**Document Number:** Nxxx
**Date:** 2022-01-09
**Revises:** None
**Reply to:** Michael Wong
Codeplay
fraggamuffin@gmail.com

# Working Draft, Extensions to C++for Transactional Memory Version 2

**Note: this is an early draft. It's known to be incomplet and incorrekt, and it has lots of bad formatting.**

# Contents

# Foreword [foreword]

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents) or the IEC list of patent declarations received (see http://patents.iec.ch).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation on the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT), see www.iso.org/iso/foreword.html.

This document was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces.*

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html.

# 1 Scope [scope]

1 This document describes requirements for implementations of an interface that computer programs written in the C++ programming language may use to invoke algorithms with concurrent execution. The algorithms described by this document are realizable across a broad class of computer architectures.

2 ISO/IEC 14882:2020 provide 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 ~~strikethrough~~ 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 concurrency in the C++ standard algorithms library. It gives advice on extensions to those vendors who wish to provide them.

# 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". The library described in C++20 §16-32 is herein called the C++ Standard Library.

3 Unless otherwise specified, the whole of the C++ Standard's Library introduction (C++20 §16) is included into this Technical Specification by reference.

4 Unless otherwise specified, these wording changes applies P2066R10 relative to C++ Working Draft, N4849.

# 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 Namespaces and headers and modifications to standard classes [general.namespaces]

1 Since the extensions described in this technical specification are experimental and not part of the C++ standard library, they are not declared directly within namespace `std`. Unless otherwise specified, all components described in this technical specification either:

(1.1) — modify an existing interface in the C++ Standard Library in-place,

(1.2) — are declared in a namespace whose name appends `::experimental::transactional_memory_v2` to a namespace defined in the C++ Standard Library, such as `std`, or

(1.3) — are declared in a subnamespace of a namespace described in the previous bullet, whose name is not the same as an existing subnamespace of namespace `std`.

2 Whenever an unqualified name is used in the specification of a declaration `D`, its meaning is established as-if by performing unqualified name lookup in the context of `D`.

[*Note 1*: Argument-dependent lookup is not performed. — *end note*]

Similarly, the meaning of a *qualified-id* is established as-if by performing qualified name lookup in the context of `D`.

[*Note 2*: Operators in expressions are not so constrained. — *end note*]

## 4.3 Feature-testing recommendations (Informative) [general.feature.test]

1 An implementation that provides support for this document should define each feature test macro defined in Table 1 if no associated headers are indicated for that macro, and if associated headers are indicated for a macro, that macro is defined after inclusion of one of the corresponding headers specified in the table.

Table 1: Feature-test macros

| Macro name | Value | Header |
|---|---|---|
| `__cpp_lib_transactional_memory_v2` | `202110` | `<experimental/transactional_memory_v2>` |

## 4.4 Future plans (Informative) [general.plans]

1 This section describes tentative plans for future versions of this technical specification and plans for moving content into future versions of the C++ Standard.

2 The C++ committee intends to release a new version of this technical specification approximately every few years, containing the transactional_memory extensions we hope to add to a near-future version of the C++ Standard. Future versions will define their contents in `std::experimental::transactional_memory_v3`, `std::experimental::transactional_memory_v4`, etc., with the most recent implemented version inlined into `std::experimental`.

3 When an extension defined in this or a future version of this technical specification represents enough existing practice, it will be moved into the next version of the C++ Standard by removing the `experimental::transactional_memory_v`*N* segment of its namespace and by removing the `experimental/` prefix from its header's path.

## 4.5 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, Hans Boehm, and Michael Spears. We also wish to thank

people who made valuable contributions within and outside these groups, including Jens Maurer, and many others not named here who contributed to the discussion.

# 5 Lexical conventions [lex]

## 5.1 Identifiers [lex.name]

1 In 5.10 [lex.name], add `atomic` to table 4 [tab:lex.name.special].

# 6 Basics [basic]

## 6.1 Program execution [basic.exec]

### 6.1.1 Sequential execution [intro.execution]

1 Change in 6.9.1 [intro.execution] paragraph 5:

A *full-expression* is

(1.1) — ...

(1.2) — 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~~

(1.3) — the start and the end of an atomic block (8.8 [stmt.tx]), or

(1.4) — an expression that is not a subexpression of another expression and that is not otherwise part of a full-expression.

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

#### 6.1.2.1 Data races [intro.races]

1 Change in 6.9.2.1 [intro.races] paragraph 6:

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

2 Add a new paragraph after 6.9.2.1 [intro.races] paragraph 20:

An atomic block that is not dynamically nested within another atomic block is termed a transaction. [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 transactions. If, in that total order, a transaction T1 is ordered before a transaction T2, then

(2.1) — no evaluation in T2 happens before any evaluation in T1 and

(2.2) — if T1 and T2 perform conflicting expression evaluations, then the end of T1 synchronizes with the start of T2.

[ Note: If the evaluations in T1 and T2 do not conflict, they might be executed concurrently. -- end note ]

Two actions are *potentially concurrent* if ...

3 Change in 6.9.2.1 [intro.races] paragraph 21:

... [Note: 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". ...

4 Add a new paragraph after 6.9.2.1 [intro.races] paragraph 21:

[ Note: 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 ]

### 6.1.2.2 Forward progress [intro.progress]

1 Change in 6.9.2.2 [intro.progress] paragraph 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 ([atomics]).

The implementation may assume that any thread will eventually terminate or evaluate an inter-thread side effect. [Note: This is intended to allow compiler transformations such as removal of empty loops, even when termination cannot be proven. — end note]

# 7 Statements [stmt.stmt]

## 7.1 Preamble [stmt.pre]

1 Add a production to the grammar in 8.1 [expr.pre]:

```
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
      \uline{atomic-statement}
```

## 7.2 Declaration statement [stmt.dcl]

1 Add a new subclause before 8.8 [stmt.dcl]:

### 8.8 Atomic statement [stmt.tx]

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

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

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 8.7 [stmt.jump]. -- end note ]

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

If the execution of an atomic block evaluates an inter-thread side effect (6.9.2.2 [intro.progress]) or if an atomic block is exited via an exception, the behavior is undefined.

*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 [exception.terminate]) or destroying any objects with static or thread storage duration (6.9.3.4 [basic.start.term]).

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

(1.1) — an *asm-declaration* (9.10 [dcl.asm]);
(1.2) — an invocation of a function **other than one of the standard library functions specified in (16.4.6.17 [atomic.use]), unless the function is inline** with a reachable definition;
(1.3) — a virtual function call (7.6.1.3 [expr.call]);
(1.4) — a function call, **unless overload resolution selects**
(1.4.1) — **a named function (12.2.2.2.2 [over.call.func]) or**
(1.4.2) — **a function call operator (12.2.2.2.3 [over.call.object]), but not a surrogate call function;**
(1.5) — a `co_await` expression (7.6.2.3 [expr.await]), a *yield-expression* (7.6.17 [expr.yield]), or a `co_return` statement (8.7.4 [stmt.return.coroutine]);
(1.6) — dynamic initialization of a block-scope variable with static storage duration; or
(1.7) — dynamic initialization of a variable with thread storage duration.

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

[ Example:

```
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: Atomic blocks are likely to perform best where they execute quickly and touch little data. -- end note ]

## 7.3 Library introduction [library]

### 7.3.1 Method of description [description]

[1] Add 16.4.6.17 [atomic.use]:

#### 7.3.2 16.4.6.17 Functions usable in an atomic block [atomic.use]

All library functions may be used in an atomic block (8.8 [stmt.tx]), except

- **error category objects ([syserr.errcat.objects])**
- time zone database ([time.zone.db])
- clocks ([time.clock])
- `signal` ([support.signal]) **and `raise` ([csignal.syn])**
- `set_new_handler`, `set_terminate`, `get_new_handler`, `get_terminate` ([handler.functions], [alloc.errors], [exception.syn])
- **`system` ([cstdlib.syn])**
- **startup and termination [support.start.term] except `abort`**
- `shared_ptr` ([util.smartptr.shared]) and `weak_ptr` ([util.smartptr.weak])
- `synchronized_pool_resource` ([mem.res.pool])
- **program-wide `memory_resource` objects ([mem.res.global])**
- `setjmp` / `longjmp` ([csetjmp.syn])
- **parallel algorithms ([algorithms.parallel])**
- **`random_device` ([rand.device])**
- `locale` construction ([locale.cons])
- input/output ([input.output])
- atomic operations ([atomics])
- thread support ([thread])