# Transactional Memory Support for C++

Authors: Victor Luchangco, victor.luchangco@oracle.com
Jens Maurer, jens.maurer@gmx.net
Mark Moir, mark.moir@oracle.com
*with other members of the transactional memory study group (SG5), including:*
Hans Boehm, hans.boehm@hp.com
Justin Gottschlich, justin.gottschlich@intel.com
Maged Michael, magedm@us.ibm.com
Torvald Riegel, triegel@redhat.com
Michael Scott, scott@cs.rochester.edu
Tatiana Shpeisman, tatiana.shpeisman@intel.com
Michael Spear, spear@cse.lehigh.edu
Michael Wong, michaelw@ca.ibm.com
Document number: N3718
Date: 2013-08-30
Project: Programming Language C++, Evolution 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. It 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. This proposal 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 the SG5's efforts towards a basic set of features that is useful and can support progress towards possible inclusion in the C++ standard. Reflecting this goal, for the first time, we present precise wording changes relative to the Working Draft of the C++ Standard (N3690) to implement this proposal. This document consists of an informal overview of the proposal, several illustrative examples, a summary of some of the discussion in SG5 and the earlier drafting group about design choices, and the above-mentioned wording changes.

## 2 Overview

Sections of code may be designated as *atomic transactions*, which appear to execute atomically. Thus, when reasoning about the behavior of their programs, programmers need not consider executions in which operations by other threads are interleaved with the operations of an atomic transaction. Some operations are prohibited within atomic transactions because it is impossible, difficult, or expensive to support executing them in atomic transactions; such operations are called *transaction-unsafe*.

Distinct from atomic transactions, *relaxed transactions* behave as if all relaxed transactions were protected by a single special mutex, and no atomic transaction appears to take effect while this mutex is held. In contrast to atomic transactions, there are no restrictions on what code can be executed within relaxed transactions. However, there is no guarantee that relaxed transactions appear to execute atomically, and

implementations must serialize relaxed transactions in at least some cases, thus reducing parallelism and scalability. The relationship between atomic transactions and relaxed transactions is discussed in section 7.1.

A transaction is expressed as a compound statement (its *body*) with a special keyword prefix. An atomic transaction statement also specifies how to handle exceptions that escape the transaction.

**Data races** Operations executed within transactions (atomic or relaxed) do not form a data race with each other. However, they may form a data race with operations outside transactions. As usual, programs with data races have undefined semantics. (See proposed wording changes for **1.10 [intro.multithread]**.)

**Exceptions** When an exception *escapes* an atomic transaction (i.e., it is thrown but not caught by the transaction), the effects of operations executed by the transaction may take effect or be discarded, or the transaction may call `std::abort` . This behavior is specified by an additional keyword in the atomic transaction statement, as described in section 3. An atomic transaction whose effects are discarded due to an escaping exception is said to be *canceled*. An atomic transaction that completes without its effects being discarded, and without calling `std::abort` , is said to be *committed*. (See proposed wording changes for **15.2 [except.ctor]**.)

**Transaction-safety** As mentioned above, transaction-unsafe operations cannot be executed in an atomic transaction. This restriction applies not only to code in the body of an atomic transaction, but also to code in the body of functions called (directly or indirectly) within the atomic transaction. To support static checking of this restriction, a keyword is provided to declare that a function or function pointer is transaction-safe, and the type of a function or function pointer is augmented to specify whether it is transaction-safe.

To reduce the burden of declaring functions transaction-safe, a function is *implicitly* declared 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 is checked at link time. This assumption does *not* apply to virtual functions, or to functions accessed through function pointers, as the callee is not generally known statically to the caller. (See proposed wording changes for **8.4.4 [dcl.fct.def.tx]**.)

# 3 Atomic Transactions

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

`transaction_atomic noexcept` { *body* }
`transaction_atomic commit_on_escape` { *body* }
`transaction_atomic cancel_on_escape` { *body* }

(See proposed wording changes for **6.x [stmt.tx]**.) The keyword following `transaction_atomic` is called its *transaction exception specifier*. It specifies the behavior when an exception escapes the transaction:

`noexcept`: `std::abort` is called; no side effects of the transaction can be observed.

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

`cancel_on_escape`: If the exception is transaction-safe (defined below), the transaction is canceled and the exception is thrown. Otherwise, `std::abort` is called. 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. (See proposed wording changes for **clause 18**.)

**Comment:** *The set of transaction-safe exceptions was deliberately chosen to be small initially, and may be expanded in the future.*

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

- it is a relaxed transaction statement;
- it contains an initialization of, assignment to, or a read from a volatile object;
- it is an unsafe asm declaration (the definition of an unsafe asm declaration is implementation-defined); or
- it contains a function call to a transaction-unsafe function, or through a function pointer that is not transaction-safe (see section 5).

(See proposed wording changes for **8.4.4 [dcl.fct.def.tx]**.)

**Note:** *Atomic transactions may be nested.*

**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 7.14.*

**Comment:** *The proposed wording changes do not yet guarantee that this initialization will be transaction-safe, and so need to be amended.*

**Note:** *Synchronization via locks and atomic objects is not allowed within transactions (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 transaction using `goto` or `switch` is prohibited. (See proposed wording changes for **clause 6**.)

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

A memory access within an atomic transaction does not race with any other memory access in an atomic transaction or a relaxed transaction. However, a memory access within an atomic transaction does race with conflicting memory accesses outside any transaction, unless these accesses are synchronized using some other mechanism. The exact rules for defining data races are defined by the memory model. (See proposed wording changes for **1.10 [intro.multithread]**.)

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

**Note:** *Although it has no observable effects, a canceled transaction still participates in data races.*

# 4 Relaxed Transactions

A *relaxed transaction statement* has the following form:

`transaction_relaxed` { *body* }

A relaxed transaction behaves as if a special mutex (one for the entire system) is acquired before executing the body and released after the body is executed (unless the relaxed transaction statement is nested within another relaxed transaction, in which case the mutex is not released until the end of the outermost relaxed transaction), and no atomic transaction appears to take effect while this special mutex is held by any other thread. (See proposed wording changes for **6.x [stmt.tx]**.)

**Note:** *Any code may be executed within a relaxed transaction.*

Jumping into the body of a relaxed transaction using `goto` or `switch` is prohibited.

# 5 Transaction-Safety for Functions

A function declaration may specify a `transaction_safe` keyword or a `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). (See proposed wording changes for **8.3.5 [dcl.fct]**.)

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 3.

**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. (See proposed wording changes for **4.14 [conv.tx]** and **clause 13**.)

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.

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. (See proposed wording changes for **10.3 [class.virtual]**.)

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 the proposed wording for details.

**Comment:** *The set of transaction-safe standard library functions was deliberately chosen to be small initially, and may be expanded in the future.*

# 6 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.

## 6.1 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).

```
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 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;
```

```
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 "..."; }
};
```

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

```
// 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) {
      transaction_atomic cancel_on_escape { item_ = obj; }
   }
   T const & get() const {
      transaction_atomic cancel_on_escape { return item_; }
   }
private:
   T item_;
};


class log {
public:
   ...
   void add(string const &s) {
      transaction_atomic cancel_on_escape { 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
                                  transaction_atomic cancel_on_escape
                                  {
// 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("...");
                                  }
```

## 6.2 Example demonstrating atomicity of atomic transactions

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

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

    void deposit(int x) {
      transaction_atomic noexcept {
        this.bal += x;
      }
```

```
    };

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

    int balance() { return bal; }
}

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

Account a1(0), a2(100);

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

transfer(a1, a2, 50);     transaction_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.

### 6.3 Example demonstrating need for cancel-on-escape

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) {
  transaction_atomic cancel_on_escape {
    log_deposit(x);        // might throw
    this.bal += x;
  }
}

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

void transfer(account a1, a2; int x;) {
  try {
    transaction_atomic cancel_on_escape {
      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.

### 6.4 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) {
  transaction_atomic cancel_on_escape {
    try {
      log_deposit(x);            // might throw
    } catch (Exception e) {
      throw TamedException(e);   // Produces transaction-safe exception
                                 // based on original exception
    }
    this.bal += x;
  }
}
```

### 6.5 Example illustrating that partial effects of cancelled transactions cannot be observed

```
#define TOO_BIG 17

int X = 0;

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

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

do_something(random());     transaction_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.

### 6.6 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 `noexcept` transaction that throws an exception cannot be observed by other threads.

```
int X = 0;

Thread 1
--------

transaction_atomic noexcept {       Thread 2
  X = 1;                            --------
  throw 0;
}                                   int x;
                                    transaction_atomic noexcept {
                                      x = X;
                                    }
                                    assert(X==0);
```

For another example, suppose Thread 1 instead executes:

```
transaction_atomic cancel_on_escape {
  X = 1;
  throw SomeFancyException();
}
```

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

### 6.7 Examples illustrating relaxed transactions and non-races between accesses within transactions (including relaxed)

Suppose we add the following method to the `Account` class shown in section 6.2.

```
void print_balances_and_total (account a1, a2) {
  transaction_relaxed {
    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 relaxed transaction cannot be replaced with an atomic transaction, 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 relaxed transaction from this example (so the calls to `balance()` in `print_balances_and_total()` are not in transactions), then this program is racy.

### 6.8 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
  transaction_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 relaxed transactions are not transaction-safe. Note that this example fails compilation even if there are no calls to `foo` within transactions.

```
void foo() transaction_safe {
  ...
  transaction_relaxed {   // ERROR: not transaction-safe
    ...
  }
}
```

### 6.9 Examples illustrating use of transaction_unsafe

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

```
void foo() transaction_unsafe;

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

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

```
void foo() transaction_unsafe;

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

### 6.10 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 transaction (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() {
  transaction_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.

### 6.11 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() {
  transaction_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:

```
void foo() transaction_unsafe;
```

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

### 6.12 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
  transaction_atomic noexcept {
    (*tsfp)();          // OK
    (*fp)();            // ERROR: call through fp transaction-unsafe
                        //        because fp is not transaction-safe
  }
}
```

### 6.13 Example illustrating function-local static initialization in atomic transactions

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
--------                                --------

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

Because the transaction is atomic, 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 7.14).

In contrast, of Thread 0 instead used a relaxed transaction, 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 relaxed transaction 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 transactions 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.)

### 6.14 Example illustrating lambdas in transactions

Lambdas that contain only transaction-safe code are transaction-safe.

```
transaction_atomic noexcept {
  [](int x){ return x*x; }(1);
}
```

### 6.15 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() {
  transaction_atomic {
    D d;
    d->f();   // ok, call to transaction-safe function
  }
}
```

# 7 Discussion of Various Issues

The proposal described in this document was developed by SG5, 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. We have deliberately scaled back the proposal in the *Draft Specification*, omitting or simplifying many of the more controversial or complicated features. This proposal represents a stage where the feature set is believed to be internally consistent and useful to programmers. Evolution of the feature set is expected to continue after the initial publication of the Technical Specification.

In this section, we summarize some of the discussion about various features (including several features in the *Draft Specification* that were omitted in this proposal), which we hope will help programmers and implementors understand the design choices made, and some of the alternatives.

## 7.1 Atomic transactions vs. relaxed transactions

Feedback on the *Draft Specification* revealed confusion over the differences between atomic transactions and relaxed transactions, the need for both, etc. We have had extensive discussions about the purpose and relative merits of atomic and relaxed transactions, their relation to each other, whether we should support both kinds of transactions, and if so, whether one ought to be considered "primary".

Some argued, for example, that the primary purpose of transactions is to provide a simpler form of critical sections, avoiding the need for complicated and error-prone locking policies, and enabling parallel execution through speculation, and that relaxed transactions provide these benefits with little semantic complication. In this view, atomic transactions are a restricted form of transaction for which we can statically guarantee atomicity.

Others argued that the guarantee of atomicity is the fundamental property of transactions that provides their primary benefit: eliminating the need to reason about executions in which the operations of a transaction are interleaved with operations of other threads. (Some even argued that relaxed transactions should not be called "transactions" since they do not guarantee atomicity.) In this view, relaxed transactions are a mechanism for managing the interaction between transactional and nontransactional code, and a concession to the fact that not all code can be executed atomically.

Much of the discussion was clouded with confusion about the exact semantics of relaxed transactions, and a significant breakthrough came in the realization that we could specify relaxed transactions in terms of a special global mutex that is also "respected" by atomic transactions (i.e., atomic transactions do not appear to execute while that mutex is owned).

Although differing views remain on the relative merits of atomic transactions and relaxed transactions, the discussion did lead to fruitful insights and clarity on the semantics of and differences between them, enabling us to converge on the proposal described in this document. We also agreed to defer discussion of whether relaxed and/or atomic transactions should be called something else.

Due to concern that the need to serialize relaxed transactions in many cases will be problematic for performance and scalability, we discussed more advanced proposals that deliver similar benefits, but allow more fine-grained synchronization. We have also deferred this discussion, as the predicted problem has not yet been demonstrated, and we believe that the proposed features are useful without supporting finer-grained synchronization between transactional and nontransactional code.

One point deserves further discussion: As noted in the *Draft Specification*, the behavior of a relaxed transaction that executes only transaction-safe code is currently indistinguishable from that of an atomic transaction executing the same code. That is, a relaxed transaction appears to execute atomically if it executes only transaction-safe code. However, if we extend transaction-safety to include some forms of synchronization, such as access to atomic objects or locks, this property may not hold in the future.

We have recently observed that initialization of function-local static variables also breaks this putative equivalence. Thus, if such code is to be considered transaction-safe, then the semantics of atomic transactions and relaxed transactions will no longer be equivalent, even for code that is entirely transaction-safe. This point is elaborated in section 7.8.

## 7.2 Transaction cancellation

The *Draft Specification* included a *cancel statement*, which could be executed within an atomic transaction to cancel that transaction. This feature was the subject of much discussion, about both the exact semantics of this feature and its utility.

The semantics of transaction cancellation is complicated by three issues:

1. What information can escape a canceled transaction? At minimum, we can (and want to be able to) detect that the transaction was canceled. But some argued that it would be helpful to be able to return more information. This issue is particularly acute in the case of exceptions, as discussed in section 7.3.

2. When a cancel statement is executed in a nested atomic transaction, which transaction is canceled? Some argued that only the innermost transaction ought to be canceled (supporting a semantics called *closed nesting*). Others thought the outermost transaction should be canceled, to allow implementations that support only *flat nesting* (which is simpler to implement than closed nesting). The *Draft Specification* included both "inner" and "outer" variants.

   During these discussions, a way to generalize and unify these apparently-different forms of cancellation was suggested: by allowing threads to determine a their current nesting depth and to specify a number of levels to be canceled, both inner and outer forms of cancellation be supported by the same mechanism, as well as intermediate forms that support cancelling any number of nested transactions. However, as summarized below, for the current proposal we have agreed to restrict cancellation only to the case that it is unambiguously necessary (see the example in section 6.3).

3. A cancel statement may be executed only within the context of an atomic transaction. To enable static checking of this requirement, the *Draft Specification* restricted "inner" cancel statements to occur within the lexical scope of an atomic transaction, and augmented the types of functions by introducing a new function attribute (`transaction_may_cancel_outer`) to characterize functions that might execute an "outer" cancel statement. Some wanted to relax the restriction on "inner" cancel statements (e.g., by further augmenting the types of functions). Some thought the effect on types was too complicated, especially in its interaction with transaction-safety.

As for the utility of transaction cancellation, proponents noted that it enabled "speculative programming", in which a transaction is used to do some computation that is valid only if the computation results in some condition being true. If that condition is not true, then the transaction can be canceled with no effect. It also may enable simpler ways to provide exception safety. Others argued that there were already ways to provide exception safety, and the promise of speculative programming was too speculative to justify the complexity introduced by cancellation.

In the end, we decided to table this issue, and omit the cancel statement from this proposal. However, we retained a limited form of cancellation for when exceptions escape from atomic transactions.

## 7.3 Handling exceptions escaping from atomic transactions

There was much discussion about the desired semantics for an exception escaping an atomic transaction. Some people argued that the transaction should be canceled, that committing the transaction would violate the programmer's expectation by executing only part of the intended transaction, and that canceling the transaction would be an elegant way to provide exception safety. Others argued that the transaction should commit and throw the exception, to provide "single global lock" semantics, and to ensure that the semantics of single-threaded code is not changed by putting code within transactions, and that some code is already exception-safe, and it would not make sense to cancel when an exception is thrown from such code.

We agreed that there were some examples for which it would be "natural" to commit, and others for which it would be "natural" to cancel. We also recognized that under the "cancel semantics", it is possible to mimic the "commit semantics" by putting the body of the atomic transaction within a try block, and that with a cancel statement (or some alternative mechanism for canceling atomic transactions), it would be similarly possible to mimic the semantics in the other direction. Nonetheless, we did not agree about which semantics ought to be the default. Therefore, we agreed to have no default: every atomic transaction must explicitly specify how to handle escaping exceptions. (Alternatively, an atomic transaction may specify that, instead of committing or canceling, `std::abort` is called whenever an exception escapes the transaction.)

As mentioned in section 7.2, canceling transactions introduces semantic complications. In particular, because the effects of the transaction are discarded when the transaction is canceled, the condition that caused the exception may not hold after cancellation, which may be a problem for the code handling the exception.

An issue of special concern is that the exception might contain references to objects allocated by the transaction. If all the effects of a transaction are discarded, then this includes the allocation of these objects, so the references to them are invalid. (Indeed, the exception itself may have been allocated by the transaction, but we agreed that this effect should not be discarded, which is consistent with the special treatment of memory allocated for exception objects in C++.) To avoid this issue, we restrict the exceptions that can cancel an atomic transaction (i.e., the transaction-safe exceptions) to exceptions with scalar types, and a few other exceptions such as `bad_alloc` that we think important to handle. We have deliberately kept this set small for now, but may expand it in the future.

This issue is discussed in greater detail in the paper *Exceptions and Transactions in C++* published in HotPar 2009, though there has been more discussion after its publication.

Since atomic transactions may be nested as noted before, if a suitable exception is thrown from an inner cancel_on_escape transaction, only the inner transaction is immediately aborted, making it possible for the outer transaction to handle the exception.

## 7.4 Transaction-safety for functions

In the *Draft Specification*, a function is transaction-safe only if it is explicitly declared transaction-safe or its definition is available and its body contains no transaction-unsafe code (including calls to functions not known to be transaction-safe). The latter condition was intended to reduce the burden of declaring functions transaction-safe, without forgoing compile-time checking. Nonetheless, many people thought that this burden remained too high (we received feedback to that effect at previous meetings, for example). Thus, we adopted the current proposal, in which functions are assumed to be transaction-safe by default, with this assumption being confirmed only at link time in some cases.

A simple alternative would be to restrict function calls within atomic transactions to functions whose definition is available earlier in the translation unit (e.g., inline functions). Calling functions through function pointers or using virtual functions would have to be prohibited entirely, since the callee is not generally known to the caller. We deemed this alternative impractical, because it would prevent calling non-inline functions in libraries.

At the other extreme, we could give up on static checking entirely, either by introducing a run-time check that a function is transaction-safe when it is called within an atomic transaction, or by declaring it the programmer's responsibility to ensure that functions called within atomic transactions are transaction-safe. We deemed forgoing static checking to be dangerous (especially with the latter variant), and likely to lead to programs with subtle and difficult-to-diagnose concurrency bugs. We also worried about the performance impact of introducing a run-time check in the first variant.

**Implementation considerations** We expect that typical implementations of this proposal will be able to execute transactions in parallel using *speculation*, under the assumption that actual conflicts between concurrent transactions are rare. In such implementations, if a conflicting access does happen, typically all effects of that section of code are undone and the entire section is retried. The definition of transaction-unsafe code is motivated in part to capture code that performs operations that cannot be undone. This implementation approach requires the compiler to instrument code executed within an atomic transaction (unless hardware support for undoing the effects of code is available).

We also expect that typical implementations will generate both instrumented and uninstrumented variants of each transaction-safe function, with the uninstrumented variant used outside transactions or with hardware support for transactions, and the instrumented variant used for software-based transactions. Since these variants must have different mangled names so that the appropriate variant can be called depending on context, that difference can be exploited to avoid explicit annotations when a named function is called: If a function called within an atomic transaction turns out to be transaction-unsafe, the variant for use within atomic transactions will not be available, and thus a program requiring that variant will fail to link.

**Practical concerns: code bloat and compilation time** The "safe by default" approach has raised concerns that treating functions as transaction-safe even though they may not be used in transactions could lead to unnecessary code bloat (because instrumented versions of functions may be produced unnecessarily). While there may be some hope for this issue to be addressed by link-time optimizations for some cases, such functionality may not be supported by all implementations.

Programmers who are concerned about this issue (or who actually experience the problem) can address it by explicitly declaring some functions to be transaction-unsafe, preventing the problem. This may seem to add back some of the burden we have sought to reduce (i.e., the previous requirement to explicitly declare all functions to be used in transactions as transaction-safe). However, we note that the burden applies only to programmers concerned about this issue, and they can declare functions transaction-unsafe incrementally, and only to the extent necessary to address their problem. In contrast, the previous requirement to declare functions transaction-safe before using them in atomic transactions applied to all programmers, and these declarations needed to be consistently applied everywhere in all programs before they could even be tested for the first time.

Finally, we note that preparing transaction-safe versions of functions for which they are not needed may unnecessarily increase compliation time, and this issue cannot be resolved by link-time optimizations. Again, those who face this issue can address it by applying transaction-unsafe declarations in an incremental fashion, and without blocking ongoing development in the meantime.

## 7.5 Choice of transaction-safe standard library functions

It seems plausible to allow most of the standard library containers, strings, and iterators to be accessed within atomic transactions. However, some implementations might have a debug mode where they perform I/O in rarely executed code paths, making the implementation transaction-unsafe. For now, these are excluded from the set of functions specified to be transaction-safe.

We expect that, in the future, means for explicitly allowing code that would violate atomicity from some perspectives to be included in atomic transactions, which would remain atomic from other perspectives. For a concrete example, suppose a function that is otherwise transaction-safe includes code for logging events to support debugging. In this case, the atomicity of transactions executing this code is preserved for all observers with the possible exception of those that examine the logged information. Thus the program's behavior is not compromised, even though a person examining the output log could detect that the transactions did not execute atomically.

### 7.6 Transaction-safety for function pointers

To support calls through function pointers within atomic transactions with static checking, we also allow function pointers to be declared transaction-safe. However, we cannot assume, as we did for functions, that function pointers are transaction-safe by default, because in a call through a function pointer, the callee is not generally known statically to the caller. Thus, a call through a function pointer is transaction-unsafe unless the function pointer is explicitly declared transaction-safe. Such a function pointer must not be assigned a pointer to a transaction-unsafe function. (It may be assigned a pointer to a function that is assumed to be transaction-safe by default, because that condition can be checked at link time if necessary.)

The *Draft Specification* allowed a function pointer declaration to specify the `transaction_unsafe` attribute. We agreed that this should not be allowed because there is no semantic difference between a function pointer declared with the `transaction_unsafe` attribute and a function pointer declared with neither the `transaction_safe` keyword nor the `transaction_unsafe` attribute.

### 7.7 Transaction-safety for virtual functions

In this proposal, functions are transaction-safe "by default", with checking deferred until link time in some cases. We decided not to extend this default to virtual functions because we cannot guarantee the transaction-safety of a call to a virtual function of a base class unless we guarantee the transaction-safety of every function that overrides it in a derived class. In this, calls to virtual functions are like calls through function pointers, which are transaction-unsafe unless the function pointer is declared with the `transaction_safe` keyword. We thought that C++ programmers would be surprised if we treated calls to virtual functions and calls through function pointers differently. We also thought that the burden on programmers might not be too great, even if some classes with virtual functions are widely used, because only those classes with virtual functions that need to be called within atomic transactions need to be modified. Also, there is a trend away from deep class hierarchies, so modifying those classes might not be so painful (if it is, we hope users will correct us).

We considered and discussed some alternatives in which virtual functions are transaction-safe by default. For example, we could require transaction-unsafe virtual functions to be explicitly declared so, and reject programs in which a transaction-unsafe virtual function overrides a transaction-safe virtual function, even though neither is declared with explicit transaction-safety attributes. However, this option may require changes to legacy code (i.e., adding the `transaction_unsafe` attribute to declarations of transaction-unsafe virtual functions). Alternatively, we could have a virtual function call in an atomic transaction throw a runtime error if it results in a call to a transaction-unsafe function from a derived class. However, this option introduces runtime errors and requires additional run-time checks to detect this possibility.

We also discussed the option of having a compiler flag to indicate that virtual functions should be considered transaction-safe by default, with a runtime error being thrown in case a transaction-unsafe function is called within a transaction (i.e., the second alternative option above). Although this would not be part of the proposed specification (which defines the language, not compiler flags), a list of this and other similar options that may ease the adoption of transactions may be useful to implementors and users. (Even with this option, a virtual function explicitly declared with the `transaction_safe` keyword must not be overridden by a transaction-unsafe function.)

### 7.8 Transaction-safety of dynamic initialization of function-local statics

We agreed that dynamic initialization of function-local statics should be transaction-safe, assuming that the initialization expression is transaction-safe. We recognize that ensuring transaction-safety for dynamic initialization of function-local statics imposes some implementation overhead when the initialization is executed outside a transaction to ensure proper synchronization, but we think this overhead should apply only

to the actual initialization (not to subsequent accesses) and should therefore not be problematic. In recent discussions, however, we realized that the proposed wording changes do not reflect this decision, and also that there are some implementation questions that we need to consider further.

The discussion about differences between the semantics of atomic transactions and relaxed transactions was often clouded by the belief held by some that their semantics are identical if the transaction body does not contain any transaction-unsafe code. Initialization of function-local static variables provides a concrete demonstration that this thinking is not correct, as shown in section 6.13 and discussed further in section 7.14.

### 7.9 The `transaction_safe` class attribute

The *Draft Specification* allowed a class to be declared `transaction_safe`, as a shorthand for declaring each of its members `transaction_safe`. This was intended to reduce the annotation burden required by that specification's rules for transaction-safety declarations. However, this issue is mitigated in the current proposal (which interprets functions as "safe by default"), so we agreed to omit these for now and see whether there is any clamor for such syntactic sugar.

### 7.10 Transaction expressions

The *Draft Specification* included transaction expressions:

> Replacing the block in braces with a parenthesized expression in either atomic or relaxed transaction statements yields a transaction expression of type T, where T is the type of the parenthesized expression. . . . It is equivalent to the corresponding transaction statement with a single assignment statement to a newly generated variable of type T with the expression as the right-hand side, and then evaluating the variable (outside the transaction) to get the value of the transaction expression.

We agreed to remove transaction expressions from this proposal because their functionality can be mimicked using lambdas that are invoked immediately. We thought it better to first teach this admittedly relatively clunky pattern, and then consider adding transaction expressions back if there is sufficient user demand.

### 7.11 Function transaction blocks

The *Draft Specification* also provided function transaction blocks by inserting `transaction_atomic` or `transaction_relaxed` after the function signature, which would cause the body of the function to be executed as a transaction. In the case of constructors, this includes all member and base class initializers. This feature was intended to support transactional construction of objects. However, the actual initialization in `C c = C()` calls the copy constructor in addition to the default constructor. The execution of the copy constructor would not be covered by a transaction inside the default constructor. Additionally, the extent of such a function transaction block might be too broad when just the atomicity of initialization for several members or base classes is desired. Therefore, we decided not to include the feature until more user experience with transactions in general is available.

### 7.12 The `transaction_callable` attribute

The *Draft Specification* included a `transaction_callable` attribute, which was a hint for the compiler that a function might be called within a relaxed transaction (it had no semantic implication). It was added to address concerns expressed about possible code bloat or poor performance due to lack of instrumentation. We decided to omit it and see whether there is indeed a problem.

## 7.13 Empty transactions

As specified, an empty transaction (i.e., a transaction whose body does not execute any code) is effectively a fence (because it still has a start and an end that synchronizes-with the starts and ends of other transactions). Several people expressed concern that we may want to allow an implementation to simply elide such a transaction, which is not possible under this interpretation; this concern is amplified by the fact that a future change to this would be a relaxation rather than a strengthening of the specification (i.e., it could break existing code). However, we did not think we would be able to agree on and precisely specify an appropriate relaxation in time for this proposal, so we decided to keep the specification as is.

## 7.14 Outstanding issues

Finally, we mention some issues that we have yet to resolve, having only realized them recently.

**Making atomicity "first class" in memory model** The proposed wording changes for the memory model (**1.10 [intro.multithread]**) assume that it is sufficient to ensure that the end of one transaction synchronizes-with the start of the next transaction (in some total order over transactions). This is based on the assumption that the current restrictions on what code can be executed in an atomic transaction imply that any concurrent code that could observe a violation of the atomicity of an atomic transaction would necessarily form a data race with that transaction.

However, we have realized that this assumption is invalid. As discussed in section 7.8, because concurrent initialization of function-local static variables are not racy, the wording changes for the memory model must more directly require atomicity of atomic transactions. We are considering a wording change along the following lines:

> If the start of an atomic transaction 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.

Because we realized this issue only recently, this condition is included only in a non-normative note. We intend to consider this issue more thoroughly. We hope that making atomicity a first-class requirement will eliminate the problem with initializing function-local static variables in atomic transactions, and also facilitate allowing locks and/or atomic variables to be accessed within atomic transactions, should we decide to allow such access in the future.

**Terminating the program from within a transaction** We have specified that `std::abort` be called when an atomic transaction violates its exception assumptions (for example, a cancel-on-escape transaction throws a non-transaction-safe exception). We previously intended to specify that `std::terminate` be called in this case, but recently realized that this raises a tricky issue: what should the status of the transaction be from the perspective of any termination handler that has been specified? Because such termination handlers may not have been prepared for use within transactions, we cannot specify required behavior in an implementation-independent way.

However, we recently realized that in some cases, normal exception behavior (independent of transactions) requires `std::terminate` to be called, which raises the same question, and cannot be dodged as easily. We have not yet had time to fully address this issue.

# 8 Acknowledgement and Related Documents

**Acknowledgement** This work is the combined dedication and contribution from many people from academia, industry, and research through many years of discussions and feedback. Some of those are all the authors and chairs of the original external TM group that produced the original TM specification, all of the current SG5 SG, as well as individuals such as Dave Abraham, Zhihao Yuan, Paul Mckenney, Lawrence Crowl, Detlef Vollmann, Ville Voutilainen, Nevin Liber, Bjarne Stroustrup, Herb Sutter, Bronek Kozicki, Tony Van Eerd, Steve Clamage, Sebastien Redl, Niall Douglas and many others whom we have forgotten inadverdently to list.

**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*

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

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.

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

# 9 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 atomic or relaxed transaction is a full-expression (1.9 intro.execution). A transaction block (6.x [stmt.tx]) that is not dynamically nested within another transaction block is called an outer transaction. [ Note: An outer transaction may be an atomic or a relaxed transaction. Due to syntactic constraints, a transaction block cannot overlap another one unless one is nested within the other. ] There is an unspecified global total order of execution for all outer transactions. 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**, transactions,** 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 transaction 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 transaction appear to be contiguous in the interleaving. – end note ]**

## Clause 2 [lex] Lexical conventions

In section 2.11 [lex.name] paragraph 2, add the contextual keywords `commit_on_escape` and `cancel_on_escape` to the table.

In section 2.12 [lex.key], add the keywords `transaction_atomic`, `transaction_relaxed`, `transaction_safe`, and `transaction_unsafe` 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).*

***TODO**: 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).*

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: ... ]

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
      transaction-statement
```

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

**Transfer out of a transaction block other than via an exception executes the end of the transaction. [ 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.tx]:

**6.x [stmt.tx] Transaction statement**

```
transaction-statement:
      transaction_atomic transaction-exception-specifier compound-statement
      transaction_relaxed compound-statement

transaction-exception-specifier:
      noexcept
      commit_on_escape
      cancel_on_escape
```

**A transaction statement is also called a transaction block. The *start of the transaction* is immediately before the opening `{` of the *compound-statement*. The *end of the transaction* 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 transaction; see 6.6 [stmt.jump]. — end note ]**

**A *transaction-statement* using the keyword `transaction_atomic` is called an *atomic transaction*. 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]). A *transaction-statement* using the keyword `transaction_relaxed` is called a *relaxed transaction*.**

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

*Drafting note: The syntax spot between the new transaction_atomic keyword and the opening brace of the compound-statement that follows is new, thus no ambiguities arise when using contextual keywords.*

## 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-qualifier*$_{opt}$ ***tx-safe***$_{opt}$ **transaction_unsafe**$_{opt}$ *exception-specification*$_{opt}$ *attribute-specifier-seq*$_{opt}$

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-seq*$_{opt}$
*ref-qualifier*$_{opt}$ ***tx-safe***$_{opt}$ **transaction_unsafe**$_{opt}$ *exception-specification*$_{opt}$ *attribute-specifier-seq*$_{opt}$

and the type of the contained *declarator-id* in the declaration T D1 is "derived-declarator-type-list T", the type of the *declarator-id* in D is "derived-declarator-type-list ***tx-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-seq*$_{opt}$
*ref-qualifier*$_{opt}$ ***tx-safe***$_{opt}$ **transaction_unsafe**$_{opt}$ *exception-specification*$_{opt}$ *attribute-specifier-seq*$_{opt}$ *trailing-retu*

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-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 a transaction. — 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-seq*$_{opt}$~~
~~*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,**
- **a relaxed transaction (6.x [stmt.tx]),**
- **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**
- **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.**

**[ 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).**

## 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 transactions**
>
> 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 transactions 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 transactions as-if the start of the transaction were the construction of an object with automatic storage duration**.

and add new paragraph 4 and 5:

> **A transaction is *terminated* according to the kind of transaction block. Terminating a relaxed transaction or an atomic transaction marked `commit_on_escape` has no effect. [Note: That is, control simply exits the transaction block. — end note] Terminating an atomic transaction marked `cancel_on_escape`, if the type of the current exception is not transaction-safe, or marked `noexcept` invokes `std::abort` (18.5 [support.start.term]). [ Footnote: If the effects of the transaction 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 transaction marked `cancel_on_escape`, if the type of the current exception is transaction-safe, *cancels* the transaction 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 transaction is executed. [ Note: This causes inter-thread synchronization. — end note ]**
> - **The temporary object replaces the exception object in the subsequent stack unwinding.**
>
> **[ Note: A cancelled transaction, although having no visible effect, still participates in data races (1.10 [intro.multithread]). — end note ]**
>
> **Exceptions of scalar type are transaction-safe, as well as those types specified as such in clauses 18 and 19.**

## Standard library

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 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 exception-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 is exception-safe.**

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 is transaction-safe.**

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 is transaction-safe.**

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 transaction block (6.x [stmt.tx])**.

Change in 19.2 [std.exceptions] paragraph 3:

> ... These exceptions are related by inheritance. **The exception classes are transaction-safe.**

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

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.]*

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 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.]*

*[ Drafting note: It is unclear whether the math functions from <math.h> may need to inspect hardware state such as the current rounding mode that is unfit for transactional execution. ]*

## Open issues

- Under which circumstances is a lambda function (implicitly) declared transaction-safe? Do we want to allow an explicit transaction_safe/unsafe annotation?
- std::exception is the base of the exception hierarchy. Should we declare its virtual what() function `transaction_safe`? This has serious ripple effects to user code, in particular if that user code is totally unaware of transactions.
- 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.
- Initialization of variables with static storage duration.