TECHNICAL SPECIFICATION

ISO/IEC TS 19841

First edition
2015-10-01

Technical Specification for C++ Extensions for Transactional Memory

Spécification technique pour les extensions C++ de la mémoire transactionnelle

Reference number
ISO/IEC TS 19841:2015(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

 N4514

Contents

 N4514

Technical Specification for C++ Extensions for Transactional Memory

1 General [intro]

1.1 Scope [general.scope]

1 This Technical Specification describes extensions to the C++ Programming Language (1.3) that enable the specification of Transactional Memory. These extensions include new syntactic forms and modifications to existing language and library.

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 **green** to represent added text and ~~strikethrough~~ to represent deleted text.

3 This Technical Specification is non-normative. Some of the functionality described by this Technical Specification may be considered for standardization in a future version of C++, but it is not currently part of any C++ standard. Some of the functionality in this Technical Specification may never be standardized, and other functionality may be standardized in a substantially changed form.

4 The goal of this Technical Specification is to build widespread existing practice for Transactional Memory. It gives advice on extensions to those vendors who wish to provide them.

1.2 Acknowledgements [general.ack]

1 This work is the result of collaboration of researchers in industry and academia, including the Transactional Memory Specification Drafting Group and the follow-on WG21 study group SG5. We wish to thank people who made valuable contributions within and outside these groups, including Hans Boehm, Justin Gottschlich, Victor Luchangco, Jens Maurer, Paul McKenney, Maged Michael, Mark Moir, Torvald Riegel, Michael Scott, Tatiana Shpeisman, Michael Spear, Michael Wong, and many others not named here who contributed to the discussion.

1.3 Normative references [general.references]

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.

— ISO/IEC 14882:2014, *Programming Languages - C++*

2 ISO/IEC 14882:2014 is hereinafter called the *C++ Standard*. Beginning with section 1.10 below, all clause and section numbers, titles, and symbolic references in [brackets] refer to the corresponding elements of the C++ Standard. Sections 1.1 through 1.5 of this Technical Specification are introductory material and are unrelated to the similarly-numbered sections of the *C++ Standard*.

1.4 Implementation compliance [intro.compliance]

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

1.5 Feature testing [intro.features]

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

Table 1 -- Feature Test Macro

Name	Value	Header
`__cpp_transactional_memory`	201505	*predeclared*

1.10 Multi-threaded executions and data races [intro.multithread]

1 Add a paragraph 9 to section 1.10 [intro.multithread] after paragraph 8:

The start and the end of each synchronized block or atomic block is a full-expression (1.9 [intro.execution]). A synchronized block (6.9 [stmt.sync]) or atomic block (6.10 [stmt.tx]) that is not dynamically nested within another synchronized block or atomic block is called an outer block. [Note: Due to syntactic constraints, blocks cannot overlap unless one is nested within the other.] There is a global total order of execution for all outer blocks. If, in that total order, T1 is ordered before T2,

- **no evaluation in T2 happens before any evaluation in T1 and**
- **if T1 and T2 perform conflicting expression evaluations, then the end of T1 synchronizes with the start of T2.**

2 Change in 1.10 [intro.multithread] paragraph 10:

> **Synchronized and atomic blocks as well as certain** ~~Certain~~ library calls *synchronize with* other **synchronized blocks, atomic blocks, and** library calls performed by another thread.

3 Change in 1.10 [intro.multithread] paragraph 21:

> The execution of a program contains a *data race* if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other. Any such data race results in undefined behavior. [Note: It can be shown that programs that correctly use mutexes**, synchronized and atomic blocks,** and `memory_order_seq_cst` operations to prevent all data races and use no other synchronization operations behave as if the operations executed by their constituent threads were simply interleaved, with each value computation of an object being taken from the last side effect on that object in that interleaving. This is normally referred to as "sequential consistency". However, this applies only to data-race-free programs, and data-race-free programs cannot observe most program transformations that do not change single-threaded program semantics. In fact, most single-threaded program transformations continue to be allowed, since any program that behaves differently as a result must perform an undefined operation. -- end note]

4 Add a new paragraph 22 after 1.10 [intro.multithread] paragraph 21:

> **[Note: Due to the constraints on transaction safety (8.4.4 [dcl.fct.def.tx]), the following holds for a data-race-free program: If the start of an atomic block T is sequenced before an evaluation A, A is sequenced before the end of T, and A inter-thread happens before some evaluation B, then the end of T inter-thread happens before B. If an evaluation C inter-thread happens before that evaluation A, then C inter-thread happens before the start of T. These properties in turn imply that in any simple interleaved (sequentially consistent) execution, the operations of each atomic block appear to be contiguous in the interleaving. -- end note]**

2 Lexical conventions [lex]

2.11 Identifiers [lex.name]

1 In section 2.11 [lex.name] paragraph 2, add `transaction_safe` and `transaction_safe_dynamic` to the table.

2.12 Keywords [lex.key]

1 In section 2.12 [lex.key] paragraph 1, add the keywords `synchronized`, `atomic_noexcept`, `atomic_cancel`, and `atomic_commit` to the table.

4 Standard conversions [conv]

4.3 Function-to-pointer conversion [conv.func]

1 Change in section 4.3 [conv.func] paragraph 1:

> An lvalue of function type T can be converted to a prvalue of type "pointer to ~~T."~~ **T". An lvalue of type "transaction-safe function" can be converted to a prvalue of type "pointer to function".** The result is a pointer to the function. [Footnote: ...]

4.14 Transaction-safety conversion [conv.tx]

1 Add a new section 4.14 [conv.tx] paragraph 1:

> **4.14 [conv.tx] Transaction-safety conversion**
> **A prvalue of type "pointer to `transaction_safe` function" can be converted to a prvalue of type "pointer to function". The result is a pointer to the function. A prvalue of type "pointer to member of type `transaction_safe` function" can be converted to a prvalue of type "pointer to member of type function". The result points to the member function.**

5 Expressions [expr]

1 Change in 5 [expr] paragraph 13:

[Note: ...] The *composite pointer type* of two operands p1 and p2 having types T1 and T2, respectively, where at least one is a pointer or pointer to member type or `std::nullptr_t`, is:

— ...
— if T1 or T2 is "pointer to cv1 void" and the other type is "pointer to cv2 T", "pointer to cv12 void", where cv12 is the union of cv1 and cv2 ;
— **if T1 is "pointer to `transaction_safe` function" and T2 is "pointer to function", where the function types are otherwise the same, T2, and vice versa;**
— ...

5.1 Primary expressions [expr.prim]

5.1.2 Lambda expressions [expr.prim.lambda]

1 Change in 5.1.2 [expr.prim.lambda] paragraph 1:

```
lambda-declarator:
    ( parameter-declaration-clause ) mutable_opt transaction_safe_opt
        exception-specification_opt attribute-specifier-seq_opt trailing-return-type_opt
```


2 Change in 5.1.2 [expr.prim.lambda] paragraph 5:

This function call operator or operator template is declared `const` (9.3.1) if and only if the *lambda-expression*'s *parameter-declaration-clause* is not followed by `mutable`. It is neither virtual nor declared volatile. **It is declared `transaction_safe` if and only if the *lambda-expression*'s *parameter-declaration-clause* is followed by `transaction_safe` or, in a non-generic *lambda-expression*, it has a transaction-safe function definition (8.4.4 [dcl.fct.def.tx]).** Any *exception-specification* specified on a *lambda-expression* applies to the corresponding function call operator or operator template. ...

3 Change in 5.1.2 [expr.prim.lambda] paragraph 6:

The closure type for a non-generic *lambda-expression* with no *lambda-capture* has a public non-virtual non-explicit const **`transaction_safe`** conversion function to pointer to function with C++ language linkage (7.5 [dcl.link]) having the same parameter and return types as the closure type's function call operator. **That pointer is a pointer to transaction-safe function if the function call operator is transaction-safe.**

5.2 Postfix expressions [expr.post]

5.2.2 Function call [expr.call]

1 Add at the end of 5.2.2 [expr.call] paragraph 1:

... [Note: ...] **A call to a virtual function that is evaluated within an atomic block (6.10 [stmt.tx]) results in undefined behavior if the virtual function is declared `transaction_safe_dynamic` and the final overrider is not declared `transaction_safe`.**

2 Add paragraph 10 after 5.2.2 [expr.call] paragraph 9:

Recursive calls are permitted, except to the function named main (3.6.1)

Calling a function that is not transaction-safe (8.4.4 [dcl.fct.def.tx]) through a pointer to or lvalue of type "transaction-safe function" has undefined behavior.

5.2.9 Static cast [expr.static.cast]

1 Change in 5.2.9 [expr.static.cast] paragraph 7:

> The inverse of any standard conversion sequence (Clause 4 [conv]) not containing an lvalue-to-rvalue (4.1 [conv.lval]), array-to-pointer (4.2 [conv.array]), function-to-pointer (4.3), null pointer (4.10), null member pointer (4.11), ~~or~~ boolean (4.12)**, or transaction-safety (4.14 [conv.tx])** conversion, can be performed explicitly using `static_cast`. ...

5.10 Equality operators [expr.eq]

1 Change in 5.10 [expr.eq] paragraph 2:

> If at least one of the operands is a pointer, pointer conversions (4.10 [conv.ptr])**, transaction-safety conversions (4.14 [conv.tx]),** and qualification conversions (4.4 [conv.qual]) are performed on both operands to bring them to their composite pointer type (clause 5 [expr]). Comparing pointers is defined as follows: **Before transaction-safety conversions, if one pointer is of type "pointer to function", the other is of type "pointer to `transaction_safe` function", and both point to the same function, it is unspecified whether the pointers compare equal. Otherwise,** ~~Two~~ **two** pointers compare equal if they are both null, both point to the same function, or both represent the same address (3.9.2), otherwise they compare unequal.

5.16 Conditional operator [expr.cond]

1 Change in 5.16 [expr.cond] paragraph 6:

> — One or both of the second and third operands have pointer type; pointer conversions (4.10 [conv.ptr])**, transaction-safety conversions (4.14 [conv.tx]),** and qualification conversions (4.4 [conv.qual]) are performed to bring them to their composite pointer type (5 [expr]). ...
> — ...