JTC1/SC22/WG14 - N2771

Title: **C23 Atomics: Problems, Issues, and Proposed Solutions**
Author: **Martin Uecker, University of Göttingen**
Date: **2021-07-11**

Introduction

C11‘s atomics were designed together with WG21 with the goal of achieving compatibility between C and C++. In addition to atomic intege types (7.17.6), one additional feature is that the atomic specifier _Atomic(T) in C and the std::atomic<T> template in C++ both specify types that are ABI compatible. A macro _Atomic(T) in C++ then makes it possible to write code that works for both C and C++. For example, this is useful in shared header files. The _Atomic(T) type specifier was added to C11. With the addition of an _Atomic() macro in C++23 the compatibility story is finally complete – at least when implementations behave as intended [1]. For example, the following code should work in both C and C++.

Example:

```
#ifdef __cplusplus
#include <atomic>
extern "C" {
#endif

void foo(_Atomic(int*) *x);

#ifdef __cplusplus
}
#endif
```

https://godbolt.org/z/oxbhbnnaK

Note that before C++23, the atomic integer types 7.17.6, such as atomic_int, could already be used in an compatible way, but this is limited to integer types.

Despite finally achieving some level of C and C++ compatibility, C11 atomics still have several shortcomings for C programmers which may prevent their use in many applications. Five problems are identified and discussed in this paper.

1. No atomic accesses on existing data structures

Because T and _Atomic(T) are different and incompatible types, is impossible to use atomic accesses on existing data structures without fist copying the data to atomic types, which is often infeasible for large data sets in compute intensive applications. Annotating existing data structures is also often undesirable in systems programming [2]. C++ later added the template std::_Atomic_ref<T> to support the HPC use case [3], but C currently does not provide an equivalent feature – making standard atomics unusable for many HPC applications. For example, a numerical algorithm that adds scattered data to a matrix can not internally use parallelization and atomic accesses without copying the complete data, which can add substantial overhead for larger data sets.

Example:

```
void add_scattered(int N, double a[N],
                   int M, const int x[M], const double y[M])
{
   parallel_for (int i = 0; i < M; i++)
     ((_Atomic T*)a)[x[i]] += y[i];      // UB!
}
```

2. Confusion related to the atomic qualifier

The atomic qualifier _Atomic looks like a qualifier and is called a qualifier but is treated differently from all other qualifiers [1]. This is reflected in confusing rules such as the following, which make the standard difficult to read and to understand.

6.2.5p29 Further, there is the _Atomic qualifier.… The phrase "qualified or unqualified type", without specific mention of atomic, does not include the atomic types.“

In fact, the atomic specifier _Atomic(T) and the atomic qualifier _Atomic T are synonymous and it is not clear why two different syntax exist at all if they are used to denote the same type.

3. No control of alignment and placement of locks

Programmers use C to have low-level control over the layout of data structures. For atomic types, the existing high-level design gives implementations freedom to use different underlying implementation techniques. For example, this is exploited to automatically increase the alignment when this makes it possible to use lockfree hardware instructions, or to automatically insert locks into the type itself. While this is convenient and in line with other library types in C++, it does not provide the low-level control required in systems programming or demanding applications and which is expected by C programmers.

4. ABI Issues

As implementations were free to choose the ABI for _Atomic(T) but this was not always done in a coordinated way, compilers are now incompatible on widely used architectures [5] and even different C++ libraries on the same platform may disagree about the representation of atomic types. The following structure type has different size and alignment on different compilers on x86_64:

Example:

```
_Atomic struct f3 {
  char a[3];
};
```

A related issue is that representation and alignment are now hard-coded in the ABI. Future CPUs with improved support for atomic operations on existing types (e.g. transactional memory) can not be used in an optimal way without breaking the ABI. For example, when a new CPU can implement lockfree accesses with less strict alignment, this can not be exploited. This is also the case when lockfree accesses are supported for new types but this then requires stricter alignment than what the existing atomic type provides. While making ABI breaking changes may be more acceptable for library types provided by C++, for C, which serves an important role as lingua franca of programming languages, the current situation where specific properties of current CPUs are encoded into fundamental types seems less ideal.

5. Atomic generic function are not implementable as first-class functions

Atomic generic functions are not implementable as first-class functions but require compiler magic. In user code, traditional generic APIs using void pointers can not be implemented for atomic operations, because for functions with parameters of type _Atomic void* implicit conversion from other _Atomic qualified types does not work (6.5.16.1). If these conversions would work correctly as for other qualifiers, a first-class function for an atomic compare and exchange operation could be used as in the following example.

Example:

```
_Bool cmpxch(_Atomic void* object, _Atomic void* expected,
             _Atomic void* desired, size_t size);

extern _Atomic int object;
_Atomic int expected = 0;
_Atomic int desired = 1;

while (cmpxch(&object, &expected,   // constraint violations
              &desired, sizeof(object)))
    ….
```

Discussion

In principle, none of these five problems would exist if atomic types were required to have the same representation and alignment as the corresponding non-atomic type, i.e. if _Atomic were a proper qualifier as suggested previously [4]:

1. Pointers to existing data structures could be converted to atomic pointers according to the usual conversion rules for qualifiers.

2. The language rules related to qualifiers would work consistently as the atomic qualifier would be a genuine qualifier. All related special cases would not exist.

3. Programmers would have more control over alignment and data layout.

4. ABI would have been fully determined to be compatible with non-atomic type, leaving no room to implementations for introducing inconsistencies. Improved support in atomics in newer CPUs could be used directly when available.

5. A generic API based on atomic-qualified void pointers could be implemented as first-class functions.

On the other hand, there are also advantages of the existing design:

* Optimal alignment or representation is automatically chosen to obtain optimal performance using lockfree accesses whenever this is currently possible.

* Optimal alignment/representation is guaranteed based on the type, which can avoid run-time checks for alignment of a pointer for lockfree accesses.

* Implementations have the freedom to completely change the representation of a type, e.g. by inserting a lock into the atomic type. Inserting a lock makes atomic accesses address-free even for types where lockfree accesses can't be used. This is important on platforms where different software component can not easily create a shared lock table (e.g. different C run-times linked into the same program).

We discuss two alternative paths forward which both aim to preserve the advantages of existing atomics while adding the missing functionality. In both cases:

* _Atomic(T) continues to specify an atomic type which does not need to have the same representation and alignment as T. Thus, full compatibility with the C++ template types can be retained and implementations are still free to include locks into such types.

* A qualifier-based solution is used to address the limitations of the existing atomic types addressing similar use cases as std::_Atomic_ref<T> in C++.

Alternative 1: A new qualifier

* The _Atomic qualifier is removed (or deprecated now and later removed).

* Instead, a new qualifier _Lock (or another name) is introduced that is a genuine qualifier and specifies a type _Lock T with the same representation and alignment as the unqualified type T. This qualifier then supports using locked operations on existing data structures similar to std::_Atomic_ref<T> in C++. For types which do not naturally support lockless accesses, this can be implemented using shared lock tables as already used for C11 atomics on most implementations and required for std::_Atomic_ref<T> on all platforms.

* _Atomic(T) may be implemented as a _Lock qualified type with stricter alignment or using a completely different implementation strategy.

Alternative 2: Reusing the atomic qualifier

In this approach, the atomic qualifiers is reused and now becomes a genuine qualifier. This means that _Atomic(T) and _Atomic T are then split into two different types. _Atomic T would be required to have the same representation and alignment as T, but _Atomic(T) is allowed to be a different type. While in principle this must be considered to be a breaking change, it seems that there is still a small chance of making this change without impacting existing code too much. This might be possible for the following reasons:

* Many compilers still do not support C11 atomics at this point in time and some support them only partially. (Atomics are an optional language feature.)

* Currently, there seems to be no C implementation which does include a lock into an atomic type. There are C++ implementations which do this and which plan to add atomic support for their corresponding C implementation. These implementations could then add a lock into _Atomic(T) for compatibility with C++ but directly start using an alternative implementation for the _Atomic qualifier (similar to std::_Atomic_ref<T>).

* For platforms already supporting C11 atomics (and where atomic types do not include locks), the consequences would be following:

    - The size of _Atomic T for some structure types T would change on some implementations because additional padding now included in the atomic type needs to be removed again. These are exactly the types where ABI divergence between compilers exists [5] and which can therefor not be used in portable APIs. Impact on existing code should therefor be minimal. For the same reason, compilers which now add padding to such structures could also consider an ABI breaking change for _Atomic(T) on C and C++. This would then restore ABI compatibility between different implementations (also between different C++ libraries).

- For some types, the alignment of _Atomic T would be less strict than the alignment of _Atomic(T). Because compilers can simply choose stricter alignment for objects the impact could be kept minimal for existing code. The necessary changes are 1) the new alignment returned by _Alignof for atomic-qualified types, 2) conversions between pointers to _Atomic T and _Atomic(T) are a constraint violation when the alignment becomes stricter and 3) pointers to atomic-qualified types may (in rare cases) now require a run-time check before using lockless accesses. This run-time check is also exactly what is required to support atomic accesses on non-atomic types. For the first and second change, a compiler diagnostic may help avoid problems, while the third change can be introduced in a backwards compatible way without affecting existing code.

Conclusion

Alternative 2 is tempting as it would simply remove all special cases for the atomic qualifier making it a genuine qualifier. This would substantially simplify the language. For many platforms the _Atomic T and _Atomic(T) could then be the same type or only differ in alignment. One downside is that in general _Atomic(T) and _Atomic T could still be completely different types, which may be confusing (but probably less confusing than the current situation). Another issue is that Alternative 2 forces an ABI change for _Atomic T on some implementations. While this could also be viewed as a good thing, as it would restore compatibility between diverging implementations on affected architectures, this might not find consensus everywhere and is certainly a change with a higher risk.

Alternative 1 would avoid confusion by introducing a new qualifier with a different name. A new qualifier also avoids all potential compatibility issues for existing code. Removing the old _Atomic qualifier would then require some code to adapt by transitioning to the specifier _Atomic(T) or the new qualifier. If the old_Atomic qualifier is not immediately removed, there would be a transitional period where the standard text will have wording for the new qualifier and also retains all the special cases which now exist for the atomic qualifier.

References

[1] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0943r6.html
[2] https://gcc.gnu.org/legacy-ml/gcc/2012-02/msg00027.html
[3] http://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p0019r8.html
[4] N1536 2010/11/04 Garst, _Atomic Qualifier Issues
[5] https://godbolt.org/z/r443Wz89E