Technical Specification for C++ Extensions for Transactional Memory

Warning
This document is not an ISO International Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an International Standard.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.
Copyright notice
This ISO document is a working draft or committee draft and is copyright-protected by ISO. While the reproduction of working drafts or committee drafts in any form for use by participants in the ISO standards development process is permitted without prior permission from ISO, neither this document nor any extract from it may be reproduced, stored or transmitted in any form for any other purpose without prior written permission from ISO.

Requests for permission to reproduce this document for the purpose of selling it should be addressed as shown below or to ISO’s member body in the country of the requester:

ISO copyright officer
Case postale 56, CH-1211 Geneva 20
Tel. +41 22 749 01 11
Fax +41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Reproduction may be subject to royalty payments or a licensing agreement.
Violators may be prosecuted.
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 Declerators ............................................................ 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
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>15.3</td>
<td>Handling an exception</td>
<td>22</td>
</tr>
<tr>
<td>15.4</td>
<td>Exception specifications</td>
<td>23</td>
</tr>
<tr>
<td>17</td>
<td>Library introduction</td>
<td>24</td>
</tr>
<tr>
<td>17.5</td>
<td>Method of description (Informative)</td>
<td>24</td>
</tr>
<tr>
<td>17.5.1</td>
<td>Structure of each clause</td>
<td>24</td>
</tr>
<tr>
<td>17.5.4</td>
<td>Detailed specifications</td>
<td>24</td>
</tr>
<tr>
<td>17.6</td>
<td>Library-wide requirements</td>
<td>24</td>
</tr>
<tr>
<td>17.6.3</td>
<td>Requirements on types and expressions</td>
<td>24</td>
</tr>
<tr>
<td>17.6.5</td>
<td>Conforming implementations</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>17.6.16 Transaction safety</td>
<td>24</td>
</tr>
<tr>
<td>18</td>
<td>Language support library</td>
<td>25</td>
</tr>
<tr>
<td>18.5</td>
<td>Start and termination</td>
<td>25</td>
</tr>
<tr>
<td>18.6</td>
<td>Dynamic memory management</td>
<td>25</td>
</tr>
<tr>
<td>18.6.1</td>
<td>Storage allocation and deallocation</td>
<td>25</td>
</tr>
<tr>
<td>18.6.2</td>
<td>Storage allocation errors</td>
<td>25</td>
</tr>
<tr>
<td>18.6.2.1</td>
<td>Class bad_alloc</td>
<td>25</td>
</tr>
<tr>
<td>18.6.2.2</td>
<td>Class bad_array_new_length</td>
<td>25</td>
</tr>
<tr>
<td>18.7</td>
<td>Type identification</td>
<td>25</td>
</tr>
<tr>
<td>18.7.2</td>
<td>Class bad_cast</td>
<td>25</td>
</tr>
<tr>
<td>18.7.3</td>
<td>Class bad_typeid</td>
<td>26</td>
</tr>
<tr>
<td>18.8</td>
<td>Exception handling</td>
<td>26</td>
</tr>
<tr>
<td>18.8.1</td>
<td>Class exception</td>
<td>26</td>
</tr>
<tr>
<td>18.8.2</td>
<td>Class bad_exception</td>
<td>26</td>
</tr>
<tr>
<td>18.10</td>
<td>Other runtime support</td>
<td>26</td>
</tr>
<tr>
<td>19</td>
<td>Diagnostics library</td>
<td>27</td>
</tr>
<tr>
<td>19.2</td>
<td>Exception classes</td>
<td>27</td>
</tr>
<tr>
<td>19.2.10</td>
<td>Class template tx_exception</td>
<td>27</td>
</tr>
<tr>
<td>20</td>
<td>General utilities library</td>
<td>28</td>
</tr>
<tr>
<td>20.2</td>
<td>Utility components</td>
<td>28</td>
</tr>
<tr>
<td>20.2.4</td>
<td>Forward/move helpers</td>
<td>28</td>
</tr>
<tr>
<td>20.7</td>
<td>Memory</td>
<td>28</td>
</tr>
<tr>
<td>20.7.3</td>
<td>Pointer traits</td>
<td>28</td>
</tr>
<tr>
<td>20.7.3.2</td>
<td>Pointer traits member functions</td>
<td>28</td>
</tr>
<tr>
<td>20.7.5</td>
<td>Align</td>
<td>28</td>
</tr>
<tr>
<td>20.7.8</td>
<td>Allocator traits</td>
<td>29</td>
</tr>
<tr>
<td>20.7.8.2</td>
<td>Allocator traits static member functions</td>
<td>29</td>
</tr>
<tr>
<td>20.7.9</td>
<td>The default allocator</td>
<td>29</td>
</tr>
<tr>
<td>20.7.9.1</td>
<td>allocators members</td>
<td>29</td>
</tr>
<tr>
<td>20.7.11</td>
<td>Temporary buffers</td>
<td>29</td>
</tr>
<tr>
<td>20.7.12</td>
<td>Specialized algorithms</td>
<td>29</td>
</tr>
<tr>
<td>20.7.12.1</td>
<td>addressof</td>
<td>29</td>
</tr>
<tr>
<td>20.7.13</td>
<td>C library</td>
<td>29</td>
</tr>
<tr>
<td>20.8</td>
<td>Smart pointers</td>
<td>30</td>
</tr>
<tr>
<td>20.8.1</td>
<td>Class template unique_ptr</td>
<td>30</td>
</tr>
<tr>
<td>21</td>
<td>Strings library</td>
<td>31</td>
</tr>
<tr>
<td>21.1</td>
<td>General</td>
<td>31</td>
</tr>
<tr>
<td>21.4</td>
<td>Class template basic_string</td>
<td>31</td>
</tr>
<tr>
<td>21.4.3</td>
<td>basic_string iterator support</td>
<td>31</td>
</tr>
<tr>
<td>21.4.4</td>
<td>basic_string capacity</td>
<td>31</td>
</tr>
<tr>
<td>21.4.5</td>
<td>basic_string element access</td>
<td>31</td>
</tr>
<tr>
<td>23</td>
<td>Containers library</td>
<td>32</td>
</tr>
<tr>
<td>23.2</td>
<td>Container requirements</td>
<td>32</td>
</tr>
<tr>
<td>23.2.1</td>
<td>General container requirements</td>
<td>32</td>
</tr>
<tr>
<td>23.2.3</td>
<td>Sequence containers</td>
<td>32</td>
</tr>
<tr>
<td>23.2.5</td>
<td>Unordered associative containers</td>
<td>32</td>
</tr>
<tr>
<td>23.3</td>
<td>Sequence containers</td>
<td>33</td>
</tr>
<tr>
<td>23.3.2</td>
<td>Class template array</td>
<td>33</td>
</tr>
</tbody>
</table>
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 ............................................ 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
1 General

1.1 Scope

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.

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.

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.

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

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 Shepisman, Michael Spear, Michael Wong, and many others not named here who contributed to the discussion.

1.3 Normative references

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++

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

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

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>__cpp_transactional_memory</td>
<td>201505</td>
<td>predeclared</td>
</tr>
</tbody>
</table>
1.10 Multi-threaded executions and data races

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.
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.

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 ]

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

2.11 Identifiers

In section 2.11 paragraph 2, add `transaction_safe` and `transaction_safe_dynamic` to the table.

2.12 Keywords

In section 2.12 paragraph 1, add the keywords `synchronized, atomic_noexcept, atomic_cancel, and atomic_commit` to the table.
4 Standard conversions

4.3 Function-to-pointer conversion

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", 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

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

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

5.1.2 Lambda expressions

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 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

5.2.2 Function call

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.

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

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

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 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

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]). ...

— ...
In 6 [stmt.stmt] paragraph 1, add two productions to the grammar:

\[
\text{statement:} \\
\text{labeled-statement} \\
\text{attribute-specifier-seqopt expression-statement} \\
\text{attribute-specifier-seqopt compound-statement} \\
\text{attribute-specifier-seqopt selection-statement} \\
\text{attribute-specifier-seqopt iteration-statement} \\
\text{attribute-specifier-seqopt jump-statement} \\
\text{declaration-statement} \\
\text{attribute-specifier-seqopt try-block} \\
\text{synchronized-statement} \\
\text{atomic-statement}
\]

Add a new paragraph 3 at the end of 6.6 [stmt.jump]:

Transfer out of an atomic block other than via an exception executes the end of the atomic block. [Note: Colloquially, this is known as committing the transaction. For exceptions, see 15.2 [except.ctor]. -- end note] Transfer out of a synchronized block (including via an exception) executes the end of the synchronized block.

Add a new section 6.9 [stmt.sync] paragraph 1:

6.9 [stmt.sync] Synchronized statement

\[
\text{synchronized-statement:} \\
\text{synchronized compound-statement}
\]

A synchronized statement is also called a synchronized block.

The start of the synchronized block is immediately before the opening } of the compound-statement. The end of the synchronized block is immediately after the closing } of the compound-statement.

A goto or switch statement shall not be used to transfer control into a synchronized block.

[ Example:

\[
\text{int f()}
\{
\text{static int i = 0;}
\text{synchronized {}
\text{printf("before \%d\n", i);}
++i;
\text{printf("after \%d\n", i);}
\text{return i;}
\text{}}
\}
\]

Each invocation of f (even when called from several threads simultaneously) retrieves a unique value (ignoring overflow). The output is guaranteed to comprise consistent before/after pairs. -- end example ]
6.10 Atomic statement

1 Add a new section 6.10 [stmt.tx] paragraph 1:

6.10 [stmt.tx] Atomic statement

\[
\text{atomic-statement:} \\
\text{atomic_noexcept compound-statement} \\
\text{atomic_cancel compound-statement} \\
\text{atomic_commit compound-statement}
\]

An atomic statement is also called an atomic block. The program is ill-formed if the compound-statement is a transaction-unsafe statement (8.4.4 [dcl.fct.def.tx]).

The start of the atomic block is immediately before the opening \{ of the compound-statement. The end of the atomic block is immediately after the closing \} of the compound-statement. [ Note: Thus, variables with automatic storage duration declared in the compound-statement are destroyed prior to reaching the end of the atomic block; see 6.6 [stmt.jump]. -- end note ]

A goto or switch statement shall not be used to transfer control into an atomic block.

[ Example:

\[
\text{int f()} \\
\{ \\
\text{static int i = 0;} \\
\text{atomic_noexcept \{} \\
\text{++i;} \\
\text{return i;} \\
\text{\}}
\]

Each invocation of f (even when called from several threads simultaneously) retrieves a unique value (ignoring overflow). -- end example ]
7 Declarations

7.4 The asm declaration

Change in 7.4 [dcl.asm] paragraph 1:

... The asm declaration is conditionally-supported; its meaning is implementation-defined. [ Note: Typically it is used to pass information through the implementation to an assembler. -- end note ] It is implementation-defined which asm declarations are transaction-safe (8.4.4 [dcl.fct.def.tx]), if any.

7.6 Attributes

7.6.6 Attribute for optimization in synchronized blocks

Add a new section 7.6.6 [dcl.attr.sync] paragraph 1:

7.6.6 [dcl.attr.sync] Attribute for optimization in synchronized blocks

The attribute-token optimize_for_synchronized specifies that a function definition should be optimized for invocation from a synchronized-statement (6.9 [stmt.sync]). It shall appear at most once in each attribute-list and no attribute-argument-clause shall be present. The attribute may be applied to the declarator-id in a function declaration. The first declaration of a function shall specify the optimize_for_synchronized attribute if any declaration of that function specifies the optimize_for_synchronized attribute. If a function is declared with the optimize_for_synchronized attribute in one translation unit and the same function is declared without the optimize_for_synchronized attribute in another translation unit, the program is ill-formed; no diagnostic required.

[ Example:

```c
// translation unit 1
[[optimize_for_synchronized]] int f(int);

void g(int x) {
    synchronized {
        int ret = f(x*x);
    }
}

// translation unit 2
#include <iostream>

extern int verbose;

[[optimize_for_synchronized]] int f(int x)
{
    if (x >= 0)
        return x;
    if (verbose > 1)
        std::cerr << "failure: negative x" << std::endl;
    return -1;
}
```

If the attribute were not present for \( \ell \), which is not declared transaction_safe, a program might have to drop out of speculative execution in \( g \)'s synchronized block every time when calling \( \ell \), although that is only actually required for displaying the error message in the rare verbose error case. -- end example ]
8 Declarators

1 Change in clause 8 paragraph 4:

parameters-and-qualifiers:
  { parameter-declaration-clause } cv-qualifier-seqopt
  ref-qualifieropt tx-qualifieropt exception-specificationopt attribute-specifier-seqopt

  tx-qualifier:
  transaction_safe
  transaction_safe_dynamic

8.3 Meaning of declarators

8.3.5 Functions

1 Change in 8.3.5 [dcl.fct] paragraph 1:

In a declaration T D where D has the form

D1 { parameter-declaration-clause } cv-qualifier-seqopt
  ref-qualifieropt tx-qualifieropt exception-specificationopt attribute-specifier-seqopt

and the type of the contained declarator-id in the declaration T D1 is "derived-declarator-type-list T", the type of the declarator-id in D is "derived-declarator-type-list tx-qualifieropt returning T", where the optional transaction_safe is present if a tx-qualifier is present. The optional attribute-specifier-seq appertains to the function type.

2 Change in 8.3.5 [dcl.fct] paragraph 2:

In a declaration T D where D has the form

D1 { parameter-declaration-clause } cv-qualifier-seqopt
  ref-qualifieropt tx-qualifieropt exception-specificationopt attribute-specifier-seqopt trailing-return-type

and the type of the contained declarator-id in the declaration T D1 is "derived-declarator-type-list T", T shall be the single type-specifier auto. The type of the declarator-id in D is "derived-declarator-type-list tx-qualifieropt returning trailing-return-type", where the optional transaction_safe is present if a tx-qualifier is present. The optional attribute-specifier-seq appertains to the function type.

3 Change in 8.3.5 [dcl.fct] paragraph 5:

... After determining the type of each parameter, any parameter of type "array of T" or "transaction_safeopt function returning T" is adjusted to be "pointer to T" or "pointer to transaction_safeopt function returning T," respectively. ...

4 Change in 8.3.5 [dcl.fct] paragraph 6:

... The return type, the parameter-type-list, the ref-qualifier, and the cv-qualifier-seq, and the transaction_safe qualifier, but not the default arguments (8.3.6 [dcl.fct.default]) or the exception specification (15.4 [except.spec]), are part of the function type. ...

5 Add paragraph 16 at the end of section 8.3.5 [dcl.fct]:

The transaction_safe Dynamic qualifier may only appear in a function declarator that declares a virtual function in a class definition. A virtual function declared with the transaction_safe Dynamic qualifier is considered to
be declared transaction_safe. [ Note: A virtual function so declared can be overridden by a function that is not transaction-safe (see 10.3 class virtual), but calling such an override from a synchronized or atomic block causes undefined behavior (see 5.2.2 expr.call). -- end note ] All declarations of a function shall be declared transaction_safe if any declaration of that function is declared transaction_safe, except that the declaration of an explicit specialization (14.7.3 [temp.expl.spec]) may differ from the declaration that would be instantiated from the template; no diagnostic is required if conflicting declarations appear in different translation units.

8.4 Function definitions

8.4.1 In general

1 Change in section 8.4.1 [dcl.fct.def.general] paragraph 2:

The declarator in a function-definition shall have the form

```
D1 [ parameter-declaration-clause ] cv-qualifier-seq
    [ ref-qualifier ... exception-specification ... attribute-specifier-seq ... parameters-and-qualifiers] trailing-return-type_opt
```

8.4.4 Transaction-safe function

1 Add a new section after 8.4.4 [dcl.fct.def.tx] paragraph 1:

8.4.4 [dcl.fct.def.tx] Transaction-safe function definitions

An expression is transaction-unsafe if it contains any of the following as a potentially-evaluated subexpression (3.2 [basic.def.odr]):

- an lvalue-to-rvalue conversion (4.1 [conv.lval]) applied to a volatile glvalue [ Note: referring to a volatile object through a non-volatile glvalue has undefined behavior; see 7.1.6.1 [dcl.type.cv] -- end note ],
- an expression that modifies an object through a volatile glvalue,
- the creation of a temporary object of volatile-qualified type or with a subobject of volatile-qualified type,
- a function call (5.2.2 expr.call) whose postfix-expression is an id-expression that names a non-virtual function that is not transaction-safe,
- an implicit call of a non-virtual function that is not transaction-safe, or
- any other call of a function, where the function type is not "transaction_safe function".

A statement is a transaction-unsafe statement if it lexically directly contains one of the following (including evaluations of default argument expressions in function calls and evaluations of brace-or-equal-initializers for non-static data members in aggregate initialization (8.5.1 dcl.init.aggr), but ignoring the declaration of default argument expressions, local classes, and the compound-statement of a lambda-expression):

- a full-expression that is transaction-unsafe,
- an asm-definition (7.4 [dcl.asm]) that is not transaction-safe,
- a declaration of a variable of volatile-qualified type or with a subobject of volatile-qualified type, or
- a statement that is transaction-unsafe (recursively).

A function has a transaction-safe definition if none of the following applies:

- any parameter has volatile-qualified type or has a subobject of volatile-qualified type,
- its compound-statement (including the one in the function-try-block, if any) is a transaction-unsafe statement,
- for a constructor or destructor, the corresponding class has a volatile non-static data member, or
- for a constructor, a full-expression in a mem-initializer or an assignment-expression in a brace-or-equal-initializer that is not ignored (12.6.2 [class.base.init]) is transaction-unsafe.

[ Example:

```cpp
extern volatile int * p = 0;
struct S {
    virtual ~S();
};
```
A function declared **transaction_safe** shall have a transaction-safe definition.

A function is **transaction-safe** if it is declared **transaction_safe** (see 8.3.5 [dcl.fct]), or if it is a non-virtual function defined before its first odr-use (3.2 [basic.def.odr]) and it has a transaction-safe function definition. A specialization of a function template or of a member function of a class template, where the function or function template is not declared **transaction_safe**, but defined before the first point of instantiation, is transaction-safe if and only if it satisfies the conditions for a transaction-safe function definition. [ Note: Even if a function is implicitly transaction-safe, its function type is not changed to "transaction_safe function". -- end note ]

While determining whether a function \( f \) is transaction-safe, \( f \) is assumed to be transaction-safe for directly and indirectly recursive calls. [ Example:

```c
int f(int x) { // is transaction-safe
    if (x <= 0)
        return 0;
    return x + f(x-1);
}
```

-- end example ]
10 Derived classes

10.3 Virtual functions

A function that overrides a function declared `transaction_safe`, but not `transaction_safe_dynamic`, is implicitly considered to be declared `transaction_safe`. [ Note: Its definition is ill-formed unless it actually has a transaction-safe definition (8.4.4 dcl.fct.def.tx). -- end note ] A function declared `transaction_safe_dynamic` that overrides a function declared `transaction_safe` (but not `transaction_safe_dynamic`) is ill-formed. [ Example:

```
struct B {
    virtual void f() transaction_safe;
    virtual ~B() transaction_safe_dynamic;
};

// pre-existing code
struct D1 : B {
    void f() override { } // ok
    ~D1() override { } // ok
};

struct D2 : B {
    void f() override { std::cout << "D2::f" << std::endl; }
    // error: transaction-safe f has transaction-unsafe definition
    ~D2() override { std::cout << "~D2" << std::endl; } // ok
};

struct D3 : B {
    void f() transaction_safe_dynamic override;
    // error: B::f() is transaction_safe
};

int main()
{
    D2 * d2 = new D2;
    B * b2 = d2;
    atomic_commit {
        B b;    // ok
        D1 d1; // ok
        B& b1 = d1;
        D2 x;     // error: destructor of D2 is not transaction-safe
        b1.f();   // ok, calls D1::f()
        delete b2; // undefined behavior: calls unsafe destructor of D2
    }
    //
    -- end example
```
13 Overloading

13.1 Overloadable declarations

1. Change in 13.1 [over.load] paragraph 2:

   Certain function declarations cannot be overloaded:
   — Function declarations that differ only in the return type cannot be overloaded.
   — Function declarations that differ only in the presence or absence of a \textit{tx-qualifier} cannot be overloaded.
   — ...

13.3 Overload resolution

13.3.3 Best viable function

13.3.3.1 Implicit conversion sequences

13.3.3.1.1 Standard conversion sequences

1. In 13.3.3.1.1 [over.ics.scs] paragraph 3, add an entry to table 12:

   
   | Conversion: Transaction-safety conversion |
   | Category: Lvalue transformation          |
   | Rank: Exact Match                        |
   | Subclause: 4.14 [conv.tx]                |

13.4 Address of overloaded function

1. Change in 13.4 [over.over] paragraph 1:

   ... The function selected is the one whose type is identical to the function type of the target type required in the context. A function with type \( F \) is selected for the function type \( FT \) of the target type required in the context if \( F \) (after possibly applying the transaction-safety conversion (4.14 [conv.tx])) is identical to \( FT \).  

   [Note: ...]

2. Change in 13.4 [over.over] paragraph 7:

   [Note: There are no standard conversions (Clause 4) of one pointer-to-function type into another. In particular, even Even  

   if \( B \) is a public base of \( D \), we have

   \[
   \begin{align*}
   & D^* f(); \\
   & B^* (*p1)() = &f; \quad // \text{error} \quad \\
   & \text{void } g(D^*); \quad \\
   & \text{void } (*p2)(B^*) = &g; \quad // \text{error} \quad \\
   \end{align*}
   \]
14 Templates

14.1 Template parameters

Change in 14.1 temp.param paragraph 8:

A non-type template-parameter of type "array of T" or "transaction_safe opt function returning T" is adjusted to be of type "pointer to T" or "pointer to transaction_safe opt function returning T", respectively. [ Example: ... ]

14.7 Template instantiation and specialization

14.7.3 Explicit specialization

Add a new paragraph 20 in 14.7.3 temp.expl.spec:

An explicit specialization of a function template or of a member function of a class template can be declared transaction_safe (8.3.5 [dcl.fct.def]) independently of whether the corresponding template entity is declared transaction_safe. [ Example:

```cpp
template<class T>
void f(T) transaction_safe;

template<>
void f(bool);   // not transaction-safe
```

-- end example ]

14.8 Function template specializations

Add a new paragraph 3 at the end of 14.8 [temp.fct.spec]:

A specialization instantiated from a function template or from a member function of a class template, where the function template or member function is declared transaction_safe, shall have a transaction-safe definition (8.4.4 [dcl.fct.def.tx]).

14.8.2 Template argument deduction

14.8.2.1 Deducing template arguments from a function call

Change in 14.8.2.1 temp.deduct.call paragraph 4:

... However, there are three cases that allow a difference:

— ... 
— The transformed A can be another pointer or pointer to member type that can be converted to the deduced A via a qualification conversion (4.4 c[conv.qual]) or a transaction-safety conversion (4.14 [conv.tx]).
— ...
15 Exception handling

15.1 Throwing an exception

1 Change in 15.1 except.throw paragraph 3:

... Evaluating a throw-expression with an operand throws an exception; the type of the exception object is determined by removing any top-level cv-qualifiers from the static type of the operand and adjusting the type from "array of T" or "transaction_safeopt function returning T" to "pointer to T" or "pointer to transaction_safeopt function returning T," respectively.

15.2 Constructors and destructors

1 Change the section heading of 15.2 [except.ctor] and paragraph 1:

Section 15.2 [except.ctor] Constructors, and destructors, and atomic blocks

As control passes from the point where an exception is thrown to a handler, destructors are invoked for all automatic objects constructed since the try block was entered yet still in scope (6.6 [stmt.jump], and atomic blocks are terminated (see below) where the start, but not the end of the block, was executed since the try block was entered (6.10 [stmt.tx]). The automatic objects are destroyed and atomic blocks are terminated in the reverse order of the completion of their construction and the execution of the start of the atomic blocks.

2 In section 15.2 [except.ctor], add new paragraphs 4 and 5:

An atomic block is terminated according to its kind, as follows: Terminating an atomic_commit block executes the end of the atomic block (1.10 intro.multithread) and has no further effect. [ Note: That is, control exits the atomic block after causing inter-thread synchronization. -- end note ] Terminating an atomic_cancel block, if the type of the current exception does not support transaction cancellation, or terminating an atomic_noexcept block, invokes std::abort (18.5 [support.start.term]). [ Footnote: If the effects of the atomic block become visible to other threads prior to program termination, some thread might make progress based on broken state, making debugging harder. -- end footnote ]. Terminating an atomic_cancel block, if the type of the current exception supports transaction cancellation, cancels the atomic block by performing the following steps, in order:

— A temporary object is copy-initialized (8.5 [dcl.init]) from the exception object. [ Note: If the initialization terminates via an exception, std::terminate is called (15.1 [except.throw]). -- end note ]
— The values of all memory locations in the program that were modified by side effects of the operations of the atomic block, except those occupied by the temporary object, are restored to the values they had at the time the start of the atomic block was executed.
— The end of the atomic block is executed. [ Note: This causes inter-thread synchronization. -- end note ]
— The temporary object is used as the exception object in the subsequent stack unwinding.

[ Note: A cancelled atomic block, although having no visible effect, still participates in data races (1.10 [intro.multithread]).-- end note ]

Non-volatile scalar types support transaction cancellation, as do those types specified as doing so in clauses 18 and 19.

15.3 Handling an exception

1 Change in 15.3 except.handle paragraph 3:

A handler is a match for an exception object of type E if

— ...
— the handler is of type cv T or const T& where T is a pointer type and E is a pointer type that can be converted to T by either or both of one or more of

§ 15.3
— a standard pointer conversion (4.10 [conv.ptr]) not involving conversions to pointers to private or protected or ambiguous classes
— a qualification conversion (4.4 [conv.qual])
— a transaction-safety conversion (4.14 [conv.tx])

15.4 Exception specifications

1 Change in 15.4 except.spec paragraph 2:

... A type cv T, "array of T", or "transaction_safeopt function returning T" denoted in an exception-specification is adjusted to type T, "pointer to T", or "pointer to transaction_safeopt function returning T", respectively.
17 Library introduction

17.5 Method of description (Informative)
17.5.1 Structure of each clause
17.5.1.4 Detailed specifications

Change in 17.5.1.4 [structure.specifications] paragraph 3:

— ...
— Synchronization: the synchronization operations (1.10) applicable to the function
— Transactions: the transaction-related properties of the function, in particular whether the function is transaction-safe (8.4.4 [dcl.fct.def.tx])
— ...

17.6 Library-wide requirements

17.6.3 Requirements on types and expressions
17.6.3.5 Allocator requirements

In table 27 in 17.6.3.5 [allocator.requirements] paragraph 2, add a note for X::rebind:

All operations that are transaction-safe on x shall be transaction-safe on y.

17.6.5 Conforming implementations
17.6.5.16 Transaction safety

Add a new section 17.6.5.16 [lib.txsafe] paragraph 1:

17.6.5.16 [lib.txsafe] Transaction safety

This standard explicitly requires that certain standard library functions are transaction-safe (8.4.4 dcl.fct.def.tx). An implementation shall not declare any standard library function signature as transaction_safe except for those where it is explicitly required.
18 Language support library

18.5 Start and termination

Change in 18.5 [support.start.term] paragraph 4:

```c
[[noreturn]] void abort(void) transaction_safe noexcept ;
```

The function `abort()` has additional behavior in this International Standard:
— The program is terminated without executing destructors for objects of automatic, thread, or static storage duration and without calling functions passed to `atexit()` (3.6.3).

18.6 Dynamic memory management

18.6.1 Storage allocation and deallocation

Add to 18.6.1 [new.delete] paragraph 1:

... The library versions of the global allocation and deallocation functions are declared `transaction_safe` (8.3.5 dcl.fct).

18.6.2 Storage allocation errors

Add a first paragraph to section 18.6.2 [alloc.errors]:

The classes `bad_alloc`, `bad_array_length`, and `bad_array_new_length` support transaction cancellation (15.2 [except.ctor]). [Note: Special support from the implementation might be necessary to successfully rethrow such an exception after leaving an atomic_cancel block. -- end note]

18.6.2.1 Class `bad_alloc`

In 18.6.2.1 [bad.alloc], add `transaction_safe` to the declaration of each non-virtual member function and add `transaction_safe_dynamic` to the declaration of each virtual member function.

18.6.2.2 Class `bad_array_new_length`

In 18.6.2.2 [new.badlength], add `transaction_safe` to the declaration of each non-virtual member function and add `transaction_safe_dynamic` to the declaration of each virtual member function.

18.7 Type identification

18.7.2 Class `bad_cast`

Change in 18.7.2 [bad.cast] paragraph 1:

The class `bad_cast` defines the type of objects thrown as exceptions by the implementation to report the execution of an invalid dynamic-cast expression (5.2.7 [expr.dynamic.cast]). The class supports transaction cancellation (15.2 [except.ctor]). [Note: Special support from the implementation might be necessary to successfully rethrow such an exception after leaving an atomic_cancel block. -- end note]
In 18.7.2 [bad.cast], add `transaction_safe` to the declaration of each non-virtual member function and add `transaction_safe_dynamic` to the declaration of each virtual member function.

18.7.3 Class `bad_typeid`  [bad.typeid]

Change in 18.7.3 [bad.typeid] paragraph 1:

The class `bad_typeid` defines the type of objects thrown as exceptions by the implementation to report a null pointer in a `typeid` expression (5.2.8 [expr.typeid]). [Note: Special support from the implementation might be necessary to successfully rethrow such an exception after leaving an `atomic_cancel` block. -- end note]

In 18.7.3 [bad.typeid], add `transaction_safe` to the declaration of each non-virtual member function and add `transaction_safe_dynamic` to the declaration of each virtual member function.

18.8 Exception handling  [support.exception]

18.8.1 Class `exception`  [exception]

In 18.8.1 [exception], add `transaction_safe` to the declaration of each non-virtual member function and add `transaction_safe_dynamic` to the declaration of each virtual member function.

18.8.2 Class `bad_exception`  [bad.exception]

Change in 18.8.2 [bad.exception] paragraph 1:

The class `bad_exception` defines the type of objects thrown as described in 15.5.2 [except.unexpected]. 15.5.2 [except.unexpected]. The class supports transaction cancellation (15.2 [except.ctor]). [Note: Special support from the implementation might be necessary to successfully rethrow such an exception after leaving an `atomic_cancel` block. -- end note]

In 18.8.2 [bad.exception], add `transaction_safe` to the declaration of each non-virtual member function and add `transaction_safe_dynamic` to the declaration of each virtual member function.

18.10 Other runtime support  [support.runtime]

Change in 18.10 [support.runtime] paragraph 4:

The function signature `longjmp(jmp_buf jbuf, int val)` has more restricted behavior in this International Standard. A `setjmp/longjmp` call pair has undefined behavior if replacing the `setjmp` and `longjmp` by `catch` and `throw` would invoke any non-trivial destructors for any automatic objects, or would transfer out of a synchronized block (6.9 [stmt.syne]) or atomic block (6.10 [stmt.tx]).
19 Diagnostics library

19.2 Exception classes

1 Change in 19.2 [std.exceptions] paragraph 3:

... These exceptions are related by inheritance. The exception classes support transaction cancellation (15.2 [except.ctor]). [ Note: Special support from the implementation might be necessary to successfully rethrow such an exception after leaving an atomic_cancel block. -- end note ].

Add the following to the synopsis in 19.2 [std.exceptions] paragraph 3:

```c++
template<class T> class tx_exception;
```

2 In 19.2 [std.exceptions], add transaction_safe to the declaration of each non-virtual member function and add transaction_safe_dynamic to the declaration of each virtual member function.

19.2.10 Class template tx_exception

1 Add a new section 19.2.10 [tx.exception] paragraph 1:

```c++
19.2.10 [tx.exception] Class template tx_exception
Class template tx_exception

namespace std {

    template<class T>
    class tx_exception : public runtime_error {
        public:
            explicit tx_exception(T value) transaction_safe;
            tx_exception(T value, const char* what_arg) transaction_safe;
            tx_exception(T value, const string& what_arg) transaction_safe;
            T get() const transaction_safe;
    };

}
```

A specialization of tx_exception supports transaction cancellation (15.2 [except.ctor]). If T is not a trivially copyable type (3.9 [basic.types]), the program is ill-formed.

```c++
    tx_exception(T value) transaction_safe;
```

**Effects:** Constructs an object of class tx_exception.

**Postcondition:** The result of calling get() is equivalent to value.

```c++
    tx_exception(T value, const char* what_arg) transaction_safe;
```

**Effects:** Constructs an object of class tx_exception.

**Postcondition:** strcmp(what(), what_arg) == 0 and the result of calling get() is equivalent to value.

```c++
    tx_exception(T value, const string& what_arg) transaction_safe;
```

**Effects:** Constructs an object of class tx_exception.

**Postcondition:** strcmp(what(), what_arg.c_str()) == 0 and the result of calling get() is equivalent to value.
20 General utilities library

20.2 Utility components

Add in 20.2 [utility] after the synopsis:

A function in this section is transaction-safe if all required operations are transaction-safe.

20.2.4 forward/move helpers

Change the signature in 20.2.4 [forward]:

```cpp
... template <class T> constexpr T&& forward(remove_reference_t<T>& t) transaction_safe noexcept
template <class T> constexpr T&& forward(remove_reference_t<T>&& t) transaction_safe noexcept
... template <class T> constexpr remove_reference_t<T>&& move(T&& t) transaction_safe noexcept;
... template <class T> constexpr conditional_t<
  !is_nothrow_move_constructible<T>::value && is_copy_constructible<T>::value,
  const T&, T&&> move_if_noexcept(T& x) transaction_safe noexcept;
```

20.7 Memory

20.7.3 Pointer traits

20.7.3.2 Pointer traits member functions

Change in 20.7.3.2 [pointer.traits.functions]:

```cpp
static pointer pointer_traits::pointer_to(see below r);
static pointer pointer_traits<T*>::pointer_to(see below r) transaction_safe noexcept;
... Transactions: The first member function is transaction-safe if the invoked member function of p is transaction-safe.
```

20.7.5 Align

Change the signature in 20.7.5 [ptr.align] paragraph 1:

```cpp
void* align(std::size_t alignment, std::size_t size,
  void*& ptr, std::size_t& space) transaction_safe;
```
20.7.8 Allocator traits

20.7.8.2 Allocator traits static member functions

1 In 20.7.8.2 [allocator.traits.members], add before paragraph 1:

A function in this section is transaction-safe if the invoked function (as specified below) is transaction-safe.

20.7.9 The default allocator

20.7.9.1 allocator members

1 In 20.7.9.1 [allocator.members], add "transaction_safe" to the declarations of the following member functions: address (twice), allocate, deallocate, max_size.

2 Change in 20.7.9.1 [allocator.members] paragraphs 12 and 13:

    template <class U, class... Args>
    void construct(U* p, Args&&... args);

**Effects:** ::new((void *)p) U(std::forward(args)...)  
**Transactions:** Transaction-safe if the invoked constructor of U is transaction-safe.

    template <class U>
    void destroy(U* p);

**Effects:** p->U()  
**Transactions:** Transaction-safe if the destructor of U is transaction-safe.

20.7.11 Temporary buffers

1 Change the signatures in 20.7.11 [temporary.buffer]:

    template <class T>
    pair<T*, ptrdiff_t> get_temporary_buffer(ptrdiff_t n) transaction_safe noexcept;

    ...

    template <class T> void return_temporary_buffer(T* p) transaction_safe;

20.7.12 Specialized algorithms

1 Change in 20.7.12 [specialized.algorithms] paragraph 1:

    ... In the following algorithms, if an exception is thrown there are no effects. Each of the following functions is transaction-safe if the constructor invoked via the placement allocation function is transaction-safe.

20.7.12.1 addressof

1 Change the signature in 20.7.12.1 [.specialized.addressof]:

    template <class T> T* addressof(T& r) transaction_safe noexcept;

20.7.13 C library

1 Add after 20.7.13 [c.malloc] paragraph 2:
The contents are the same as the Standard C library header <stdlib.h>, with the following changes:

The functions are transaction-safe.

2 Change in 20.7.13 [c.malloc] paragraph 7:

The contents are the same as the Standard C library header <string.h>, with the change to `memchr()` specified in 21.8 [c.strings]. The functions are transaction-safe.

20.8 Smart pointers

20.8.1 Class template `unique_ptr`

1 Change in 20.8.1 [unique.ptr] paragraph 5:

... The template parameter T of `unique_ptr` may be an incomplete type. Each of the functions in this section is transaction-safe if either no functions are called or all functions called are transaction-safe.
21 Strings library

21.1 General


All functions in this Clause are transaction-safe if the required operations on the supplied allocator (17.6.3.5 [allocator.requirements]) and character traits (21.2.1 [char.traits.require]) are transaction-safe.

21.4 Class template basic_string

21.4.3 basic_string iterator support

[1] In 21.4.3 [string.iterators], add "transaction_safe" to the declarations of all member functions.

21.4.4 basic_string capacity

[1] In 21.4.4 [string.capacity], add "transaction_safe" to the declarations of all member functions.

21.4.5 basic_string element access

[1] In 21.4.5 [string.access], add "transaction_safe" to the declarations of all member functions.
23 Containers library
[containers]

23.2 Container requirements
[container.requirements]

23.2.1 General container requirements
[container.requirements.general]

1 Add a new paragraph 4 in 23.2.1 [container.requirements.general] after paragraph 3:

Unless unconditionally specified to be transaction-safe, a function in this Clause is transaction-safe if all required operations are transaction-safe. [Note: This includes operations on the element type, on std::allocator_traits, and on Compare, Pred, or Hash objects, depending on the respective function. -- end note]

2 In table 96 in 23.2.1 [container.requirements.general] paragraph 4, add a note for X::iterator and X::const_iterator:

all functions required for the iterator category are transaction-safe

3 Add in 23.2.1 [container.requirements.general] after paragraph 6:

... If the container is empty, then begin() == end(). The member functions begin, end, cbegin, cend, size, max_size, and empty are transaction-safe.

4 Add in 23.2.1 [container.requirements.general] after paragraph 10:

If the iterator type of a container belongs to the bidirectional or random access iterator categories (24.2 [iterator.requirements]), the container is called reversible and satisfies the additional requirements in Table 97.

[ table ]
The member functions rbegin, rend, crbegin, and crend are transaction-safe.

23.2.3 Sequence containers
[sequence.reqmts]

1 Add in 23.2.3 [sequence.reqmts] before paragraph 17:

[ table ]
The member functions front, back, and at as well as the indexing operation a[n] are transaction-safe.

The member function at() provides bounds-checked access to container elements. at() throws out_of_range if n >= a.size().

23.2.5 Unordered associative containers
[unord.req]

1 Add in 23.2.5 [unord.req] after paragraph 12:

The behavior of a program that uses operator== or operator!= on unordered containers is undefined unless the Hash and Pred function objects respectively have the same behavior for both containers and the equality comparison operator for Key is a refinement [Footnote: ...] of the partition into equivalent-key groups produced by Pred.

The member functions bucket_count, max_bucket_count, bucket_size, begin, end, cbegin, cend, load_factor, and max_load_factor are transaction-safe.
23.3 Sequence containers

23.3.2 Class template array

23.3.2.1 Class template array overview

In 23.3.2.1 [array.overview] and the corresponding subsections, add "transaction_safe" to the declarations of all member functions except fill and swap.

23.3.3 Class template deque

23.3.3.1 Class template deque overview

In 23.3.3.1 [deque.overview], add "transaction_safe" to the declarations of all variants of the begin and end member functions and to the declarations of size, max_size, empty, operator[], front, back.

23.3.4 Class template forward_list

23.3.4.1 Class template forward_list overview

In 23.3.4.1 [forwardlist.overview], add "transaction_safe" to the declarations of all variants of the begin and end member functions and to the declarations of max_size, empty, front, splice_after, and reverse.

23.3.4.6 forward_list operations

In 23.3.4.6 [forwardlist.ops], add "transaction_safe" to the declarations of all variants of the begin and end member functions and to the declarations of max_size, empty, front, splice_after, and reverse.

23.3.5 Class template list

23.3.5.1 Class template list overview

In 23.3.5.1 [list.overview], add "transaction_safe" to the declarations of all variants of the begin and end member functions and to the declarations of size, max_size, empty, front, back, splice, and reverse.

23.3.5.5 list operations

In 23.3.5.5 [list.ops], add "transaction_safe" to the declarations of all variants of the begin and end member functions and to the declarations of size, max_size, empty, front, back, splice, and reverse.

23.3.6 Class template vector

23.3.6.1 Class template vector overview

In 23.3.6.1 [vector.overview], add "transaction_safe" to the declarations of all variants of the begin and end member functions and to the declarations of size, max_size, capacity, empty, operator[], front, back, data.
23.3.6.3 **vector capacity**

In 23.3.6.3 [vector.capacity], add "transaction_safe" to the declarations of all variants of the `begin` and `end` member functions and to the declarations of `size`, `max_size`, `capacity`, `empty`, `operator[]`, `front`, `back`, `data`.

23.3.6.4 **vector data**

In 23.3.6.4 [vector.data], add "transaction_safe" to the declarations of all variants of the `begin` and `end` member functions and to the declarations of `size`, `max_size`, `capacity`, `empty`, `operator[]`, `front`, `back`, `data`.

23.3.7 **Class vector<bool>**

In 23.3.7 [vector.bool], add "transaction_safe" to the declarations of all variants of the `begin` and `end` member functions, to the declarations of `size`, `max_size`, `capacity`, `empty`, `operator[]`, `front`, `back`, and `flip`, and to the static member function `swap`.

23.4 **Associative containers**

23.4.4 **Class template map**

23.4.4.1 **Class template map overview**

In 23.4.4.1 [map.overview], add "transaction_safe" to the declarations of all variants of the `begin` and `end` member functions and to the declarations of `size`, `max_size`, `empty`.

23.4.5 **Class template multimap**

23.4.5.1 **Class template multimap overview**

In 23.4.5.1 [multimap.overview], add "transaction_safe" to the declarations of all variants of the `begin` and `end` member functions and to the declarations of `size`, `max_size`, `empty`.

23.4.6 **Class template set**

23.4.6.1 **Class template set overview**

In 23.4.6.1 [set.overview], add "transaction_safe" to the declarations of all variants of the `begin` and `end` member functions and to the declarations of `size`, `max_size`, `empty`.

23.4.7 **Class template multiset**

23.4.7.1 **Class template multiset overview**

In 23.4.7.1 [multiset.overview], add "transaction_safe" to the declarations of all variants of the `begin` and `end` member functions and to the declarations of `size`, `max_size`, `empty`. 
23.5 Unordered associative containers

23.5.4 Class template unordered_map

23.5.4.1 Class template unordered_map overview

In 23.5.4.1 [unord.map.overview], add "transaction_safe" to the declarations of all variants of the begin and end member functions and to the declarations of size, max_size, empty, operator[], bucket_count, max_bucket_count, bucket_size, load_factor, and max_load_factor.

23.5.5 Class template unordered_multimap overview

23.5.5.1 Class template unordered_multimap overview

In 23.5.5.1 [unord.multimap.overview], add "transaction_safe" to the declarations of all variants of the begin and end member functions and to the declarations of size, max_size, empty, operator[], bucket_count, max_bucket_count, bucket_size, load_factor, and max_load_factor.

23.5.6 Class template unordered_set

23.5.6.1 Class template unordered_set overview

In 23.5.6.1 [unord.set.overview], add "transaction_safe" to the declarations of all variants of the begin and end member functions and to the declarations of size, max_size, empty, operator[], bucket_count, max_bucket_count, bucket_size, load_factor, and max_load_factor.

23.5.7 Class template unordered_multiset

23.5.7.1 Class template unordered_multiset overview

In 23.5.7.1 [unord.multiset.overview], add "transaction_safe" to the declarations of all variants of the begin and end member functions and to the declarations of size, max_size, empty, operator[], bucket_count, max_bucket_count, bucket_size, load_factor, and max_load_factor.

23.6 Container adaptors

23.6.1 In general

Add in 23.6.1 [container.adaptors.general] after paragraph 3:

For container adaptors, no swap function throws an exception unless that exception is thrown by the swap of the adaptor's Container or Compare object (if any).

A member function \(\mathcal{E}\) of a container adaptor is transaction-safe if the required member functions of the adaptor's Container and Compare (if any) are transaction-safe, as given by the specification for \(\mathcal{E}\).
24 Iterators library

24.4 Iterator primitives

24.4.4 Iterator operations

1 Change in 24.4.4 [iterator.operations] paragraph 1:

Since only random access iterators provide + and - operators, the library provides two function templates advance and distance. These function templates use + and - for random access iterators (and are, therefore, constant time for them); for input, forward and bidirectional iterators they use ++ to provide linear time implementations. A specialization of a function template specified in this Clause is transaction-safe if all operations required for the template arguments are transaction-safe.

24.5 Iterator adaptors

24.5.1 Reverse iterators

1 Change in 24.5.1 [reverse.iterators] paragraph 1:

Class template reverse_iterator is an iterator adaptor that iterates from the end of the sequence defined by its underlying iterator to the beginning of that sequence. The fundamental relation between a reverse iterator and its corresponding iterator i is established by the identity: *(reverse_iterator(i)) == *(i - 1). A member function specified in this Clause is transaction-safe if all operations required for the template argument of reverse_iterator are transaction-safe.

24.5.2 Insert iterators

1 Add a new paragraph after 24.5.2 [insert.iterators] paragraph 2:

A function or function template specified in this Clause is transaction-safe if all operations required for the template arguments are transaction-safe.

24.5.3 Move iterators

1 Add a new paragraph after 24.5.3 [move.iterators] paragraph 2:

A member function specified in this Clause is transaction-safe if all operations required for the template arguments are transaction-safe.

24.7 range access

1 Change in 24.7 [iterator.range] paragraph 1:

In addition to being available ..., and <vector>. A specialization of a function template specified in this Clause is transaction-safe if all required operations (as specified by the Returns element) are transaction-safe.

2 In 24.7 [iterator.range], add "transaction_safe" to the declarations of begin(T (&array)[N]) and end(T (&array)[N])
25 Algorithms library

25.1 General

1 Add a new 25.1 [algorithms.general] paragraph 13:

A specialization of a function template specified in this Clause is transaction-safe if all functions and operations required for the template arguments are transaction-safe. [ Example: The fill function (25.3.6 [alg.fill]) is transaction-safe if all required operations of its ForwardIterator template argument are transaction-safe and if T's copy assignment operator is transaction-safe. -- end example ]
A specialization of a function template specified in this Clause is transaction-safe if all functions and operations required for the template arguments are transaction-safe (see 25.1 [algorithms.general]).

The functions from `<stdlib.h>`, including the additional overloads in `<cstdlib>` (see below), but excluding `rand` and `srand`, are transaction-safe.