Standard Wording for Transactional Memory Support for C++

Authors:	Victor Luchangco, victor.luchangco@oracle.com
	Jens Maurer, jens.maurer@gmx.net
	Michael Wong, michaelw@ca.ibm.com
	with other members of the transactional memory study group (SG5), including:
	Hans Boehm, hans.boehm@hp.com
	Justin Gottschlich, justin.gottschlich@intel.com
	Paul McKenney, paulmck@linux.vnet.ibm.com
	Maged Michael, maged.michael@gmail.com
	Mark Moir, mark.moir@oracle.com
	Torvald Riegel, triegel@redhat.com
	Michael Scott, scott@cs.rochester.edu
	Tatiana Shpeisman, tatiana.shpeisman@intel.com
	Michael Spear, spear@cse.lehigh.edu
Document number:	N3999
Date:	2014-05-23
Project:	Programming Language C++, Evolution Working Group, Core Working Group
Reply-to:	Michael Wong, michaelw@ca.ibm.com (Chair of SG5)
Revision:	1

1 Introduction

Transactional memory supports a programming style that is intended to facilitate parallel execution with a comparatively gentle learning curve. This document describes a proposal developed by SG5 to introduce transactional constructs into C++ as a Technical Specification.

This proposal is based in part on the *Draft Specification for Transactional Constructs in C++ (Version 1.1)* published by the Transactional Memory Specification Drafting Group in February 2012. It represents a pragmatic basic set of features, and omits or simplifies a number of controversial or complicated features from the *Draft Specification*. Our goal has been to focus SG5's efforts towards a basic set of features that is useful and can support progress towards possible inclusion in the C++ standard.

In addition to a description of the proposal, this document contains a summary of the issues discussed Section 1.1 as well as Standard Wording for Core Section 8. Library wording is handled in a separate document.

1.1 Changes in previous revisions

- N3999 (this paper): From N3919, we updated with discussion results from EWG, and the TM Evening Session of Issaquah. We considered the memory model effect from Chandler on whether a transaction that does not access any shared data induces any happens-before arcs – and in particular whether transactions that are provably thread-local must incur synchronization costs. We describe the result of that discussion in Section 7.1.

  We also added Core Standard Wording to support TM TS in preparation for the Draft Document. Core Standard Wording will be reviewed in a separate telecon call on June 2, and likely again at Rapperswil. Library wording is posted in N4000 and will be reviewed in Rapperswil.

- N3919: From 3859, this was the EWG approved version that was also voted in Issaquah as the indicated document for the TM TS NP.

  We added examples, and modified the syntax as directed by EWG vote.

- N3859: From N3718, we made several changes based on feedback and discussion at and since the Chicago meeting, including:

  - The *atomic transactions* and *relaxed transactions* of N3718 were renamed atomic blocks and synchronized blocks respectively (and the relevant keywords were changed to refect this).
  - Synchronized blocks are defined before and without reference to atomic blocks.
  - We eliminated the use of *escape* to refer to an exception being thrown but not caught within a transaction (and the relevant keywords were changed to reflect this).
  - Synchronized blocks may be nested within atomic blocks. (In N3718, atomic transactions could be nested within relaxed transactions, but relaxed transactions could not be nested within atomic transactions.)
  - We decided to expand the set of functions in the standard library designated transaction-safe, and the set of exceptions that can cancel an atomic block. (This document does not fully reflect the intended change; instead, it includes relevant comments at appropriate places.)

We believe that these changes clarify the concepts being defined and may facilitate programmers' adoption of synchronized and atomic blocks. See Section 7 for further discussion.

2   Overview

We introduce two kinds of blocks to exploit transactional memory: *synchronized blocks* and *atomic blocks*. Synchronized blocks behave as if all synchronized blocks were protected by a single global recursive mutex. Atomic blocks (also called *atomic transactions*, or just *transactions*) appear to execute atomically and not concurrently with any synchronized block (unless the atomic block is executed within the synchronized block). Some operations are prohibited within atomic blocks because it may be impossible, difficult, or expensive to support executing them in atomic blocks; such operations are called *transaction-unsafe*. An atomic block also specifies how to handle an exception thrown but not caught within the atomic block.

Some noteworthy points about synchronized and atomic blocks:

Data races Operations executed within synchronized or atomic blocks do not form data races with each other. However, they may form data races with operations not executed within any synchronized or atomic block. As usual, programs with data races have undefined semantics.

Exceptions When an exception is thrown but not caught within an atomic block, the effects of operations executed within the block may take effect or be discarded, or `std::abort` may be called. This behavior is specified by an additional keyword in the atomic block statement, as described in Section 4. An atomic block whose effects are discarded is said to be *canceled*. An atomic block that completes without its effects being discarded, and without calling `std::abort`, is said to be *committed*.

Transaction-safety As mentioned above, transaction-unsafe operations are prohibited within an atomic block. This restriction applies not only to code in the body of an atomic block, but also to code in the body of functions called (directly or indirectly) within the atomic block. To support static checking of this restriction, we introduce a keyword to declare that a function or function pointer is transaction-safe, and augment the type of a function or function pointer to specify whether it is transaction-safe. We also introduce an attribute to explicitly declare that a function is *not* transaction-safe.

To reduce the burden of declaring functions transaction-safe, a function is assumed to be transaction-safe if its definition does not contain any transaction-unsafe code and it is not explicitly declared transaction-unsafe. Furthermore, unless declared otherwise, a non-virtual function whose definition is unavailable is assumed to be transaction-safe. (This assumption does *not* apply to virtual functions because the callee is not generally known statically to the caller.) These assumptions are checked at link time.

3   Synchronized Blocks

A *synchronized block* has the following form:

      `synchronized` { *body* }

The evaluation of any synchronized block synchronizes with every evaluation of any synchronized block (whether it is an evaluation of the same block or a different one) by another thread, so that the evaluations of non-nested synchronized blocks across all threads are totally ordered by the synchronizes-with relation. That is, the semantics of a synchronized block is equivalent to having a single global recursive mutex that is acquired before executing the body and released after the body is executed (unless the synchronized block is nested within another synchronized block). Thus, an operation within a synchronized block never forms a data race with any other operation within a synchronized block (the same block or a different one).

Note: *Entering and exiting a nested synchronized block (i.e., a synchronized block within another synchronized block) has no effect.*

Jumping into the body of a synchronized block using `goto` or `switch` is prohibited.

Use of synchronized blocks   Synchronized blocks are intended in part to address some of the difficulties with using mutexes for synchronizing memory access by raising the level of abstraction and providing greater implementation flexibility. (See *Generic Programming Needs Transactional Memory* by Gottschlich and Boehm in Transact 2013 for a discussion of some of these issues.) With synchronized blocks, a programmer need not associate locks with memory locations, nor obey a locking discipline to avoid deadlock: Deadlock cannot occur if synchronized blocks are the only synchronization mechanism used in a program.

Although synchronized blocks can be implemented using a single global mutex, we expect that some implementations of synchronized blocks will exploit recent hardware and software mechanisms for transactional memory to improve performance relative to mutex-based synchronization. For example, threads may use speculation and conflict detection to evaluate synchronized blocks concurrently, discarding speculative outcomes if conflict is detected. Programmers should still endeavor to reduce the size of synchronized blocks and the conflicts between synchronized blocks: poor performance is likely if synchronized blocks are too large or concurrent conflicting evaluations of synchronized blocks are common. In addition, certain operations, such as I/O, cannot be executed speculatively, so their use within synchronized blocks may hurt performance.

4   Atomic Blocks

An *atomic block* can be written in one of the following forms:

      `atomic_noexcept` { *body* }
      `atomic_commit` { *body* }
      `atomic_cancel` { *body* }

The keyword following `atomic` is the atomic block's *exception specifier*. It specifies the behavior when an exception escapes the transaction:

`atomic_noexcept`: This is undefined behavior and is not allowed; no side effects of the transaction can be observed.

`atomic_commit`: The transaction is committed and the exception is thrown.

`atomic_cancel`: If the exception is transaction-safe (defined below), the transaction is canceled and the exception is thrown. Otherwise, it is undefined behavior. In either case, no side effects of the transaction can be observed.

An exception is *transaction-safe* if its type is `bad_alloc`, `bad_array_length`, `bad_array_new_length`, `bad_cast`, `bad_typeid`, or a scalar type.

Comment: *Based on feedback on our earlier proposal (i.e., N3718), we intend to expand the set of transaction-safe exceptions. However, we have not investigated this sufficiently to expand it at this time.*

Code within the body of a transaction must be *transaction-safe*; that is, it must not be transaction-unsafe. Code is *transaction-unsafe* if:

- it contains an initialization of, assignment to, or a read from a volatile object;

- it is a transaction-unsafe `asm` declaration (the definition of a transaction-unsafe `asm` declaration is implementation-defined); or

- it contains a call to a transaction-unsafe function, or through a function pointer that is not transaction-safe (see Section 5).

Note: *The dynamic initialization of function-local statics is transaction-safe (assuming the code in the initialization expression is transaction-safe) even though it likely involves some nonatomic synchronization under the covers. However, see Section 6.*

Note: *Synchronization via locks and atomic objects is not allowed within atomic blocks (operations on these objects are calls to transaction-unsafe functions).*

Comment: *This restriction may be relaxed in a future revision of the Technical Specification.*

Jumping into the body of an atomic block using `goto` or `switch` is prohibited.

The body of an atomic block appears to take effect atomically: no other thread sees any intermediate state of an atomic block, nor does the thread executing an atomic block see the effects of any operation of other threads interleaved between the steps within the atomic block.

The evaluation of any atomic block synchronizes with every evaluation of any atomic or synchronized block by another thread, so that the evaluations of non-nested atomic and synchronized blocks across all threads are totally ordered by the synchronizes-with relation. Thus, a memory access within an atomic block does not race with any other memory access in an atomic or synchronized block. However, a memory access within an atomic block may race with conflicting memory accesses not within any atomic or synchronized block. The exact rules for defining data races are defined by the memory model.

Note: *As usual, programs with data races have undefined semantics.*

Note: *Although it has no observable effects, a canceled atomic block may still participate in data races.*

Note: *This proposal provides "closed nesting" semantics for nested atomic blocks.*[1]

Use of atomic blocks   Atomic blocks are intended in part to replace many uses of mutexes for synchronizing memory access, simplifying the code and avoiding many problems introduced by mutexes (e.g., deadlock). We expect that some implementations of atomic blocks will exploit hardware and software transactional memory mechanisms to improve performance relative to mutex-based synchronization. Nonetheless, programmers should still endeavor to reduce the size of atomic blocks and the conflicts among atomic blocks and with synchronized blocks: poor performance is likely if atomic blocks are too large or concurrent conflicting executions of atomic and synchronized blocks are common.

5   Transaction-Safety for Functions

A function declaration may specify the `transaction_safe` keyword or the `transaction_unsafe` attribute.

Declarations of function pointers and typedef declarations involving function pointers may specify the `transaction_safe` keyword (but not the `transaction_unsafe` attribute).

A function is *transaction-unsafe* if

- any of its declarations specifies the `transaction_unsafe` attribute,

- it is a virtual function that does not specify the `transaction_safe` keyword and does not override a function whose declaration specifies the `transaction_safe` keyword,

- any of its parameters are declared volatile,

- it is a constructor or destructor whose corresponding class has a non-static volatile data member, or

- its definition contains transaction-unsafe code as defined in Section 4.

Note: *This definition covers lambdas and implicitly defined member functions.*

Note: *A function with multiple declarations is transaction-unsafe if any of its declarations satisfies the definition above.*

No declaration of a transaction-unsafe function may specify the `transaction_safe` keyword. A function is *transaction-safe* if it is not transaction-unsafe. The transaction-safety of a function is part of its type.

Note: *A transaction-safe function cannot overload a transaction-unsafe function with the same signature, and vice versa.*

A function pointer is *transaction-safe* if it is declared with the `transaction_safe` keyword. A call through a function pointer is transaction-unsafe unless the function pointer is transaction-safe.

A transaction-safe function pointer is implicitly convertible to an ordinary (i.e., not transaction-safe) function pointer; such conversion is treated as an identity conversion in overloading resolution.

A compiler-generated constructor/destructor/assignment operator for a class is transaction-unsafe if any of the corresponding operations on any of the class's direct base classes is transaction-unsafe.

---

[1]For a description of closed nesting, see *Transactional Memory* by Harris, Larus and Rajwar, for example.

A member function declared with the `transaction_safe` keyword or `transaction_unsafe` attribute in a base class preserves that attribute in any derived class, unless that member is redefined or overridden. Functions brought into a class via a `using` declaration preserve the attribute in the original scope. A virtual function of transaction-safe type must not be overridden by a virtual function of transaction-unsafe type.

Because a compilation unit might not contain all declarations of a function, the transaction safety of a function is confirmed only at link time in some cases.

Transaction-Safety of Functions in the Standard Library Certain functions in the standard library are designated as transaction-safe. See N3862 for details.

Allocation and deallocation functions are necessarily transaction-unsafe, and user-defined `new` and `delete` functions are also transaction-unsafe. See Section 7.8 for further discussion of this issue.

6 Outstanding Issues

We have yet to resolve two outstanding issues from N3718:

- The proposed wording changes for the memory model in N3718 assume that code that could observe a violation of the atomicity of an atomic transaction would necessarily form a data race with that transaction. However, this assumption is invalid because concurrent initialization of function-local static variables are not racy. We think we should add wording to the memory model specification to directly ensure the atomicity of atomic blocks.

- We specify that when the exception assumptions of an atomic block are violated, `std::abort` rather than `std::terminate` is called. This avoids the question of the status of the transaction from the perspective of a termination handler. However, in some cases, normal exception behavior (independent of atomic blocks) requires `std::terminate` to be called, raising the question we hoped to avoid.

See Section 7.14 of N3718 for more discussion of these issues.

7 Discussion of Feedback and Other Issues

This proposal is a mostly expository and syntactic revision from the proposal in N3718, presented at the C++ meeting in Chicago in September 2013, incorporating feedback we received there and subsequent discussion within SG5. In this section, we summarize some of the feedback we received at the Chicago meeting and the subsequent discussion within SG5, which we hope will help readers understand the changes relative to N3718, and also some aspects we opted not to change. For issues discussed before the Chicago meeting, please refer to the relevant section of N3718.

7.1 Transactions that are provably thread-local should incur Synchronization

We considered the memory model effect from Chandler on whether a transaction that does not access any shared data induces any happens-before arcs and in particular whether transactions that are provably thread-local must incur synchronization costs.

While we have not come to a definitive conclusion, we have considered the following options:

1. Leave semantics of atomic transactions as it is - simple, but potentially imposing unnecessary SC fence cost when transactions do not access shared memory.

2. Keep global ordering but explicitly allow elimination of transactions that do not access shared memory just because we said so.

3. Use weaker ordering that do not guarantee global order but naturally allows elimination of transactions that do not access shared memory

4. Stick to 1 for atomic transactions and use a new keyword for option 2 or 3.

This continues to be discussed and we anticipate if any changes need to be made, a paper will be written against the TS draft.

7.2   Making the proposal more modular

At the Chicago meeting, several people expressed concern that our transactions were unnecessarily entangling separable notions of *synchronization* and *exception safety*, the former having to do with coordinating access to shared memory and the latter with dealing with exceptional cases. They also worried that enabling atomic transactions to provide exception safety was a substantial undertaking that would hold up progress on the more pressing need for the synchronization provided by relaxed transactions (everyone, it seemed, agreed that mutexes were inadequate).

After much discussion, we agreed to clarify that atomic transactions and relaxed transactions embodied rather different ideas, and that an implementation could usefully provide relaxed transactions without atomic transactions, and to achieve this by specifying relaxed transactions without reference to atomic transactions. We also thought that we could avoid some confusion by not referring to both as transactions, and thus started to refer to relaxed transactions as *synchronized blocks*. (See Section 7.3 for more discussion on terminology.)

In rewriting the proposal, we considered whether we ought to emphasize the distinction even more by splitting the proposal into two, one describing synchronized blocks and the other describing atomic transactions (and transaction-safety). However, there was unanimous consensus against this among the active participants of SG5. Among the cited concerns were that it could derail momentum on atomic transactions and that splitting synchronized blocks into a separate proposal that doesn't reference atomic transactions might lead to their developing without due consideration of their intended interaction with atomic transactions. We also noted that we explicitly chose to propose a Technical Specification because we expect the proposal to evolve as we gain experience with these constructs as they are actually used by programmers. As such, implementors need not implement the entire proposal for compliance, but we should not encourage them to abandon atomic transactions by splitting them off entirely.

7.3   Terminology

As mentioned above, we found that the term *transaction* evoked different notions to different people, and thus sometimes introduced more confusion than intuition, particularly in its use to refer to both atomic and relaxed transactions. Also, since we decided to describe relaxed transactions prior to and without reference to atomic transactions, it would be confusing to call them *relaxed* transactions. Therefore, we renamed them *synchronized blocks*, following the intuition expressed by several people at the Chicago meeting that they provide what they think of as synchronization. We have reservations about this term, in part because of its use in Java, but have not yet found a preferred alternative.

After revising the proposal as described above, some people opined that we ought to also rename atomic transactions (or just *transactions* at that point). We decided to revert back to an early syntax, in which transactions were introduced by the keyword `atomic` (this had been abandoned in the *Draft Specification*, and in N3718, in part to unify atomic and relaxed transactions, but had been revived by some at the Chicago meeting). A concern with using `atomic` is whether it would introduce syntactic or conceptual confusion

with C++ atomic objects. However, we think that place in the grammar is sufficiently distinguished to avoid syntactic confusion, and that conceptual connection to atomic objects may actually be an advantage, because the concepts are similar. With this syntactic change, we called the blocks *atomic blocks*.

The use of *escape* to refer to an exception thrown but not caught within a block seemed to confuse more than aid understanding at the Chicago meeting, so we eliminated it in this revision.

7.4   Nesting synchronized blocks within atomic blocks

In N3718, relaxed transactions were transaction-unsafe, and so could not be executed within an atomic transaction. This restriction was natural in N3718 because the motivation for relaxed transactions was to allow a transaction to contain transaction-unsafe code: a transaction guaranteed to execute only transaction-safe code (as any code executed within an atomic transaction must be) should be designated an atomic transaction rather than a relaxed transaction. However, this restriction is not necessary, and is less compelling in this proposal, which more cleanly distinguishes synchronized blocks and atomic blocks, and specifically envisions that synchronized blocks might be implemented prior to atomic blocks, so that code intended to be atomic might be in a synchronized block rather than an atomic block. Thus, we agreed to allow synchronized blocks within atomic blocks.

7.5   Static checking of transaction-safety

The value to statically check the transaction-safety was questioned at the Chicago meeting, particularly because the `transaction_safe` keyword and changes to function and function-pointer types are needed only to enable this static check. Herb Sutter noted that in the past, when presented with the choice of introducing linguistic complexity or forgoing static checking, the C++ committee has typically favored forgoing static checking. Nonetheless, most of SG5 felt that there was significant value to static checking, not only in the greater assurance it provides, but also in helping to "transactionalize" existing programs safely. (See *Transactionalizing Legacy Code* by Vyas, Liu and Spear in Transact 2013, for example.) If we ultimately decide to forgo static checking, then it may be helpful to introduce an optional annotation that enables static checking by a separate tool.

7.6   Choice of transaction-safe standard library functions

In N3718, we noted that we deliberately chose to designate only a small set of standard library fuctions as transaction-safe. Our reasoning was that this minimized work that had to be done immediately and provided greater flexibility for the future. (See Section 7.5 of N3718 for more discussion of this point.) However, people at the Chicago meeting pointed out several people at the Chicago meeting pointed out that this restriction may hinder adoption. Therefore, we have begun investigating how to make more of the standard library transaction-safe. This issue is discussed in a separate document (N3862).

7.7   Choice of transaction-safe exceptions

In N3718, we also noted that we deliberately defined a small set of exceptions as transaction-safe, with the intention to expand this as necessary. As with the standard library functions, we intend to revisit this decision and expland this set more aggressively.

7.8   Allocation and deallocation within atomic blocks

Standard containers are a vital building block of most application programs, and use of containers should be permitted within atomic blocks. Even the most simple of containers, such as `std::list` or `std::vector`,

allocate and free memory for basic operations such as adding an element to the container. In some cases—at essentially unpredictable times—an allocation function must call out to the operating system to obtain additional storage. However, operating system calls are transaction-unsafe, which conflicts with the goal of supporting memory allocation within atomic blocks.

One implementation technique (used in gcc) is to abort (cancel) hardware-based transaction execution upon encountering a call to an allocation function and redo the transaction in software. The transaction is suspended while the allocation function executes, i.e. the allocation function does not execute in transaction context. When the allocation function returns the requested memory range, transactional execution of the calling code resumes. If that execution eventually encounters a need to cancel the transaction, a previously registered cleanup handler deallocates the memory.

This machinery would be entirely invisible to the application, except that C++ provides for user-replaceable allocation and deallocation functions (see Section 18.6 of the C++ standard). Repeated calls to these functions are thus observable. As a first step, spurious calls to allocation and deallocation functions (caused by transaction cancel and redo) must be expressly allowed, provided they occur in matched pairs.

As a next step, it might be desirable to start executing the allocation function while still in (hardware) transaction context. In most cases, the requested memory is probably available in a thread-local quick-list, not requiring an operating system call. For such a fast-path, canceling the hardware transaction and redoing in software would be seriously detrimental to performance. Only when the quick-list is exhausted should execution drop out of the transaction to allow operating system calls for acquiring more memory.

What should the standard-specified interface for allocation functions look like? It certainly makes a difference if the allocation function starts executing within a surrounding transaction, or as-if entirely outside any transaction. In the former case, what is the mechanism to explicitly request suspending the transactional execution to permit operating system calls? What is the mechanism to register a cleanup handler for a potential subsequent cancel? Is there maybe general value in such mechanisms, beyond allocation functions? This is unexplored design space in the context of this proposal. The most conservative way forward is to disallow user replacement of allocation and deallocation functions if a call to such appears within a transaction anywhere in the program. This preserves all options for future evolution.

7.9 Transaction-safety as part of type for overloading resolution

In the current proposal, transaction-safety is not used in overloading resolution: it is not permissible to define two functions whose headers differ only in whether they are transaction-safe. At the Chicago meeting, Faisal Vali suggested that relaxing this restriction (i.e., by permitting transaction-safety to be used in overloading resolution) might address some problems encountered with the current proposal. In particular, this would allow a function to use one definition when called within an atomic block and another when called not within an atomic block. Such functionality would also enable programmers to take different paths depending on whether they are executing within an atomic block. However, supporting such a change will likely have ripple effects throughout the system, and thus requires further study before it can be incorporated into this proposal.

7.10 Restricting code permitted within synchronized blocks

We may want to restrict the operations that may be executed within a synchronized block, to enable certain implementation techniques and/or reduce the likelihood of unacceptable performance. For example, we may forbid I/O within a synchronized block, which would invalidate any speculative evaluation of the block.

Unless function calls are forbidden within synchronized blocks, statically checking any such restriction would likely require extending function types. As discussed in Section 7.5, such extension may receive pushback from the C++ committee.

7.11   Domains for synchronized blocks

Several people expressed concern that although "single global lock" semantics may seem attractive for its simplicity, it may induce too much synchronization in realistic large programs. Thus, it may be desirable to reduce synchronization by allowing synchronized blocks to specify "*domains*", and requiring (evaluations of) synchronized blocks to be ordered only if they have the same domain. In that case, a programmer could specify different domains for synchronized blocks known to never conflict. We might allow a synchronized block to specify multiple domains, and a synchronized block that does not specify any domain might be considered to have every domain (i.e., its evaluations must be ordered with respect to the evaluations of every other synchronized block).

Syntactically, the domain might be specified as an "argument" between the synchronized keyword and the body of the block:

```
synchronized (domain) { body }
```

If every synchronized block has exactly one domain (and the system does not recognize any relation among the domains), then this is logically equivalent to having different mutexes for each domain. However, this notion is more general than mutexes if synchronized blocks can specify multiple domains.

Introducing domains reintroduces many of the difficulties of mutex-based synchronization. For example, domains would likely be associated with memory locations, and the programmer would need to determine the correct domain(s) for each synchronized block. Also, a nested synchronization block may introduce deadlock, unless it specifies only domains specified by its (dynamically) enclosing synchronization block. If we impose such a restriction, then a synchronized block specifying a domain not specified by its enclosing synchronized block would be unsafe code, as discussed in Section 7.10.

If we do have a notion of unsafe code, and the definition of unsafe code ensures that a safe synchronized block (i.e., one that does not execute unsafe code) can be rolled back, then some of the problems with having multiple domains can be avoided for safe synchronized blocks. In particular, deadlock can be avoided. In this case, it is important to specify when the semantics guarantees that deadlock is avoided (whether it is statically checked).

7.12   Condition synchronization

Recent experience suggests that support for condition synchronization would be helpful. (See, for example, the experience reports by Vyas, Liu and Spear and by Skyrme and Rodriguez in Transact 2013.) As specified, we cannot use condition variables for this purpose because a synchronized block does not specify a mutex that can be passed to the condition variable. Although there is interest in providing such a mechanism, we do not yet have a specific proposal for it.

8   Proposed Core Standard Wording

Issues

Open issues

- `std::exception` is the base of the exception hierarchy. Should we declare its virtual what() function `transaction_safe`? [no; instead review definition of derived classes such as length_error] This has serious ripple effects to user code, in particular if that user code is totally unaware of transactions.
- Introduce a
- `template<class T>`
- `class txexception : exception { ... };`

  with a transaction-safe "what()" function and where "T" can be memcpy'd.

- Exception handling might call std::terminate, which invokes a terminate handler. That terminate handler is a function pointer not declared transaction-safe, so there is a gap in the specification if std::terminate ends up being called inside an atomic transaction. The intersection of desirable and implementable semantics for this case is still under discussion.
- Extend the library specification to prescribe standard-library containers plus `std::string` as transaction-safe. [see guidlines
- Initialization of variables with static storage duration.
- For "similar" in 4.4 [conv.qual] and for "composite pointer type" in 5 [expr], address that T might be "pointer to function" (not transaction-safe) vs. "pointer to transaction-safe function". This can be unified to "pointer to function" (not transaction-safe).
- Consider describing synchronized blocks entirely as-if taking a global mutex.
- At first sight, the math functions from <math.h> (e.g. "sin") are transaction-safe. However, they may need to inspect hardware state such as the current rounding mode that might make them unfit for transactional execution.
- Add concise examples.

Resolved issues

The wording below presents a decision, which might not represent the (evolving) consensus of the TM group. Please voice any concerns on the mailing list.

- Under which circumstances is a lambda function (implicitly) declared transaction-safe? [It is declared tx-safe if its definition is (directly) safe and all invoked functions are tx-safe.] Do we want to allow an explicit transaction_safe/unsafe annotation? [not supported right now]
- Is the term "transaction-statement" still ok, or should that be renamed to "atomic-statement", in line with the (new) spelling of the keywords? *Telco 2014-03-31: use "atomic block" and "transaction-safe".*

- Rephrase memory ordering constraints for transactions to focus on memory accesses, not syntactic start/end of the transaction. [Chandler Carruth in Issaquah] *Telco 2014-03-31: weaker semantics have a visible correctness cost for real-world programs, therefore the suggestion is rejected*

Wording

Section 1.10 [intro.multithread] Multi-threaded executions and data races

Add a paragraph 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.x [stmt.sync]) or atomic block (6.x [stmt.tx]) that is not dynamically nested within another synchronized block or atomic block is called an outer block. [Note: Due to syntactic constraints, such a block cannot overlap another one 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, then the end of T1 synchronizes with the start of T2.

Drafting notes: Together with 1.9p14, the first sentence ensures the appropriate (thread-local) sequencing. Inter-thread ordering is ensured by establishing a synchronizes-with relationship in the last sentence.

Change in 1.10 [intro.multithread] paragraph 21, and add a new paragraph following it:
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]

> **[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, 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]

Clause 2 [lex] Lexical conventions

In section 2.12 [lex.key], add the keywords `synchronized`, `atomic_noexcept`, `atomic_cancel`, and `atomic_commit` to the table.

Clause 4 [conv] Standard conversions

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." **Moreover, an lvalue of type "transaction-safe function" can be converted to a prvalue of type "pointer to function" (not transaction-safe).** The result is a pointer to the function. [Footnote: ...]

Drafting note: This ensures that overload resolution doesn't perceive dropping the "transaction-safe" as two conversions instead of just one. The same trick was applied for converting unscoped enumerations with fixed underlying type to the promoted underlying type (4.5p4).

Add a new section 4.14 [conv.tx]:
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" (i.e., not transaction-safe). The result is a pointer to the function.

Drafting note: Is there a need to convert a pointer to transaction-safe member function to a pointer to member function?

Clause 5 [expr] Expressions

Change in clause 5 [expr] paragraph 13:

- if either p1 or p2 is a null pointer constant, T2 or T1, respectively;
- ...

Change in 5.1.2 [expr.prim.lambda] paragraph 5:
... An *attribute-specifier-seq* in a *lambda-declarator* appertains to the type of the corresponding function call operator or operator template. **If the function call operator is not a template, it is declared transaction-safe if it does not have a transaction-unsafe definition (8.4.4 [dcl.fct.def.tx]) and if each function invoked as a potentially-evaluated expression in its function body is declared transaction-safe.** [Note: ...]
Change in 5.1.2 [expr.prim.lambda] paragraph 6:
The closure type for a non-generic *lambda-expression* with no *lambda-capture* has a public non-virtual non-explicit const conversion function to pointer to function with C++ language linkage (7.5 dcl.link) having the same parameter and return types as the closure type's function call

operator. **That pointer is a pointer to transaction-safe function if the function call operator is transaction-safe.**
Add 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 does not have transaction-safe linkage (8.4.4 [dcl.fct.def.tx]) through a pointer to or lvalue of type "transaction-safe function" is undefined.

Drafting note: This restriction might not be required if there is no defined way of obtaining a pointer to transaction-safe function from a pointer to (non-transaction-safe) function. One such way is precluded by the next change.

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. ...
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]). ...
Change in 5.16 [expr.con] 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). ...
- ...

Clause 6 [stmt.stmt] Statements

In 6 [stmt.stmt] paragraph 1, add a production 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
```

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]

Add a new section 6.x [stmt.sync]:

6.x [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.

Add a new section 6.x [stmt.tx]:

6.x [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 or if it invokes a function as a potentially-evaluated expression that does not have transaction-safe linkage (8.4.4 [dcl.fct.def.tx]); no diagnostic is required in the latter case.

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.

Section 7.4 [dcl.asm] The asm declaration

Change in 7.4 [dcl.asm] paragraph 1:
... The asm declaration is conditionally-supported; its meaning is implementation-defined. [Note: Typically it is used to pass information through the implementation to an assembler. -- end note] **It is implementation-defined which asm declarations are transaction-safe, if any.**

Section 8 [dcl.decl] Declarators

Change in clause 8 paragraph 4:

```
parameters-and-qualifiers:
     ( parameter-declaration-clause ) cv-qualifier-seq>opt
              ref-qualifieropt tx-safeopt transaction_unsafeopt exception-
specificationopt attribute-specifier-seqopt
```

and add a grammar production

```
tx-safe:
        transaction_safe
```

Change in 8.3.5 [dcl.fct] paragraphs 1 and 2:

In a declaration T D where D has the form

```
         D1 ( parameter-declaration-clause ) cv-qualifier-seqopt
                  ref-qualifieropt tx-safeopt transaction_unsafeopt exception-
specificationopt attribute-specifier-seqopt
```

and the type of the contained *declarator-id* in the declaration T D1 is "derived-declarator-type-list T", the type of the *declarator-id* in D is "derived-declarator-type-list $tx\text{-}safe_{opt}$ function of (parameter-declaration-clause) *cv-qualifier-seq$_{opt}$ ref-qualifier$_{opt}$* returning T". The optional *attribute-specifier-seq* appertains to the function type.

In a declaration T D where D has the form

```
         D1 ( parameter-declaration-clause ) cv-qualifier-seqopt
                  ref-qualifieropt tx-safeopt transaction_unsafeopt exception-
specificationopt attribute-specifier-seqopt trailing-return-type
```

and the type of the contained *declarator-id* in the declaration T D1 is "derived-declarator-type-list T", T shall be the single *type-specifier* auto. The type of the *declarator-id* in D is "derived-declarator-type-list $tx\text{-}safe_{opt}$ function of of (parameter-declaration-clause) *cv-qualifier-seq$_{opt}$ ref-qualifier$_{opt}$* returning *trailing-return-type*". The optional *attribute-specifier-seq* appertains to the function type.

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 *tx-safe*,** but not the default arguments (8.3.6 [dcl.fct.default])~~,~~ ~~or~~ the *exception specification* (15.4 [except.spec]), **or the *attribute-specifier-seq* (7.6 [dcl.attr]),** are part of the function type. ...

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

The first declaration of a function shall be explicitly declared `transaction_safe` if any declaration of that function is declared `transaction_safe`. The token `transaction_unsafe` shall only appear in the *declarator* of a function declaration; that function shall not be declared `transaction_safe`. [Note: The purpose of `transaction_unsafe` is to avoid code generation for transaction-safe and transaction-unsafe variants of a function that does not have a transaction-unsafe definition, if that function is never called inside an atomic block. -- end note]

[Drafting note: The (non-)semantics of "transaction_unsafe" are good for an attribute (or may ride on its grammar productions, similar to alignment specifiers), I believe, but transaction_safe needs to continue to be separate, since it changes the type system.]

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-seqopt
```

~~*ref-qualifier*~~_{~~opt~~} ~~*exception-specification*~~_{~~opt~~}
~~*attribute-specifier-seq*~~_{~~opt~~}
parameters-and-qualifiers *trailing-return-type*_{opt}

[Drafting note: This is intended to reduce the grammar redundancies around function declarators.]

Add a section after 8.4.3 [dcl.fct.def.delete]:

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,**
- **an invocation of a virtual function that is not declared transaction-safe (including implicit calls),**
- **a function call (5.2.2 [expr.call]) where a function or member function name is not used and the type of the *postfix-expression* does not refer to a "transaction-safe function", or**
- **a function call where a function or member function name is used and the function is declared `transaction_unsafe`.**

A statement is a *transaction-unsafe statement* if one of its lexically directly contained elements is one of the following:

- **an *expression* that is transaction-unsafe,**
- **an *asm-definition* (7.4 [dcl.asm]) that is not transaction-safe,**
- **a declaration of a volatile object, or**
- **a statement that is transaction-unsafe (recursively).**

[Drafting note: This wording is intended to recurse through the "statement" grammar, but not inside expressions. In particular, the compound-statement of a lambda determines the transaction-safety of the lambda's operator() function, but, unless called, does not influence the transaction-safety of the surrounding context.]

A function has a *transaction-unsafe definition* if

- **it is declared `transaction_unsafe`,**
- **any parameter is declared volatile,**
- **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**

17

- for a constructor, an *assignment-expression* in a *brace-or-equal-initializer* that is not ignored (12.6.2 class.base.init) or an expression in the *ctor-initializer* is transaction-unsafe.

[Drafting note: These definitions cover when a function is definitely not transaction-safe. The only remaining uncertainty is whether a called function is transaction-safe or not. This issue is addressed by the concept of "transaction-safe linkage" below.]

A function has transaction-safe linkage if it does not have a transaction-unsafe definition and each function invoked as a potentially-evaluated expression in its function body

- **is declared transaction-safe or**
- **has transaction-safe linkage (recursively).**

While determining whether a function `f` has transaction-safe linkage, `f` is assumed to have transaction-safe linkage for directly and indirectly recursive calls. [Example:

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

A function declared transaction-safe is ill-formed if it does not have transaction-safe linkage; no diagnostic is required unless the function has a transaction-unsafe definition.

[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
  i = *p;      // error: read through volatile glvalue
  S s;         // error: invocation of unsafe destructor
}
-- end example ]
```

Drafting note: Implicitly-defined special member functions and lambda expressions should be automatically covered by the wording above.

Section 10.3 [class.virtual] Virtual functions

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

A function that overrides a function declared `transaction_safe` is implicitly considered to be declared `transaction_safe`.

Clause 13 [over] Overloading

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

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

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 F1 is selected for the function type F of the target type required in the context if**

- **F1 (after possibly applying the transaction-safety conversion (4.14 [conv.tx])) is identical to F or**
- **F is "transaction-safe function" and F1 is not transaction-safe, but otherwise the same as F.**

If F is "transaction-safe function", F1 shall have transaction-safe linkage (8.4.4 [dcl.fct.def.tx]. [Note: ...]

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
]
```

Clause 14 [temp] Templates

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

An instantiation of a function template declared transaction_safe shall have transaction-safe linkage (8.4.4 dcl.fct.def.tx); no diagnostic required.

Clause 15 [except] Exception handling

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 **and still in scope (6.6 [stmt.jump], and atomic blocks are terminated whose start, but not end, was executed since the try block was entered (6.x [stmt.tx])**. The automatic objects are destroyed in the reverse order of the completion of their construction**, interleaved with terminating atomic blocks as-if the start of the transaction were the construction of an object with automatic storage duration**.

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

An atomic block is *terminated* according to its kind. Terminating an `atomic_commit` block has no effect. [Note: That is, control simply exits the transaction block. -- 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 transaction except those occupied by the temporary object are restored to the values they had at the time the start of the transaction 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 well as those types specified as such in clauses 18 and 19.

Standard library

Drafting note: The following guidelines were employed for transaction-safety requirements in the standard library, roughly oriented on the guidelines for `constexpr` *and* `noexcept`:

- If a function can unconditionally satisfy its contract without invoking user-defined code, it is declared `transaction_safe`. Functions declared `noexcept` strongly hint in that direction. Example: `size()` member function of containers.
- If a function is expected to call potentially user-defined code, that function is specified in prose to have transaction-safe linkage under the condition that all of the potentially invoked user-defined functions are transaction-safe. Example: A copy constructor of a container is only transaction-safe if all required functions of the Allocator are transaction-safe and if all required functions for the contained type T are transaction-safe, where "required function" is defined by the respective requirements tables.
- There is no code-level specification of conditional transaction-safety. When simply calling such functions, there is no issue, because the safe-by-default approach (specified as transaction-safe linkage) provides a transaction-enabled variant of a function whenever

possible. Only when the address of a function is taken (e.g.
`&std::vector<Foo>::push_back`) will the resulting pointer to (member) function type
not be transaction-safe, even though the function actually called might be.

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
- ...

Add a new section in 17.6.5 [conforming]:

17.6.5.16 [lib.txsafe] Transaction safety

For a function unconditionally specified to have transaction-safe linkage, an implementation may declare the function `transaction_safe`. Other functions shall not be declared `transaction_safe`.

Change in 18.5 [support.start.term] paragraph 4:
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).
- The `abort()` function is declared transaction-safe.

Add to 18.6.1 [new.delete] paragraph 1:
... The library versions of the global allocation and deallocation functions are declared transaction-safe.
Add a first paragraph to section 18.6.2 [alloc.errors]:
The classes `bad_alloc`, `bad_array_length`, and `bad_array_new_length` are transaction-safe.

For each declaration of a member function in 18.6.2.1 [bad.alloc], 18.6.2.2 [bad.array.length], and 18.6.2.3 [new.badlength], add `transaction_safe`.

Change in 18.7.2 [bad.cast]:
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.

For each declaration of a member function in 18.7.2 [bad.cast], add `transaction_safe`.

Change in 18.7.3 [bad.typeid]:

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.**

For each declaration of a member function in 18.7.3 [bad.typeid], add `transaction_safe`.

Change in 18.8.2 [bad.exception]:
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.**
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.x [stmt.sync]) or transaction block (6.x [stmt.tx])**.
Change in 19.2 [std.exceptions] paragraph 3:
... These exceptions are related by inheritance. **The exception classes support transaction cancellation.**

For each declaration of a constructor taking a pointer to const char in 19.2.x, add `transaction_safe`.

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

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: **Has transaction-safe linkage if the invoked constructor of U has transaction-safe linkage.**
```
template <class U>
  void destroy(U* p);
```
Effects: `p->~U()`
Transactions: **Has transaction-safe linkage if the destructor of U has transaction-safe linkage.**

Add after 20.8.13 [c.malloc] paragraph 2:

The contents are the same as the Standard C library header <stdlib.h>, with the following changes:

The functions are declared transaction-safe.

[Drafting note: This covers calloc, malloc, free, and realloc.]

22

Change in 20.8.13 [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 declared
> transaction-safe.**

[Drafting note: This covers memchr, memcmp, memcpy, memmove, and memset.]

Add after 21.1 [strings.general] paragraph 1:
**All functions in this Clause have transaction-safe linkage if the required operations on the
supplied allocator (17.6.3.5 allocator.requirements) and character traits (21.2.1
char.traits.require) have transaction-safe linkage.**

In 21.4.3 [string.iterators], 21.4.4 [string.capacity], 21.4.5 [string.access] add
"`transaction_safe`" to the declarations of all member functions.

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> are declared transaction-safe.

[Drafting note: This covers abs, ldiv, rand, div, llabs, srand, labs, and lldiv.]

9   Related Documents

Some related documents and papers are listed below:

N3341: *Transactional Language Constructs for C++*

N3422: *SG5: Software Transactional Memory (TM) Status Report*

N3423: *SG5: Software Transactional Memory (TM) Meeting Minutes*

N3529: *SG5: Transactional Memory (TM) Meeting Minutes 2012/10/30-2013/02/04*

N3544: *SG5: Transactional Memory (TM) Meeting Minutes 2013/02/25-2013/03/04*

N3589: *Summary of Progress Since Portland towards Transactional Language Constructs for C++*

N3591: *Summary of Discussions on Explicit Cancellation in Transactional Language Constructs for C++*

N3592: *Alternative cancellation and data escape mechanisms for transactions*

N3690: *Programming Languages — C++*

N3695: *SG5 Transactional Memory (TM) Meeting Minutes 2013/03/11-2013/06/10*

N3717: *SG5 Transactional Memory (TM) Meeting Minutes 2013/06/24-2013/08/26*

N3718: *Transactional Memory Support for C++* (an earlier version of this proposal)

N3725: *Original Draft Specification of Transactional Language Constructs for C++, Version 1.1 (February 3, 2012)*

N3862: *Transactionalizing the C++ Standard Library*

Ali-Reza Adl-Tabatabai, Victor Luchangco, Virendra J. Marathe, Mark Moir, Ravi Narayanaswamy, Yang Ni, Dan Nussbaum, Xinmin Tian, Adam Welc, Peng Wu. *Exceptions and Transactions in C++*. USENIX Workshop on Hot Topics in Parallelism (HotPar), 2009.

Justin Gottschlich, Hans Boehm. *Generic Programming Needs Transactional Memory*. Workshop on Transactional Computing (TRANSACT), 2013.

Trilok Vyas, Yujie Liu, Michael Spear. *Transactionalizing Legacy Code: An Experience Report Using GCC and Memcached*. Workshop on Transactional Computing (TRANSACT), 2013.

Alexandre Skyrme and Noemi Rodriguez. *From Locks to Transactional Memory: Lessons Learned from Porting a Real-World Application*. Workshop on Transactional Computing (TRANSACT), 2013.

Tim Harris, James Larus, Ravi Rajwar. *Transactional Memory, 2nd edition*, in Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2010.

Resources from the Transactional Memory Specification Drafting Group predating SG5 are available from `https://sites.google.com/site/tmforcplusplus/`.

10   Examples

The first example below illustrates how transactions can elegantly solve a generic programming problem that is not possible to solve with locks. Subsequent examples are intended to clarify the features specified in this proposal.

10.1   Example illustrating importance of transactions for generic programming

Below we show an attempt to use locks for generic programming, and explain a fundamental problem with it. After that, we show how the same problem can be elegantly solved using transactions. These examples are based on examples in *Generic Programming Needs Transactional Memory* by Justin Gottschlich and Hans Boehm (TRANSACT 2013).

```cpp
template <typename T>
class concurrent_sack
{
public:
   ...
   void set(T const &obj) {
      lock_guard<mutex> _(m_);
      item_ = obj;
   }
   T const & get() const {
      lock_guard<mutex> _(m_);
      return item_;
   }
private:
   T item_;
   mutex m_;
};


class T {
public:
   ...
   T& operator=(T const &rhs) {
      if (!check_invariants(rhs))
      { L.add("T invariant error"); }
   }
   bool check_invariants(T const& rhs)
   { return /* type-specific check */; }
   string to_str() const { return "..."; }
};
```

```cpp
class log {
public:
   ...
   void add(string const &s) {
      lock_guard<recursive_mutex> _(m_);
      l_ += s;
   }
   void lock() { m_.lock(); }
   void unlock() { m_.unlock(); }
private:
   recursive_mutex m_;
   string l_;
} L;
```

Given the declarations above, the following program results in deadlock. There is no way to order the locks to avoid this.

```cpp
// Globally define sack
concurrent_sack<T> sack;

Thread 1                              Thread 2
--------                              --------
                                      // acquires L.m_
                                      lock_guard<log> _(L);

// acquires sack::m_
sack.set(T());
                                      // tries to acquire sack::m_
                                      // (deadlock)
                                      L.add(sack.get().to_str());
                                      L.add("...");

// tries to acquire L.m_ (deadlock)
```

```
// if T::operator==()'s call to
// check_invariants() returns false
```

Next we revisit the same problem using transactions.

```
template <typename T>
class concurrent_sack
{
public:
   ...
   void set(T const &obj) {
      atomic_cancel { item_ = obj; }
   }
   T const & get() const {
      atomic_cancel { return item_; }
   }
private:
   T item_;
};


class log {
public:
   ...
   void add(string const &s) {
      atomic_cancel { l_ += s; }
   }
private:
   string l_;
} L;

class T {
public:
   ...
   T& operator=(T const &rhs) {
      if (!check_invariants(rhs))
      { L.add("invariant error"); }
   }
   bool check_invariants(T const& rhs)
   { return /* type-specific check */; }
   string to_str() const { return "..."; }
};
```

With these declarations, the problem can be solved as follows. Note that the order in which the transactions are invoked does not matter, because no named locks are involved that could be misordered leading to deadlock as shown in the prior example.

Instead, transactions are used for this generic programming example enabling the generic programmer to build the system the way he or she believes it should be built, without leaking the implementation details to the end programmer.

Likewise, the end programmer can program in the most natural fashion for him or her without worrying about violating some embedded locking order within the generic programming code that he or she is using.

```
// Globally define sack
```

```
concurrent_sack<T> sack;

Thread 1                            Thread 2
--------                            --------
                                    // begins local transaction
                                    atomic_cancel
                                    {
// begins sack transaction
sack.set(T());

// begins L transaction if
// T::operator=()'s  call to
// check_invariants()
// returns false

                                        // begins sack transaction,
                                        // then L transaction
                                        L.add(sack.get().to_str());
                                        L.add("...");
                                    }
```

10.2   Example demonstrating atomicity of atomic blocks

This simple bank account example demonstrates the atomicity of atomic blocks.

```
class Account {
    int bal;
  public:
    Account(int initbal) { bal = initbal; };

    void deposit(int x) {
      atomic_noexcept {
        this.bal += x;
      }
    };

    void withdraw(int x) {
      deposit(-x);
    };

    int balance() { return bal; }
}

void transfer(Account a1, a2; int x;) {
  atomic_noexcept {
    a1.withdraw(x);
    a2.deposit(x);
  }
};

Account a1(0), a2(100);

Thread 1                    Thread 2
--------                    --------
```

```
transfer(a1, a2, 50);        atomic_noexcept {
                               r1 = a1.balance() + a2.balance();
                             }
                             assert(r1 == 100);
```

The assert cannot fire, because the transfer happens atomically and the two calls to `balance` happen atomically.

10.3 Example demonstrating need for atomic_cancel

Here, we extend the above example slightly so that transactions are logged by a function that may throw an exception, for example due to allocation failure.

```
void deposit(int x) {
  atomic_cancel {
    log_deposit(x);          // might throw
    this.bal += x;
  }
}

void withdraw(int x) {
  deposit(-x);
}

void transfer(account a1, a2; int x;) {
  try {
    atomic_cancel {
      a1.withdraw(x);
      a2.deposit(x);
    } catch (...) {
      printf("Transfer failed");
    }
  }
}
```

If the call from `transfer()` to `a2.deposit()` throws an exception, we should not simply commit the transaction, because the withdrawal has happened but the deposit has not. Canceling the transaction provides an easy way to recover to a good state, without violating the invariant the transaction in `transfer()` is intended to preserve. In this simple example, an error message is printed indicating that the transfer did not happen.

10.4 Example illustrating limitation regarding types of exceptions that can escape and workaround

If `log_deposit()` might throw an exception that is not transaction-safe, programmers can work around this by translating the exception to one that is transaction-safe before allowing it to escape.

```
void deposit(int x) {
  atomic_cancel {
    try {
      log_deposit(x);                // might throw
    } catch (Exception e) {
      throw TamedException(e);   // Produces transaction-safe exception
```

```
                                    // based on original exception
    }
    this.bal += x;
  }
}
```

10.5 Example illustrating that partial effects of cancelled transactions cannot be observed

```
#define TOO_BIG 17

int X = 0;

void do_something(int x) {
  atomic_cancel {
    X = x;
    if (x > 5)
      throw TOO_BIG;
  }
}


Thread 1                   Thread 2
--------                   --------

do_something(random());    atomic_noexcept {
                             r1 = X;
                           }
                           assert(r1 <= 5);
```

The assert cannot fire because a transaction that writes a value greater than 5 is canceled and therefore its effects cannot be observed by other threads.

10.6 Examples illustrating that partial effects of transactions that cause std::abort to be called cannot be observed

The first example shows that partial effects of a `atomic_noexcept` transaction that throws an exception cannot be observed by other threads.

```
int X = 0;

Thread 1                             Thread 2
--------                             --------

atomic_noexcept {                    int x;
  X = 1;                             atomic_noexcept {
  throw 0;                             x = X;
}                                    }
                                     assert(X==0);
```

For another example, suppose Thread 1 instead executes:

```
atomic_cancel {
  X = 1;
  throw SomeFancyException();
}
```

Again, Thread 2's assert cannot fire because partial effects of the `atomic_cancel` transaction that throws a non-transaction-safe exception cannot be observed by other threads.

10.7   Examples illustrating sychronized blocks and non-races between accesses within transactions (including synchronized blocks)

Suppose we add the following method to the `Account` class shown in Section 10.2.

```
void print_balances_and_total (account a1, a2) {
  synchronized {
    printf("First account balance: %ld", a1.balance());
    printf("Second account balance: %ld", a2.balance());
    printf("Total: %ld", a1.balance() + a2.balance());
  }
}
```

Observations:

- This program is data-race-free: all concurrent accesses are within transactions.

- The synchronized block cannot be replaced with an atomic block, as I/O is not transaction-safe (due to calls to `printf`, which is a transaction-unsafe function).

- Balances will be consistent and total will equal sum of balances displayed.

- If we eliminate the synchronized block from this example (so the calls to `balance()` in `print_balances_and_total()` are not in transactions), then this program is racy.

10.8   Examples illustrating use of transaction_safe

A simple example explicitly declaring a function to be transaction-safe at its definition. This example is correct only if there is no previous declaration of `deposit`, or if the first such declaration is also explicitly transaction-safe.

```
void deposit(int x) transaction_safe {  // OK, deposit is transaction-safe
  atomic noexcept {
    this.bal += x;
  }
}
```

If a function is explicitly declared transaction-safe, this must (also) be included on the first declaration:

```
void foo();

void foo() transaction_safe {    // ERROR: must be on first declaration
  x++;                           //        if included at all
}
```

The next example illustrates that syncrhonized blocks are still allowed to be transaction-safe. Note that this example passes compilation and it is different from N3718, the first version of this proposal that was discussed in Chicago in September, 2013.

```
void foo() transaction_safe {
  ...
  synchronized {   // Passes in the current proposal; different then N3718
    ...
  }
}
```

10.9   Examples illustrating use of transaction_unsafe

A function declared transaction-unsafe cannot subsequently be declared transaction-safe.

```
[[transaction_unsafe]] void foo();

void foo() transaction_safe {   // ERROR: inconsistent declarations
   ...
}
```

A function declared transaction-unsafe cannot be called in an atomic block.

```
[[transaction_unsafe]] void foo();

void bar() {
  atomic_noexcept {
    foo();   // ERROR: foo is explicitly transaction_unsafe
  }
}
```

10.10   Examples illustrating "safe by default" (no error)

Functions (such as `foo()` in the following example) that do not include any transaction-unsafe code and do not call any functions that are not transaction-safe are implicitly transaction-safe. Furthermore, a function called within an atomic block (or an explicitly transaction-safe function) is assumed to be transaction-safe. Therefore, the following example compiles and links successfully.

```
common.h
--------

void foo();

file1.cxx
---------

#include "common.h"

void bar() {
  atomic_noexcept {
    foo();
  }
}
```

```
file2.cxx
---------

#include "common.h"

void foo() {
  // only transaction-safe stuff here
}
```

If `common.h` instead contained

```
void foo() transaction_safe;
```

the example would still compile and link successfully.

10.11 Example illustrating "safe by default" (error)

This example is similar to the one above, but the definition of `foo` contains transaction-unsafe code. As before, both files compile successfully. However, they do not link successfully, because compilation of `file1.cxx` assumed `foo()` to be transaction-safe, but its definition in `file2.cxx` is not.

```
common.h
--------

void foo();

file1.cxx
---------

#include "common.h"

void bar() {
  atomic_noexcept {
    foo();
  }
}

file2.cxx
---------

#include "common.h"

void foo() {
  printf("unsafe");              // transaction-unsafe due to I/O
}
```

If `common.h` instead included:

```
void foo() transaction_safe;
```

then compilation of `file2.cxx` would fail because `foo` contains something transaction-unsafe (I/O).
If `common.h` instead included:

```
[[transaction_unsafe]] void foo();
```

then compilation of `file1.cxx` would fail because `foo` is declared transaction-unsafe.

10.12   Examples illustrating transaction-safe function pointers

The following example illustrates combinations of transaction-safe and transaction-unsafe function and function pointers.

```
void (*fp)();
void (*tsfp)() transaction_safe;

void safefunc() {
  // nothing that is transaction-unsafe
}

void unsafefunc() {
  printf("Hello");
}

void bar() {
  fp = &unsafefunc;     // OK
  fp = &safefunc;       // OK
  tsfp = &unsafefunc;   // ERROR: can't assign transaction-unsafe function
                        //        to transaction-safe function pointer
  tsfp = &safefunc;     // OK
  fp = tsfp;            // OK: implicit conversion
  tsfp = fp;            // ERROR: can't assign transaction-unsafe function
                        //        pointer to transaction-safe one
  (*fp)();              // OK
  (*tsfp)();            // OK
  atomic noexcept {
    (*tsfp)();          // OK
    (*fp)();            // ERROR: call through fp transaction-unsafe
                        //        because fp is not transaction-safe
  }
}
```

10.13   Example illustrating function-local static initialization in atomic blocks

Consider this example:

```
std::pair<int,int>  f(int i) {
      static int x = i;
      static int y = i;
      return std::pair(x,y);
}
```

```
Thread 0                                 Thread 1
--------                                 --------

atomic_noexcept {                        f(1);
    auto r1 = f(0);
}
```

Because the transaction is an atomic block, it is not possible for `r1` to get (0,1): if Thread 0 initializes `x` to 0, then `f(1)`'s attempt to initialize `x` comes after Thread 0's transaction has completed, so Thread 1 does not initialize `y` before Thread 0 does (but see Section 6).

In contrast, of Thread 0 instead used a synchronized block, it *would* be possible for `r1` to get (0,1). This is because Thread 1's call to `f(1)` could start after Thread 0's synchronized block has initialized `x` but has not yet initialized `y`. There is no synchronization in the program to prevent this possibility, whereas the requirement for atomic blocks to be atomic does. (We note that concurrent, unsynchronized initialization of the same function-local static variable is explicitly not racy; see C++ standard section 6.7 paragraph 4.)

10.14 Examples illustrating virtual functions and overriding

A member function declared with a `transaction_safe` keyword or `transaction_unsafe` attribute in a base class preserves that attribute in any derived class, unless that member is redefined or overridden. Functions brought into a class via a `using` declaration preserve the attribute in the original scope. A virtual function of transaction-safe type must not be overridden by a virtual function of transaction-unsafe type.

```
struct B {
  virtual int f() {
    printf("not safe");  // not transaction-safe
    return 0;
  }

  virtual int g() transaction_safe {
    return 1;              // OK, transaction-safe definition
  }

  virtual int h() {
    return 1;              // not transaction-safe; must be explicit on
  }                        // virtual functions
};

struct D : B {
  virtual int f() transaction_safe { // OK, transaction-safe override
                                     // of transaction-unsafe virtual function
    return 5;                        // OK, transaction-safe definition
  }

  virtual int g() {        // implicitly declared transaction-safe
    printf("not safe");  // ERROR: call to transaction-unsafe function
    return 0;
  }

  virtual int h() transaction_safe {   // ERROR: overridden virtual
                                       // function not explicitly
                                       // transaction-safe
    return 5;
  }
};
```

The following example demonstrates that a function inherited from a base class remains transaction-safe; the same is true for transaction-unsafe.

```
struct B {
```

```
  void f() transaction_safe;
};

struct D : B {
  // when naming B::f through D, B::f stays transaction_safe
};

void g() {
  atomic {
    D d;
    d->f();   // ok, call to transaction-safe function
  }
}
```

11 Acknowledgement