Implementability of P0943's C++ `stdatomic.h`

Doc. No: P4230R0
Contact: Hans Boehm (hboehm@google.com)
Audience: LEWG, SG1
Date: May 11, 2026
Target: none
(Thanks to many other contributors in the cited discussions and the LEWG discussion that preceded this. Jonathan Wakely largely started the discussion.)

Abstract

P0943 was accepted into C++23. One of the national body comments for C++26 proposed its removal. SG1 voted without objection to retain it. LEWG almost obtained consensus to remove it. This is an attempt to outline the issues, as mostly exposed in the LEWG discussion, a later reflector discussion ("stdatomic.h in C++", starting March 27, 2026), and some discussion on a github implementation CL. This summarizes points we think should be kept in mind.

We do not propose any change to the standard. However, we are confident that this issue will recur, and should at least be documented. We also expect some of our claims here remain controversial.

Introduction

C and C++ atomics were originally designed as a unit, with C supporting only the specific named atomic types like `atomic_int`, and not the template instances. They were intended to be entirely compatible. However, before P0943 the `stdatomic.h` header was never officially part of C++. And the whole issue became dramatically more complicated with C's introduction of the `_Atomic` type specifiers and qualifiers.

Nonetheless we believe that it is essential to provide an easy way to implement a header file that is both shared between C and C++ and mentions atomics.

P0943 did so (or at least we can agree that it attempted to do so), with an unorthodox approach, by defining stdatomic.h in C++ as a header that defined the C primitives in terms of the C++ ones, so that _Atomic(int) in fact means std::atomic<int> in C++, and relying on the types to be represented identically at the binary level.

P0943 made it clear that the utility of this solution depended on ensuring the C representation and the C++ representation of these atomic types was in fact the same. This was known to not be true at the time.

Current state of affairs

P0943 is based on an Android implementation that has been in use for nearly a decade.

Clang has also long provided an alternate C compatibility mechanism by exposing C's `_Atomic` in C++ as a C-compatible alternative to `std::atomic`. Coexistence of these two facilities is viewed as particularly problematic.

We found this problematic when the initial P0943-like implementation was developed for Android. I recently tried this again on the following simple code using both atomic types, with similar unsatisfactory results:

```
#include <atomic>
#include <stdatomic.h>
_Atomic int x, y;
std::atomic<int> z;

void f() {
    atomic_fetch_add(&x, 1);
    std::atomic_fetch_add(&z, 1);
}
```

It is unclear to us whether the clang approach here is really workable. We suspect that most clients of clangs _Atomic could be converted easily to P0943 by including P0943's `stdatomic.h`.

As far as we can tell, the Android approach works in practice, at least for object sizes that are small powers of 2, provided the alignment is the same as the object size.

In cases with weaker alignment, current implementations are often problematic in that they do not enforce a consistent ABI across compilers, even within the same language. In addition, some implementation strategies perform very poorly. I currently often recommend avoiding these other cases altogether given the ABI issues. As far as I can tell, this is not a unique insight, and there are relatively few such actual

Richard Smith pointed out that the ARM64 ABI actually defines a set of conventions that look good to me. But it seems to not be consistently followed, leading to both serious ABI incompatibilities and occasionally unusable performance for something like an atomic 3-byte struct. IMO, consistent use of this ABI would be a significant step forward. This would also

provide the opportunity, perhaps even necessity, to remove C vs C++ atomic representation incompatibilities.

Concerns about fundamental C vs C++ incompatibilities

C and C++ atomic types often do not share the same implementation. C++ atomics have class type, and C atomics like _Atomic(int) (or _Atomic int) are normally viewed as scalars. This raised concerns that they fundamentally have to be incompatible.

We do not feel that this is the case in most situations, since it really only matters that they have the same binary implementations.

The reflector discussions exposed inherent compatibility issues when atomics are passed as parameters or returned as results, since structs are sometimes passed or returned differently than scalars, even if the bit layout is the same.

However, it is generally not meaningful to pass an atomic (as opposed to a pointer to one) as a parameter. While it may make sense in corner cases to return an atomic in C++ due to guaranteed copy elision, this appears to be a corner case that is easily avoided.

Concerns about ODR violations

There has been a lot of discussion arguing that the C++23 spec essentially forces ODR violations. by implementing atomics as both a struct/class (in C++) and a scalar (in C)

IMO, that depends on your perspective. If you view C code as also C++ code, then ODR violations in a combined C with C++ program make sense. This is the common view of C types in C++.

It is fundamentally not what wg21.link/P0943 does. We treat C and C++ as different languages trying to access the same data structures in memory. The only way in which it builds on the relationship between C and C++ is by observing that we have types that should have the same binary representation on both sides, and the syntax is similar enough that we can provide some definitions (in C++'s version of stdatomic.h) that essentially allow the same syntax to work in either language..

In my view, claiming an ODR violation here is similar to saying that Java's new FFM facility for accessing C memory introduces ODR violations. (See here, if you care)

We agree there may be issues here for whole program optimization across both C and C++.