

Document Number: 5036
Date: 2026-02-22
Reply to: Michael Wong
fraggamuffin@gmail.com

ISO/IEC JTC1/SC22/WG21 White Paper N5036, Extensions to C++ for Transactional Memory Version 2

Contents

1	Scope	1
2	Normative references	2
3	Terms and definitions	3
4	General	4
4.1	Implementation compliance	4
4.2	Acknowledgments	4
5	Lexical conventions	5
5.1	Identifiers	5
6	Basics	6
6.9	Program execution	6
8	Statements	8
8.1	Preamble	8
8.8	Atomic statement	8
15	Preprocessor	10
15.11	Predefined macro names	10
16	Library introduction	11
16.4	Library-wide requirements	11

1 Scope

[scope]

- ¹ This document describes requirements for implementations of an interface that computer programs written in the C++ programming language can use to express groups of operations (known as transactions) that appear to execute atomically in relation to other transactions (concurrent or otherwise).
- ² ISO/IEC 14882:2020 provides important context and specification for this document. This document is written as a set of changes against that specification. Instructions to modify or add paragraphs are written as explicit instructions. Modifications made directly to existing text from ISO/IEC 14882:2020 use underlining to represent added text and ~~striketrough~~ to represent deleted text.
- ³ This document is non-normative. Some of the functionality described by this document 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 document may never be standardized, and other functionality may be standardized in a substantially changed form.
- ⁴ The goal of this document is to build widespread existing practice for eventually adopting transactional memory in C++.

2 Normative references

[refs]

- 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.
- (1.1) — ISO/IEC 14882:2020, *Programming Languages — C++*
- ² ISO/IEC 14882:2020 is herein called the C++ Standard. References to clauses within the C++ Standard are written as “C++20 §3.2”.

3 Terms and definitions

[defs]

¹ No terms and definitions are listed in this document. ISO and IEC maintain terminological databases for use in standardization at the following addresses:

- (1.1) — IEC Electropedia: available at <https://www.electropedia.org/>
- (1.2) — ISO Online browsing platform: available at <https://www.iso.org/obp>

4 General

[general]

4.1 Implementation compliance

[general.compliance]

- ¹ Conformance requirements for this document are those defined in C++20 §4.1, as applied to a merged document consisting of C++20 amended by this document.

[*Note 1*: Conformance is defined in terms of the behavior of programs. — *end note*]

4.2 Acknowledgments

[general.ack]

This work is the result of a collaboration of researchers in industry and academia. We wish to thank the original authors of this document, Michael Wong, Jens Maurer, Hans Boehm, Michael Spear, Michael L. Scott, Victor Luchangco, Maged Michael, and Paul McKenney. We also wish to thank people who made valuable contributions within and outside these groups, including all SG 5 members and the original TM group, and many others not named here who contributed to the discussion.

5 Lexical conventions

[lex]

5.1 Identifiers

[lex.name]

In C++20 §5.10, add atomic to the table of identifiers with special meaning (Table 4).

6 Basics

[basic]

6.9 Program execution

[basic.exec]

6.9.1 Sequential execution

[intro.execution]

Change in C++20 §6.9.1 paragraph 5 as indicated:

- 5 A *full-expression* is
- (5.1) — ...
 - (5.2) — an invocation of a destructor generated at the end of the lifetime of an object other than a temporary object (6.7.7) whose lifetime has not been extended, ~~or~~
 - (5.3) — the start or the end of an atomic block (8.8), or
 - (5.4) — an expression that is not a subexpression of another expression and that is not otherwise part of a full-expression.

6.9.2 Multi-threaded executions and data races

[intro.multithread]

6.9.2.2 Data races

[intro.races]

Change in C++20 §6.9.2.2 paragraph 6 as indicated:

- 6 ~~Certain~~ Atomic blocks as well as certain library calls may *synchronize with* other atomic blocks and library calls performed by another thread.

Add a new paragraph after C++20 §6.9.2.2 paragraph 20:

- 21 The execution of an atomic block that is not dynamically nested within another atomic block is termed a *transaction*.
 [Note 21: Due to syntactic constraints, blocks cannot overlap unless one is nested within the other. — end note]
 There is a global total order of execution for all transactions. If, in that total order, a transaction T1 is ordered before a transaction T2, then
- (21.1) — no evaluation in T2 happens before any evaluation in T1 and
 - (21.2) — if T1 and T2 perform conflicting expression evaluations, then the end of T1 synchronizes with the start of T2.
- [Note 22: If the evaluations in T1 and T2 do not conflict, they might be executed concurrently. — end note]
- 22 Two actions are *potentially concurrent* if ...

Change in C++20 §6.9.2.2 paragraph 21:

- 21 ...
- [Note 21: It can be shown that programs that correctly use mutexes, 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". ... — end note]

Add a new paragraph after C++20 §6.9.2.1 paragraph 21:

- 22 [Note 22: 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*, *A* strongly happens before some evaluation *B*, and *B* is not sequenced before the end of *T*, then the end of *T* strongly happens before *B*. If an evaluation *C* strongly happens before that evaluation *A* and *C* is not sequenced after the start of *T*, then *C* strongly 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*]

6.9.2.3 Forward progress

[intro.progress]

Change in C++20 §6.9.2.3 paragraph 1 as indicated:

- 1 ~~The implementation may assume that any thread will eventually do~~ An *inter-thread side effect*
 is one of the following:
- (1.1) — ~~terminate,~~
 - (1.2) — a call to a library I/O function,
 - (1.3) — an access through a volatile glvalue, or
 - (1.4) — a synchronization operation or an atomic operation (Clause 31).

The implementation may assume that any thread will eventually terminate or evaluate an inter-thread side effect.

[*Note 1*: This is intended to allow compiler transformations such as removal of empty loops, even when termination cannot be proven. — *end note*]

8 Statements

[stmt.stmt]

8.1 Preamble

[stmt.pre]

- ¹ Add a production to the grammar in C++20 §8.1 as indicated:

```
statement:
    labeled-statement
    attribute-specifier-seqopt expression-statement
    attribute-specifier-seqopt compound-statement
    attribute-specifier-seqopt selection-statement
    attribute-specifier-seqopt iteration-statement
    attribute-specifier-seqopt jump-statement
    declaration-statement
    attribute-specifier-seqopt try-block
    atomic-statement
```

Add a new subclause before C++20 §8.8:

8.8 Atomic statement

[stmt.tx]

```
atomic-statement:
    atomic do compound-statement
```

- ¹ An *atomic-statement* is also called an *atomic block*.

- ² The start of the atomic block is immediately after the opening { of the *compound-statement*. Evaluation of the end of the atomic block occurs when the invocation of the destructor for a hypothetical automatic variable of class type declared at the start of the atomic block would happen (8.7.1).

[*Note 1*: Thus, variables with automatic storage duration declared in the *compound-statement* are destroyed prior to reaching the end of the atomic block; see 8.7. — *end note*]

- ³ A **case** or **default** label appearing within an atomic block shall be associated with a **switch** statement (8.5.3) within the same atomic block. A label (8.2) declared in an atomic block shall only be referred to by a statement in the same atomic block.

- ⁴ If the execution of an atomic block evaluates an inter-thread side effect (6.9.2.3) or if an atomic block is exited via an exception, the behavior is undefined.

- ⁵ *Recommended practice*: In case an atomic block is exited via an exception, the program should be terminated without invoking a terminate handler (17.9.5) or destroying any objects with static or thread storage duration (6.9.3.4).

- ⁶ If the execution of an atomic block evaluates any of the following outside of a manifestly constant-evaluated context (7.7), the behavior is implementation-defined:

- (6.1) — an *asm-declaration* (9.10);
- (6.2) — an invocation of a function, unless
 - (6.2.1) — the function is inline with a reachable definition or
 - (6.2.2) — the function is a library function that may be used in an atomic block 16.4.6.17;
- (6.3) — a virtual function call (7.6.1.3);
- (6.4) — a function call, unless overload resolution selects
 - (6.4.1) — a named function (12.4.2.2.2) or
 - (6.4.2) — a function call operator (12.4.2.2.3), but not a surrogate call function;
- (6.5) — a **co_await** expression (7.6.2.4), a *yield-expression* (7.6.17), or a **co_return** statement (8.7.5);
- (6.6) — dynamic initialization of a block-scope variable with static storage duration; or
- (6.7) — dynamic initialization of a variable with thread storage duration. [*Note*: That includes the case when such an initialization is evaluated within an atomic block because the initialization was deferred (6.9.3.3). -- *end note*]

[*Note 2*: The implementation can define that the behavior is undefined in some or all of the cases above. — *end note*]

[*Example 1*:

```
unsigned int f()
{
    static unsigned int i = 0;
    atomic do {
        ++i;
        return i;
    }
}
```

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

[*Note 3*: Atomic blocks are likely to perform best where they execute quickly and touch little data. — *end note*]

15 Preprocessor

[cpp]

15.11 Predefined macro names

[cpp.predefined]

Add a row to Table 19 in C++20 §15.11:

Table 19: Feature-test macros

Macro name	Value
<code>__cpp_transactional_memory</code>	202602

16 Library introduction

[library]

16.4 Library-wide requirements

[requirements]

Add a new subclause after C++20 §16.4.6.16:

16.4.6.17 Functions usable in an atomic block

[atomic.use]

- 1 All library functions may be used in an atomic block (8.8), except
- (1.1) — error category objects (19.5.3.5)
 - (1.2) — time zone database (19.5.3.5)
 - (1.3) — clocks (27.7)
 - (1.4) — `signal` (17.13.5) and `raise` (17.13.4)
 - (1.5) — `set_new_handler`, `set_terminate`, `get_new_handler`, `get_terminate` (16.4.5.7, 17.6.4, 17.9.2)
 - (1.6) — `system` (17.2.2)
 - (1.7) — startup and termination (17.5) except `abort`
 - (1.8) — `shared_ptr` (20.11.3) and `weak_ptr` (20.11.4)
 - (1.9) — `synchronized_pool_resource` (20.12.5)
 - (1.10) — program-wide `memory_resource` objects (20.12.4)
 - (1.11) — `setjmp` / `longjmp` (17.13.3)
 - (1.12) — parallel algorithms (25.3)
 - (1.13) — `random_device` (26.6.7)
 - (1.14) — `locale` construction (28.3.1.3)
 - (1.15) — `locale::global` (28.3.1.6)
 - (1.16) — input/output (Clause 29)
 - (1.17) — atomic operations (Clause 31)
 - (1.18) — thread support (Clause 32)