TECHNICAL SPECIFICATION

ISO/IEC TS 21544

First edition
2018-05

Programming languages — Extensions to C++ for modules

Langages de programmation — Extensions C++ pour les modules

Reference number
ISO/IEC TS 21544:2018(E)

ISO copyright office
Ch. de Blandonnet 8 • CP 401
CH-1214 Vernier, Geneva, Switzerland
Tel. +41 22 749 01 11
Fax +41 22 749 09 47
copyright@iso.org
www.iso.org

Contents

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 voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see the following URL: www.iso.org/iso/foreword.html.

This document was prepared by Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

1 Scope [intro.scope]

1 This document describes extensions to the C++ Programming Language (Clause 2) that introduce modules, a functionality for designating a set of translation units by symbolic name and ability to express symbolic dependency on modules, and to define interfaces of modules. These extensions include new syntactic forms and modifications to existing language semantics.

2 ISO/IEC 14882 provides important context and specification for this document. 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 ISO/IEC 14882 use underlining to represent added text and ~~strikethrough~~ to represent deleted text.

2 Normative references [intro.refs]

1

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements 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 14882:2017, *Programming Languages – C++*

ISO/IEC 14882:2017 is hereafter called the *C++ Standard*. The numbering of clauses, subclauses, and paragraphs in this document reflects the numbering in the C++ Standard. References to clauses and subclauses not appearing in this document refer to the original, unmodified text in the C++ Standard.

3 Terms and definitions [intro.defs]

No terms and definitions are listed in this document.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at http://www.electropedia.org

4 General [intro]

4.1 Implementation compliance [intro.compliance]

1 Conformance requirements for this document are those defined in ISO 14882:2017, 4.1 except that references to the C++ Standard therein shall be taken as referring to the document that is the result of applying the editing instructions. Similarly, all references to the C++ Standard in the resulting document shall be taken as referring to the resulting document itself. [*Note:* Conformance is defined in terms of the behavior of programs. — *end note*]

4.2 Acknowledgments [intro.ack]

1 This document is based, in part, on the design and implementation described in the paper P0142R0 "*A Module System for C++*".

5 Lexical conventions [lex]

5.1 Separate translation [lex.separate]

Modify paragraph 5.1/2 as follows

2 [*Note:* Previously translated translation units and instantiation units can be preserved individually or in libraries. The separate translation units of a program communicate (6.5) by (for example) calls to functions whose identifiers have external or module linkage, manipulation of objects whose identifiers have external or module linkage, or manipulation of data files. Translation units can be separately translated and then later linked to produce an executable program (6.5). —*end note*]

5.2 Phases of translation [lex.phases]

Modify bullet 7 of paragraph 5.2/1 as follows:

7. White-space characters separating tokens are no longer significant. Each preprocessing token is converted into a token (5.6). The resulting tokens are syntactically and semantically analyzed and translated as a translation unit. [*Note:* The process of analyzing and translating the tokens may occasionally result in one token being replaced by a sequence of other tokens (17.2). —*end note*] It is implementation-defined whether the source for module interface units for modules on which the current translation unit has an interface dependency (10.7.3) is required to be available. [*Note:* Source files, translation units and translated translation units need not necessarily be stored as files, nor need there be any one-to-one correspondence between these entities and any external representation. The description is conceptual only, and does not specify any particular implementation. —*end note*]

Add new paragraphs as follows:

2 The result of processing a translation unit from phases 1 through 7 is a directed graph called the *abstract semantics graph* of the translation unit:

- Each vertex, called a *declset*, is a citation (10.7.3), or a collection of non-local declarations and redeclarations (Clause 10) declaring the same entity or other non-local declarations of the same name that do not declare an entity.
- A directed edge (D_1, D_2) exists in the graph if and only if the declarations contained in D_2 declare an entity mentioned in a declaration contained in D_1.

The *abstract semantics graph of a module* is the subgraph of the abstract semantics graph of its module interface unit generated by the declsets the declarations of which are in the purview of that module interface unit. [*Note:* The abstract semantics graphs of modules, as appropriately restricted (10.7.6), are used in the processing of *module-import-declaration*s (10.7.3) and module implementation units. —*end note*]

3 An entity is *mentioned* in a declaration D if that entity is a member of the *basis* of D, a set of entities determined as follows:

- If D is a *namespace-definition*, the basis is the union of the bases of the *declaration*s in its *namespace-body*.
- If D is a *nodeclspec-function-declaration*,
  - if D declares a contructor, the basis is the union of the type-bases of the parameter types

  - if D declares a conversion function, the basis is the type-basis of the return type
  - otherwise, the basis is empty.
- If D is a *function-definition*, the basis is the type-basis of the function's type
- If D is a *simple-declaration*
  - if D declares a *typedef-name*, the basis is the type-basis of the aliased type
  - if D declares a variable, the basis is the type-basis of the type of that variable
  - if D declares a function, the basis is the type-basis of the type of that function
  - if D defines a class type, the basis is the union of the type-bases of its direct base classes (if any), and the bases of its *member-declaration*s.
  - otherwise, the basis is the empty set.
- If D is a *template-declaration*, the basis is the union of the basis of its *declaration*, the set consisting of the entities (if any) designated by the default template template arguments, the default non-type template arguments, the type-bases of the default type template arguments. Furthermore, if D declares a partial specialization, the basis also includes the primary template.
- If D is an *explicit-instantiation* or an *explicit-specialization*, the basis includes the primary template, and all the entities in the basis of the *declaration* of D.
- If D is a *linkage-specification*, the basis is the union of all the bases of the *declaration*s contained in D.
- If D is a *namespace-alias-definition*, the basis is the singleton consisting of the namespace denoted by the *qualified-namespace-specifier*.
- If D is a *using-declaration*, the basis is the union of the bases of all the declarations introduced by the *using-declarator*.
- If D is a *using-directive*, the basis is the singleton consisting of the nominated namespace.
- If D is an *alias-declaration*, the basis is the type-basis of its *defining-type-id*.
- Otherwise, the basis is empty.

The *type-basis* of a type T is

- If T is a fundamental type, the type-basis is the empty set.
- If T is a cv-qualified type, the type-basis is the type-basis of the unqualified type.
- If T is a member of an unknown specialization, the type-basis is the type-basis of that specialization.
- If T is a class template specialization, the type-basis is the union of the set consisting of the primary template and the template template arguments (if any) and the non-dependent non-type template arguments (if any), and the type-bases of the type template arguments (if any).
- If T is a class type or an enumeration type, the type-basis is the singleton $\{T\}$.
- If T is a reference to U, or a pointer to U, or an array of U, the type-basis is the type-basis of U.
- If T is a function type, the type-basis is the union of the type-basis of the return type and the type-bases of the parameter types.
- If T is a pointer to data member of a class X, the type-basis is the union of the type-basis of X and the type-basis of member type.
- If T is a pointer to member function type of a class X, the type-basis is the union of the type-basis of X and the type-basis of the function type.
- Otherwise, the type-basis is the empty set.

4 [*Note:* The basis of a declaration includes neither non-fully evaluated expressions nor entities used in those expressions. [*Example:*


```
const int size = 2;
int ary1[size];                          // size not in ary1's basis
constexpr int identity(int x) { return x; }
int ary2[identity(2)];                   // identity not in ary2's basis

template<typename> struct S;
template<typename, int> struct S2;
constexpr int g(int);

template<typename T, int N>
S<S2<T, g(N)>> f();                      // f's basis: {S, S2}
```

— end example] *— end note*]

5.11 Keywords [lex.key]

In 5.11, add these two keywords to Table 5 in paragraph 5.11/1: module and import.

Modify note in paragraph 5.11/1 as follows:

1 ...

[*Note:* The ~~export and~~ `register` keyword~~s are~~ is unused but ~~are~~ is reserved for future use. *— end note*]

6 Basic concepts [basic]

Modify paragraph 6/3 as follows:

3 An *entity* is a value, object, reference, function, enumerator, type, class member, bit-field, template, template specialization, namespace, module, or parameter pack.

Modify paragraph 6/4 as follows:

4 A *name* is a use of an *identifier* (5.10), *operator-function-id* (16.5), *literal-operator-id* (16.5.8), *conversion-function-id* (15.3.2), ~~or~~ *template-id* (17.2), or *module-name* (10.7) that denotes an entity or *label* (9.6.4, 9.1).

Add a sixth bullet to paragraph 6/8 as follows:

- they are *module-name*s composed of the same dotted sequence of *identifier*s.

6.1 Declarations and definitions [basic.def]

Modify paragraph 6.1/1 as follows:

1 A declaration (Clause 10) may introduce one or more names into a translation unit or redeclare names introduced by previous declarations. If so, the declaration specifies the interpretation and ~~attributes~~semantic properties of these names. [...]

Append the following two bullets to paragraph 6.1/2:

2 A declaration is a *definition* unless

— ...
— it is an explicit specialization (17.7.3) whose *declaration* is not definition~~.~~,
— it is a *module-import-declaration*,
— it is a *proclaimed-ownership-declaration*.

[*Example:*

```
import std.io;                  // make names from std.io available
export module M;                // toplevel declaration for M
export struct Point {           // define and export Point
  int x;
  int y;
};
```

— *end example*]

6.2 One-definition rule [basic.def.odr]

Replace paragraph 6.2/1 with:

1 A variable, function, class type, enumeration type, or template shall not be defined where a prior definition is reachable (6.4).

Modify opening of paragraph 6.2/6 as follows

