ISO/IEC JTC 1/SC 22/WG 21 N4514
Date: 2015-05-08
ISO/IEC JTC1 SC22
Secretariat: ANSI

Technical Specification for C++ Extensions for Transactional Memory

Document type: Draft Technical Specification
Document subtype:
Document stage: (40) Enquiry
Document language: E

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

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]). ...
>
> — ...

6 Statements [stmt.stmt]

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

statement:
  labeled-statement
  attribute-specifier-seq$_{opt}$ expression-statement
  attribute-specifier-seq$_{opt}$ compound-statement
  attribute-specifier-seq$_{opt}$ selection-statement
  attribute-specifier-seq$_{opt}$ iteration-statement
  attribute-specifier-seq$_{opt}$ jump-statement
  declaration-statement
  attribute-specifier-seq$_{opt}$ try-block
  synchronized-statement
  atomic-statement

6.6 Jump statements [stmt.jump]

1 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.

6.9 Synchronized statement [stmt.sync]

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

6.9 [stmt.sync] Synchronized statement

***synchronized-statement*:**
  `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:

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

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 [stmt.tx]

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

6.10 [stmt.tx] Atomic statement

```
atomic-statement:
        atomic_noexcept compound-statement
        atomic_cancel compound-statement
        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:

```
int f()
{
  static int i = 0;
  atomic_noexcept {
    ++i;
    return i;
  }
}
```

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

7 Declarations [dcl.dcl]

7.4 The *asm* declaration [dcl.asm]

1 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 [dcl.attr]

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

1 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:

```
// 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 `f`, 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 `f`, although that is only actually required for displaying the error message in the rare verbose error case. -- end example]

8 Declarators [dcl.decl]

1 Change in clause 8 paragraph 4:

> *parameters-and-qualifiers*:
> (*parameter-declaration-clause*) *cv-qualifier-seq*$_{opt}$
> *ref-qualifier*$_{opt}$ ***tx-qualifier***$_{\mathbf{opt}}$ *exception-specification*$_{opt}$ *attribute-specifier-seq*$_{opt}$
>
> ***tx-qualifier*****:**
> **`transaction_safe`**
> **`transaction_safe_dynamic`**

8.3 Meaning of declarators [dcl.meaning]

8.3.5 Functions [dcl.fct]

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-seq*$_{opt}$
> *ref-qualifier*$_{opt}$ ***tx-qualifier***$_{\mathbf{opt}}$ *exception-specification*$_{opt}$ *attribute-specifier-seq*$_{opt}$
>
> 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 **`transaction_safe`**$_{\mathbf{opt}}$ function of (parameter-declaration-clause) *cv-qualifier-seq*$_{opt}$ *ref-qualifier*$_{opt}$ 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-seq*$_{opt}$
> *ref-qualifier*$_{opt}$ ***tx-qualifier***$_{\mathbf{opt}}$ *exception-specification*$_{opt}$ *attribute-specifier-seq*$_{opt}$ *tr*
>
> 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 **`transaction_safe`**$_{\mathbf{opt}}$ function of (parameter-declaration-clause) *cv-qualifier-seq*$_{opt}$ *ref-qualifier*$_{opt}$ 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_safe`**$_{\mathbf{opt}}$ function returning T" is adjusted to be "pointer to T" or "pointer to **`transaction_safe`**$_{\mathbf{opt}}$ 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 overrider 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 [dcl.fct.def]

8.4.1 In general [dcl.fct.def.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*$_{opt}$~~
~~*ref-qualifier*$_{opt}$ *exception-specification*$_{opt}$ *attribute-specifier-seq*$_{opt}$~~
parameters-and-qualifiers *trailing-return-type*$_{opt}$

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

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-initializer*s 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:

```
extern volatile int * p = 0;
struct S {
  virtual ~S();
};
```

```
int f() transaction_safe {
  int x = 0;   // ok: not volatile
  p = &x;      // ok: the pointer is not volatile
  int i = *p;      // error: read through volatile glvalue
  S s;         // error: invocation of unsafe destructor
}
```

-- end example]

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:

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

-- end example]

10 Derived classes [class.derived]

10.3 Virtual functions [class.virtual]

1 Add a new paragraph 17 at the end of section 10.3 [class.virtual]:

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 [over]

13.1 Overloadable declarations [over.load]

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 *tx-qualifier* cannot be overloaded.**
- ...

13.3 Overload resolution [over.match]

13.3.3 Best viable function [over.match.best]

13.3.3.1 Implicit conversion sequences [over.best.ics]

13.3.3.1.1 Standard conversion sequences [over.ics.scs]

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 [over.over]

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

```
D* f();
B* (*p1)() = &f;     // error
void g(D*);
void (*p2)(B*) = &g; // error
```

]

14 Templates [temp]

14.1 Template parameters [temp.param]

1 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 [temp.spec]

14.7.3 Explicit specialization [temp.expl.spec]

1 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:**

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

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

> **-- end example]**

14.8 Function template specializations [temp.fct.spec]

1 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 [temp.deduct]

14.8.2.1 Deducing template arguments from a function call [temp.deduct.call]

4 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[onv.qual]) **or a transaction-safety conversion (4.14 [conv.tx])**.
>
> — ...

15 Exception handling [except]

15.1 Throwing an exception [except.throw]

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_safe`$_{opt}$** function returning T" to "pointer to T" or "pointer to **`transaction_safe`$_{opt}$** function returning T," respectively.

15.2 Constructors and destructors [except.ctor]

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 [except.handle]

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**

- — 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 [except.spec]

1 Change in 15.4 except.spec paragraph 2:

> ... A type cv T, "array of T", or "`transaction_safe`$_{opt}$ function returning T" denoted in an exception-specification is adjusted to type T, "pointer to T", or "pointer to `transaction_safe`$_{opt}$ function returning T", respectively.

17 Library introduction [library]

17.5 Method of description (Informative) [description]

17.5.1 Structure of each clause [structure]

17.5.1.4 Detailed specifications [structure.specifications]

1 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 [requirements]

17.6.3 Requirements on types and expressions [utility.requirements]

17.6.3.5 Allocator requirements [allocator.requirements]

1 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 [conforming]

17.6.5.16 Transaction safety [lib.txsafe]

1 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 [language.support]

18.5 Start and termination [support.start.term]

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

```
[[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 [support.dynamic]

18.6.1 Storage allocation and deallocation [new.delete]

1 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 [alloc.errors]

1 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` [bad.alloc]

1 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` [new.badlength]

1 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 [support.rtti]

18.7.2 Class `bad_cast` [bad.cast]

1 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]**

2 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]

1 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]). **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]**

2 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]

1 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]

1 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]**

2 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]

1 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.sync]) or atomic block (6.10 [stmt.tx])**.

19 Diagnostics library [diagnostics]

19.2 Exception classes [std.exceptions]

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:

```
    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` [tx.exception]

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

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.

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

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

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

```
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`.**

```
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 [utilities]

20.2 Utility components [utility]

1 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 [forward]

1 Change the signature in 20.2.4 [forward]:

```
...
  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 [memory]

20.7.3 Pointer traits [pointer.traits]

20.7.3.2 Pointer traits member functions [pointer.traits.functions]

1 Change in 20.7.3.2 [pointer.traits.functions]:

```
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 `Ptr` is transaction-safe.**

20.7.5 Align [ptr.align]

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

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

20.7.8 Allocator traits [allocator.traits]

20.7.8.2 Allocator traits static member functions [allocator.traits.members]

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 [default.allocator]

20.7.9.1 `allocator` members [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 [temporary.buffer]

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 [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` [specialized.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 [c.malloc]

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 [smartptr]

20.8.1 Class template `unique_ptr` [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 [strings]

21.1 General [strings.general]

1 Add after 21.1 [strings.general] paragraph 1:

> **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` [basic.string]

21.4.3 `basic_string` iterator support [string.iterators]

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

21.4.4 `basic_string` capacity [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 [string.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 [sequences]

23.3.2 Class template `array` [array]

23.3.2.1 Class template `array` overview [array.overview]

1 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` [deque]

23.3.3.1 Class template `deque` overview [deque.overview]

1 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` [forwardlist]

23.3.4.1 Class template `forward_list` overview [forwardlist.overview]

1 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 [forwardlist.ops]

1 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` [list]

23.3.5.1 Class template `list` overview [list.overview]

1 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 [list.ops]

1 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` [vector]

23.3.6.1 Class template `vector` overview [vector.overview]

1 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 [vector.capacity]

1 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 [vector.data]

1 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>` [vector.bool]

1 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 [associative]

23.4.4 Class template `map` [map]

23.4.4.1 Class template `map` overview [map.overview]

1 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`, and `empty`.

23.4.5 Class template `multimap` [multimap]

23.4.5.1 Class template `multimap` overview [multimap.overview]

1 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`, and `empty`.

23.4.6 Class template `set` [set]

23.4.6.1 Class template `set` overview [set.overview]

1 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`, and `empty`.

23.4.7 Class template `multiset` [multiset]

23.4.7.1 Class template `multiset` overview [multiset.overview]

1 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`, and `empty`.

23.5 Unordered associative containers [unord]

23.5.4 Class template `unordered_map` [unord.map]

23.5.4.1 Class template `unordered_map` overview [unord.map.overview]

1 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 [unord.multimap]

23.5.5.1 Class template `unordered_multimap` overview [unord.multimap.overview]

1 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` [unord.set]

23.5.6.1 Class template `unordered_set` overview [unord.set.overview]

1 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` [unord.multiset]

23.5.7.1 Class template `unordered_multiset` overview [unord.multiset.overview]

1 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 [container.adaptors]

23.6.1 In general [container.adaptors.general]

1 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 `f` 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 `f`.**

24 Iterators library [iterators]

24.4 Iterator primitives [iterator.primitives]

24.4.4 Iterator operations [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 [predef.iterators]

24.5.1 Reverse iterators [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 [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 [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 [iterator.range]

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 [algorithms]

25.1 General [algorithms.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]

26 Numerics library [numerics]

26.7 Generalized numeric operations [numeric.ops]

26.7.1 Header <numeric> synopsis [numeric.ops.overview]

1 Add a new paragraph after 26.7.1 [numeric.ops.overview] paragraph 1:

> **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]).**

26.8 C library [c.math]

1 Add after 26.8 [c.math] paragraph 4:

> The contents of these headers are the same as the Standard C library headers <math.h> and <stdlib.h> respectively, with the following changes:
> **The functions from <stdlib.h>, including the additional overloads in <cstdlib> (see below), but excluding rand and srand, are transaction-safe.**