard library, known as "The Palo Alto" report (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 report and this Technical Specification, the report can be seen as a large-scale test of the expressiveness of this Technical Specification.

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

2 Lexical conventions [lex]

2.1 Keywords [lex.key]

In 2.1, add the keywords `concept` and `requires` to Table 4.

5 Expressions [expr]

Modify paragraph 8 to include a reference to *requires-expressions*.

1 In some contexts, unevaluated operands appear (5.1.4, 5.2.8, 5.3.3, 5.3.7).

5.1 Primary expressions [expr.prim]

5.1.1 General [expr.prim.general]

In this section, add the *requires-expression* to the rule for *primary-expression*.

primary-expression:
  literal
  `this`
  `(` *expression* `)`
  id-expression
  lambda-expression
  fold-expression
  requires-expression

In paragraph 8, add `auto` and *constrained-type-name* to *nested-name-specifier*:

8 *nested-name-specifier:*
  `::`
  type-name `::`
  namespace-name `::`
  decltype-specifier `::`
  `auto ::`
  constrained-type-name `::`
  nested-name-specifier identifier `::`
  nested-name-specifier `template`$_{opt}$ `::`

Add a new paragraph after paragraph 11:

12 In a *nested-name-specifier* of the form `auto::` or `C::`, where `C` is a *constrained-type-name*, that *nested-name-specifier* designates a placeholder that will be replaced later according to the rules for placeholder deduction in 7.1.6.4. If a placeholder designated by a *constrained-type-specifier* is not a placeholder type, the program is ill-formed. [*Note:* A *constrained-type-specifier* can designate a placeholder for a non-type or template (7.1.6.4.2). — *end note*] The replacement type deduced for a placeholder shall be a class or enumeration type. [*Example:*

```
template<typename T> concept bool C = sizeof(T) == sizeof(int);
template<int N> concept bool D = true;

struct S1 { int n; };
struct S2 { char c; };
struct S3 { struct X { using Y = int; }; };

int auto::* p1 = &S1::n; // auto deduced as S1
int D::* p2 = &S1::n;    // error: D does not designate a placeholder type
int C::* p3 = &S1::n;    // OK: C deduced as S1
char C::* p4 = &S2::c;   // error: deduction fails because constraints are not satisfied
```

```
void f(typename auto::X::Y);
f(S1()); // error: auto cannot be deduced from S1()
f<S3>(0); // OK
```

In the declaration of `f`, the placeholder appears in a non-deduced context (14.8.2.5). It may be replaced later through the explicit specification of template arguments. — *end example*]

Add a new paragraph after paragraph 13:

14 A program that refers explicitly or implicitly to a function with associated constraints that are not satisfied (14.10.2), other than to declare it, is ill-formed. [*Example:*

```
void f(int) requires false;

f(0);                         // error: cannot call f
void (*p1)(int) = f;          // error: cannot take the address of f
decltype(f)* p2 = nullptr;    // error: the type decltype(f) is invalid
```

In each case the associated constraints of `f` are not satisfied. In the declaration of `p2`, those constraints are required to be satisfied even though `f` is an unevaluated operand (Clause 5). — *end example*]

5.1.2 Lambda expressions [expr.prim.lambda]

Insert the following paragraph after paragraph 4 to define the term "generic lambda".

5 A *generic lambda* is a *lambda-expression* where one or more placeholders (7.1.6.4) appear in the parameter-type-list of the *lambda-declarator*.

Modify paragraph 5 so that the meaning of a generic lambda is defined in terms of its abbreviated member function template call operator.

The closure type for a non-generic *lambda-expression* has a public inline function call operator (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 (14.5.2) whose *template-parameter-list* consists of one invented type *template-parameter* for each occurrence of `auto` 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 (8.3.5). 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.~~ The closure type for a generic lambda has a public inline function call operator member template that is an abbreviated function template whose parameters and return type are derived from the *lambda-expression*'s *parameter-declaration-clause* and *trailing-return-type* according to the rules in (8.3.5).

Add the following example after those in paragraph 5 in the C++ Standard.

[*Example:*

```
template<typename T> concept bool C = true;

auto gl = [](C& a, C* b) { a = *b; }; // OK: denotes a generic lambda

struct Fun {
  auto operator()(C& a, C* b) const { a = *b; }
} fun;
```

C is a *constrained-type-specifier*, signifying that the lambda is generic. The generic lambda gl and the function object fun have equivalent behavior when called with the same arguments. — *end example*]

5.1.3 Fold expressions [expr.prim.fold]

Add this section after 5.1.2.

1 A fold expression performs a fold of a template parameter pack (14.6.3) over a binary operator.

fold-expression:
(*cast-expression fold-operator* ...)
(... *fold-operator cast-expression*)
(*cast-expression fold-operator* ... *fold-operator cast-expression*)

fold-operator: one of

```
+    -    *    /    %    ^    &    |    <<    >>
+=   -=   *=   /=   %=   ^=   &=   |=   <<=   >>=   =
==   !=   <    >    <=   >=   &&   ||   ,     .*    ->*
```

2 An expression of the form (... *op* e) where *op* is a *fold-operator* is called a *unary left fold*. An expression of the form (e *op* ...) where *op* is a *fold-operator* is called a *unary right fold*. Unary left folds and unary right folds are collectively called *unary folds*. In a unary fold, the *cast-expression* shall contain an unexpanded parameter pack (14.6.3).

3 An expression of the form (e1 *op1* ... *op2* e2) where *op1* and *op2* are *fold-operator*s is called a *binary fold*. In a binary fold, *op1* and *op2* shall be the same *fold-operator*, and either e1 shall contain an unexpanded parameter pack or e2 shall contain an unexpanded parameter pack, but not both. If e2 contains an unexpanded parameter pack, the expression is called a *binary left fold*. If e1 contains an unexpanded parameter pack, the expression is called a *binary right fold*. [*Example:*


```
template<typename ...Args>
bool f(Args ...args) {
  return (true && ... && args); // OK
}

template<typename ...Args>
bool f(Args ...args) {
  return (args + ... + args); // error: both operands contain unexpanded parameter packs
}
```

— *end example*]

5.1.4 Requires expressions [expr.prim.req]

Add this section to 5.1.

1 A *requires-expression* provides a concise way to express requirements on template arguments. A requirement is one that can be checked by name lookup (3.4) or by checking properties of types and expressions.

requires-expression:
  `requires` *requirement-parameter-list*$_{opt}$ *requirement-body*

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

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

requirement-seq:
  requirement
  requirement-seq requirement

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

2 A *requires-expression* defines a constraint (14.10) based on its parameters (if any) and its nested *requirements*.

3 A *requires-expression* has type `bool` and is an unevaluated expression (5). [*Note:* A *requires-expression* is transformed into a constraint in order to determine if it is satisfied (14.10.2). — *end note*]

4 A *requires-expression* shall appear only within a concept definition (7.1.7), or within the *requires-clause* of a *template-declaration* (Clause 14) or function declaration (8.3.5). [*Example:* A common use of *requires-expressions* is to define requirements in concepts such as the one below:

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


A *requires-expression* can also be used in a *requires-clause* as a way of writing ad hoc constraints on template arguments such as the one below:

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

The first `requires` introduces the *requires-clause*, and the second introduces the *requires-expression*. — *end example*] [*Note:* Such requirements can also be written by defining them within a concept.

```
template<typename T>
  concept bool C = requires (T x) { x + x; };

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

— *end note*]

5 A *requires-expression* may introduce local parameters using a *parameter-declaration-clause* (8.3.5). A local parameter of a *requires-expression* shall not have a default argument. Each name introduced by a local parameter is in scope from the point of its declaration until the closing brace of the *requirement-body*. These parameters have no linkage, storage, or lifetime; they are only

used as notation for the purpose of defining *requirements*. The *parameter-declaration-clause* of a *requirement-parameter-list* shall not terminate with an ellipsis. [*Example:*

```
template<typename T>
  concept bool C1() {
    requires(T t, ...) { t; }; // error: terminates with an ellipsis
  }

template<typename T>
  concept bool C2() {
    requires(T t, void (*p)(T*, ...)) // OK: the parameter-declaration-clause of
    { p(t); };                        // the requires-expression does not terminate
  }                                   // with an ellipsis
```

— *end example*]

6 The *requirement-body* is comprised of a sequence of *requirements*. These *requirements* may refer to local parameters, template parameters, and any other declarations visible from the enclosing context. Each *requirement* appends a constraint (14.10) to the conjunction of constraints defined by the *requires-expression*. Constraints are appended in the order in which they are written.

7 The substitution of template arguments into a *requires-expression* may result in the formation of invalid types or expressions in its requirements. In such cases, the constraints corresponding to those requirements are not satisfied; it does not cause the program to be ill-formed. If the substitution of template arguments into a *requirement* would always result in a substitution failure, the program is ill-formed; no diagnostic required. [*Example:*

```
template<typename T> concept bool C =
  requires {
    new int[-(int)sizeof(T)]; // ill-formed, no diagnostic required
  };
```

— *end example*]

5.1.4.1 Simple requirements [expr.prim.req.simple]

simple-requirement:
  expression ;

1 A *simple-requirement* introduces an expression constraint (14.10.1.3) for its *expression*. [*Note:* An expression constraint asserts the validity of an expression. — *end note*]

[*Example:*

```
template<typename T> concept bool C =
  requires (T a, T b) {
    a + b;  // an expression constraint for a + b
  };
```

— *end example*]

5.1.4.2 Type requirements [expr.prim.req.type]

type-requirement:
  `typename` *nested-name-specifier*$_{opt}$ *type-name* ;

1 A *type-requirement* introduces a type constraint (14.10.1.4) for the type named by its optional *nested-name-specifier* and *type-name*. [*Note:* A type requirement asserts the validity of an associated type, either as a member type, a class template specialization, or an alias template. It is not used to specify requirements for arbitrary *type-specifiers*. — *end note*] [*Example:*

```
template<typename T> struct S { };
template<typename T> using Ref = T&;

template<typename T> concept bool C =
  requires () {
    typename T::inner; // required nested member name
    typename S<T>;     // required class template specialization
    typename Ref<T>;   // required alias template substitution
  };
```

— *end example*]

5.1.4.3 Compound requirements [expr.prim.req.compound]

compound-requirement:
 { *expression* } noexcept$_{opt}$ *trailing-return-type*$_{opt}$;

1 A *compound-requirement* introduces a conjunction of one or more constraints for the *expression* E. The order in which those constraints are introduced is:

(1.1) — the *compound-requirement* introduces an expression constraint for E (14.10.1.3);

(1.2) — if the noexcept specifier is present, the *compound-requirement* appends an exception constraint for E (14.10.1.7);

(1.3) — if the *trailing-return-type* is present, the *compound-requirement* appends one or more constraints derived from the type T named by the *trailing-return-type*:

(1.3.1) — if T contains one or more placeholders (7.1.6.4), the requirement appends a deduction constraint (14.10.1.6) of E against the type T.

(1.3.2) — otherwise, the requirement appends two constraints: a type constraint on the formation of T (14.10.1.4) and an implicit conversion constraint from E to T (14.10.1.5).

[*Example:*

```
template<typename T> concept bool C1 =
  requires(T x) {
    {x++};
  };
```

The *compound-requirement* in C1 introduces an expression constraint for x++. It is equivalent to a *simple-requirement* with the same *expression*.

```
template<typename T> concept bool C2 =
  requires(T x) {
    {*x} -> typename T::inner;
  };
```

The *compound-requirement* in C2 introduces three constraints: an expression constraint for *x, a type constraint for typename T::inner, and a conversion constraint requiring *x to be implicitly convertible to typename T::inner.

```
template<typename T> concept bool C3 =
  requires(T x) {
    {g(x)} noexcept;
  };
```

The *compound-requirement* in C3 introduces two constraints: an expression constraint for g(x) and an exception constraint for g(x).