# _Atomic Qualifier Issues
# N1536

Blaine Garst
Nov 3, 2010

This is the first of two promised papers on possible resolutions to issues raised with the current formulation of atomics in C1x draft n1516, as amended by adoption of several proposals through the end of the meeting on November 3.

The key issue not fully resolved is the impact of the statement in Section 6.2.5 paragraph 27 allowing a qualified type to have different size, representation, and/or alignment.

27 Further, there is the **`_Atomic`** qualifier, which may combine with **`volatile, const`** and **`restrict`**. The size, representation, and alignment of an _Atomic-qualified type need not be the same as those of the corresponding unqualified type. (Atomic types are a conditional feature that implementations need not support; see 6.10.8.3.)

This is in conflict, for example, with Section 6.3.2.3 paragraph 2

2 For any qualifier *q*, a pointer to a non-*q*-qualified type may be converted to a pointer to the *q*-qualified version of the type; the values stored in the original and converted pointers shall compare equal.

(By way of explanation, my simplistic reading of the type compatibility section when I crafted n???? seemed to indicate that types with differing alignments were in fact compatible and that, as such, implicit conversions were already in place to handle the representation differences.)

Forbidding the conversion of a pointer to an _Atomic-qualified type to that of its unqualified counterpart must be done, particularly because any pointer arithmetic done on one cannot be translated back to the other even if no accesses are attempted. The wording is quite complicated, however, since *there is a deep assumption that one can speak of qualified and unqualified pointers to compatible types*. The restriction can only be worded as an exception to this rule if the representation of the _Atomic-qualified type matches that of the unqualified type.

There were approximately eight such issues that needed resolution as of Wednesday morning.

My review of these issues has led to two more that deal with _Atomic-qualified types where the R&A is allowed to be different.

First, is the following legal:

```
union {
   _Atomic(long double) ld;
  _Atomic (char [16]) atomicArray;
  char nonAtomicArray[16];
} u;

u.ld = ATOMIC_VAR_INIT(sin(2.2));
char c = u.atomicArray[10];    // allowed??
char d = u.nonAtomicArray[10];  // illegal: 6.7.3 access via non-_Atomic value
u.atomicArray[3] = ATOMIC_VAR_INIT(10);  // legal?
```

If _Atomic(char) required extra R&A then how did it get initialized? How can type-punning of these types ever work?

One can argue that judicious use of ATOMIC_VAR_INIT and the **`atomic_is_lock_free`** generic library function might save the day but, frankly, I'm not sure. If both _Atomic-qualified types are not lock free where in the specification does it forbid type punning among them then, and allow it otherwise?

This situation also raises the issue of a data race. Clearly two threads can smash bit representations via legal atomic operations unless they coordinate by some other means; this seems germane to Larry Jones' issue as to whether _Atomic structure and union members should be allowed even if undefined behavior may result.

I recall Paul McKenney offering to look into issues with _Atomic-qualified union members at the meeting in Colorado. I think there probably are some.

The second issue is closely related, but my reading of the treatment of allocated storage seems to imply that one can also use and reuse arbitrary sections of allocated memory with relative impunity, including the ability to set up a data race though pointers to _Atomic-qualified types.

## The Alternative

The simple alternative is to remove the provision that allows _Atomic-qualified types to have different size, representation, and alignment:

Change Section 6.2.5 paragraph 27 to:

27 Further, there is the **`_Atomic`** qualifier, which may combine with **`volatile, const`** and **`restrict`**. (Atomic types are a conditional feature that implementations need not support; see 6.10.8.3.)

This resolves all eight known issues (leaving some minor considerations).

**What have we lost if we do this? What have we gained?**

For C1x implementors, we take away the compiler's right to do more optimal size and alignment behind your back. Without reading the compiler manual, one might get surprised when one's _Atomic struct stat statb has different layout than your normal one (presuming that _Atomic-qualifier propagates to members as does volatile and const). In fact, given that even pointer casting is disallowed, it's hard to say that silent but different alignment is a good idea for any existing C data structures, including the hordes defined by various operating systems.

Given that we have introduced alignment directives, a savvy programmer building new data structures has reasonably well-defined tools to build a data structure of known shape that is optimal for as many platforms (or compilers) that she is interested in.

This handles the issues of size and alignment that are going to be most prevalent. Any implementation that requires a lock, by any means, for atomic updates is, by definition, not going to be performant w.r.t. other native data types that do not. I strongly believe that these data structures will never (long) be in a critical path without being re-engineered, and that simple correctness is all that the Standard needs to assure.

Any compiler that choses differing size, representation, or alignment for _Atomic-qualified types has an ABI interoperability issue with other C1x compilers for that platform as well as C++0x compilers. This is a very strong disincentive.

The gain is, what you see is again what you get, other than the requirements for atomicity, from a representation and type system viewpoint.  Issues of aliasing don't get even more complicated, type punning remains viable to _Atomic types, and others.

My list of known existing commercial architectures that would in some way require differing R&A is currently zero.  There were reportedly some early versions of sparc and hpux kernels that chose to use a 24-bit integer representation to allow a locking byte.

**Where does this leave compatibility with C++0x?**

As Bill Plauger noted, the continued existence of the atomic_xyz types continues to provide the C++0x minimum level of compatibility that was already acceptable.  The worst case is that a C1x implementation chooses to implement those compatibility types with extended RS&A, and that they do not interchange with their _Atomic-qualified counterparts.  Those types may or may not be representationally the same as their corresponding _Atomic-qualified equivalents, but that doesn't matter - the compatibility and interoperability across languages is still there.

The better case is that for that list of explicit types there is no RS&A difference, and the implementation chooses to equate them with _Atomic-qualified equivalents, as is described today.

For all C++0x compilers for all types that do not require differing R&A, the types are still compatible, and so we still have a C1x solution that has greater compatibility.  These will, in my opinion, be the dominant use cases on all major platforms.

## Additional Issues

The only remaining issue regarding representation has to do with platform supporting 1's complement representations.  These types will not have compatible _Atomic-qualified and unqualified forms.

My recommendation is to forbid this for comforting implementations:

Section 6.10.8.3 Conditional feature macros

Add to __STD_NO_THREADS__:

Implementations with integral types using 1's complement representation shall define this macro.