
Technical Specification for C++ Extensions for Transactional Memory

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

iTeh STANDARD PREVIEW
(standards.iteh.ai)

[ISO/IEC TS 19841:2015](https://standards.iteh.ai/catalog/standards/sist/7a387f44-0153-48e2-978b-5609ff629ea6/iso-iec-ts-19841-2015)

<https://standards.iteh.ai/catalog/standards/sist/7a387f44-0153-48e2-978b-5609ff629ea6/iso-iec-ts-19841-2015>

iTeh STANDARD PREVIEW
(standards.iteh.ai)

ISO/IEC TS 19841:2015
<https://standards.iteh.ai/catalog/standards/sist/7a387f44-0153-48e2-978b-5609ff629ea6/iso-iec-ts-19841-2015>



COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2015, Published in Switzerland

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized otherwise in any form or by any means, electronic or mechanical, including photocopying, or posting on the internet or an intranet, without prior written permission. Permission can be requested from either ISO at the address below or ISO's member body in the country of the requester.

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

1	General	6
1.1	Scope	6
1.2	Acknowledgements	6
1.3	Normative references	6
1.4	Implementation compliance	6
1.5	Feature testing	6
1.10	Multi-threaded executions and data races	7
2	Lexical conventions	9
2.11	Identifiers	9
2.12	Keywords	9
4	Standard conversions	10
4.3	Function-to-pointer conversion	10
4.14	Transaction-safety conversion	10
5	Expressions	11
5.1	Primary expressions	11
5.1.2	Lambda expressions	11
5.2	Postfix expressions	11
5.2.2	Function call	11
5.2.9	Static cast	12
5.10	Equality operators	12
5.16	Conditional operator	12
6	Statements	13
6.6	Jump statements	13
6.9	Synchronized statement	13
6.10	Atomic statement	14
7	Declarations	15
7.4	The asm declaration	15
7.6	Attributes	15
7.6.6	Attribute for optimization in synchronized blocks	15
8	Declarators	16
8.3	Meaning of declarators	16
8.3.5	Functions	16
8.4	Function definitions	17
8.4.1	In general	17
8.4.4	Transaction-safe function	17
10	Derived classes	19
10.3	Virtual functions	19
13	Overloading	20
13.1	Overloadable declarations	20
13.3	Overload resolution	20
13.3.3	Best viable function	20
13.3.3.1	Implicit conversion sequences	20
13.3.3.1.1	Standard conversion sequences	20
13.4	Address of overloaded function	20
14	Templates	21
14.1	Template parameters	21
14.7	Template instantiation and specialization	21
14.7.3	Explicit specialization	21
14.8	Function template specializations	21
14.8.2	Template argument deduction	21
14.8.2.1	Deducing template arguments from a function call	21
15	Exception handling	22
15.1	Throwing an exception	22
15.2	Constructors and destructors	22

15.3	Handling an exception	22
15.4	Exception specifications	23
17	Library introduction	24
17.5	Method of description (Informative)	24
17.5.1	Structure of each clause	24
17.5.1.4	Detailed specifications	24
17.6	Library-wide requirements	24
17.6.3	Requirements on types and expressions	24
17.6.3.5	Allocator requirements	24
17.6.5	Conforming implementations	24
17.6.5.16	Transaction safety	24
18	Language support library	25
18.5	Start and termination	25
18.6	Dynamic memory management	25
18.6.1	Storage allocation and deallocation	25
18.6.2	Storage allocation errors	25
18.6.2.1	Class bad_alloc	25
18.6.2.2	Class bad_array_new_length	25
18.7	Type identification	25
18.7.2	Class bad_cast	25
18.7.3	Class bad_typeid	26
18.8	Exception handling	26
18.8.1	Class exception	26
18.8.2	Class bad_exception	26
18.10	Other runtime support	26
19	Diagnostics library	27
19.2	Exception classes	27
19.2.10	Class template tx_exception	27
20	General utilities library	28
20.2	Utility components	28
20.2.4	forward/move helpers	28
20.7	Memory	28
20.7.3	Pointer traits	28
20.7.3.2	Pointer traits member functions	28
20.7.5	Align	28
20.7.8	Allocator traits	29
20.7.8.2	Allocator traits static member functions	29
20.7.9	The default allocator	29
20.7.9.1	allocator members	29
20.7.11	Temporary buffers	29
20.7.12	Specialized algorithms	29
20.7.12.1	addressof	29
20.7.13	C library	29
20.8	Smart pointers	30
20.8.1	Class template unique_ptr	30
21	Strings library	31
21.1	General	31
21.4	Class template basic_string	31
21.4.3	basic_string iterator support	31
21.4.4	basic_string capacity	31
21.4.5	basic_string element access	31
23	Containers library	32
23.2	Container requirements	32
23.2.1	General container requirements	32
23.2.3	Sequence containers	32
23.2.5	Unordered associative containers	32
23.3	Sequence containers	33
23.3.2	Class template array	33

23.3.2.1	Class template array overview	33
23.3.3	Class template deque	33
23.3.3.1	Class template deque overview	33
23.3.4	Class template forward_list	33
23.3.4.1	Class template forward_list overview	33
23.3.4.6	forward_list operations	33
23.3.5	Class template list	33
23.3.5.1	Class template list overview	33
23.3.5.5	list operations	33
23.3.6	Class template vector	33
23.3.6.1	Class template vector overview	33
23.3.6.3	vector capacity	34
23.3.6.4	vector data	34
23.3.7	Class vector<bool>	34
23.4	Associative containers	34
23.4.4	Class template map	34
23.4.4.1	Class template map overview	34
23.4.5	Class template multimap	34
23.4.5.1	Class template multimap overview	34
23.4.6	Class template set	34
23.4.6.1	Class template set overview	34
23.4.7	Class template multiset	34
23.4.7.1	Class template multiset overview	34
23.5	Unordered associative containers	35
23.5.4	Class template unordered_map	35
23.5.4.1	Class template unordered_map overview	35
23.5.5	Class template unordered_multimap overview	35
23.5.5.1	Class template unordered_multimap overview	35
23.5.6	Class template unordered_set	35
23.5.6.1	Class template unordered_set overview	35
23.5.7	Class template unordered_multiset	35
23.5.7.1	Class template unordered_multiset overview	35
23.6	Container adaptors	35
23.6.1	In general	35
24	Iterators library	36
24.4	Iterator primitives	36
24.4.4	Iterator operations	36
24.5	Iterator adaptors	36
24.5.1	Reverse iterators	36
24.5.2	Insert iterators	36
24.5.3	Move iterators	36
24.7	range access	36
25	Algorithms library	37
25.1	General	37
26	Numerics library	38
26.7	Generalized numeric operations	38
26.7.1	Header <numeric> synopsis	38
26.8	C library	38

iTeh STANDARD PREVIEW
(standards.iteh.ai)

[ISO/IEC TS 19841:2015](https://standards.iteh.ai/catalog/standards/sist/7a387f44-0153-48e2-978b-5609ff629ea6/iso-iec-ts-19841-2015)

<https://standards.iteh.ai/catalog/standards/sist/7a387f44-0153-48e2-978b-5609ff629ea6/iso-iec-ts-19841-2015>

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
<code>__cpp_transactional_memory</code>	201505	<i>predeclared</i>

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.**

iTeh STANDARD PREVIEW (standards.iteh.ai)

[ISO/IEC TS 19841:2015](#)

<https://standards.iteh.ai/catalog/standards/sist/7a387f44-0153-48e2-978b-5609ff629ea6/iso-iec-ts-19841-2015>

- 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]

iTeh STANDARD PREVIEW
(standards.iteh.ai)

ISO/IEC TS 19841:2015

<https://standards.iteh.ai/catalog/standards/sist/7a387f44-0153-48e2-978b-5609ff629ea6/iso-iec-ts-19841-2015>

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.

iTeh STANDARD PREVIEW (standards.iteh.ai)

[ISO/IEC TS 19841:2015](https://standards.iteh.ai/catalog/standards/sist/7a387f44-0153-48e2-978b-5609ff629ea6/iso-iec-ts-19841-2015)

<https://standards.iteh.ai/catalog/standards/sist/7a387f44-0153-48e2-978b-5609ff629ea6/iso-iec-ts-19841-2015>

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.

iTeh STANDARD PREVIEW
(standards.iteh.ai)

ISO/IEC TS 19841:2015

<https://standards.iteh.ai/catalog/standards/sist/7a387f44-0153-48e2-978b-5609ff629ea6/iso-iec-ts-19841-2015>

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 ) mutableopt transaction_safeopt
  exception-specificationopt attribute-specifier-seqopt trailing-return-typeopt
```

- 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]). ...
- ...

iTeH STANDARD PREVIEW
<https://standards.iteh.ai/catalog/standards/sist/7a387f44-0153-48e2-978b-5609ff629ea6/iso-iec-ts-19841-2015>