M A Y 2 7 , 2 0 1 0
Draft 10

ATOMIC PROPOSAL, N1485

BLAINE GARST

APPLE INC.

blaine@apple.com

Introduction

The C1x atomics section 7.16 was crafted with a keen eye to compatibility with C++0x so that there were clear and unambiguous ways of writing code portable across the two languages. A somewhat informally determined design constraint at the beginning precluded direct language treatment for atomics in C, unlike that which atomics enjoys in C++0x. In fact, the library-only approach for C1x allows only the integral types and a limited set of predefined atomic typedefs for some of the other, but not all, typedefs in C1x. There is no compile time mechanism possible to extend the typedefs to other popular platform specific typedefs such as int32_t, thus limiting portability of code that would otherwise use such useful typedefs.

The proposal is to add an _Atomic type qualifier to the C1x language such that atomics can also apply to the full set of C1x types in a natural way, including direct use of the compound assignment operators. Through several revisions incorporating suggestions from Lawrence Crowl, Paul McKenney, Hans Boehm, and Thomas Plum, **this proposal unambiguously increases compatibility with C++0x**. This is achieved by preserving not only the operational semantics of sequential-consistency but also representational equivalence leading to direct interoperability of _Atomic qualified objects with their C++0x atomic<T> counterparts. Moreover, an additional declarative syntax is proposed such that C1x atomic programs can be written that compile, run, and interoperate directly with C++0x.

The interoperability is so great that we, at Lawrence's suggestion, can define the fundamental atomic_*T* types introduced in the atomics section to be _Atomic *T* and gain operator overloading of these types much as is possible in C++0x.

High Level Description

1. A new keyword _Atomic is introduced. It is used, alone, as a type qualifier. An implementation is allowed to relax the requirement of having the same representation and alignment of the corresponding non-atomic type, as long as appropriate conversions are made, including via the cast operator.

2. The _Atomic keyword also can also be used in the form _Atomic(T), where T is a type, as a type specifier equivalent to _Atomic T. Thus, _Atomic(T) x, y; declares x and y with the same type, even if T is a pointer type. This allows for trivial C++0x compatibility with a C++ only _Atomic(T) macro definition as atomic<T>.

3. The existing atomic_int et al. typedefs are redefined to be typedefs of _Atomic counterparts.

4. The compound assignment operators (e.g. +=, /=, ^= , ...,) and all forms of ++ and -- are overloaded on atomic objects to provide sequentially-consistent read-modify-write atomic behavior, whether achieved directly via hardware instructions, by small inline compare-and-swap code sequences, or by support from compiler intrinsics, as the implementation decides. Thus, even atomic float identifiers can make use of sequentially-consistent operators..

5. Ordinary read and write access to _Atomic identifiers is sequentially-consistent, requiring acquire fences as necessary on the hardware before every read, and release fences after writes so that dependent data can be properly provided.

6. A new ?= : ternary assignment operator is introduced to express an assign value if identifier has the specific value operation. When used upon an atomic qualified identifier, a sequentially-consistent compare-and-swap operation will be performed. This is the only mechanism for doing compare-and-swap on atomic bitfields.

_Atomic applies uniformly to all types, including scalars, arrays, and aggregates, yet has little to no meaning for function results and parameters. When applied to structures and unions _Atomic offers guarantees of complete loads and stores via assignment. Field access of atomic structures and unions presents an inherent data-race that is expressly left to the implementation to decide and hence leads non-portable undefined behavior which very closely matches the restrictions in C++0x which are impossible to implement exactly.

Modifications

The following are the formal descriptions of changes necessary to the **N1425** proposed draft standard to reflect the preceding high level descriptions of atomic.

1. Section 6.2.5 para 26+1 (new paragraph after 26)

Further, there is the _Atomic qualifier which may combine with **volatile** and **restrict**. The size and alignment for _Atomic qualified objects are allowed to be implementation defined independently of the underlying unqualified type as long as requisite representational conversions are done for all implicit or explicit cases.

2. Section 6.2.6.1

Loads and stores of _Atomic objects are done with sequentially-consistent memory order.

3. Section 6.4.1,

Add _Atomic to the list of keywords.

4. Section 6.4.6

Add ?= to list of punctuators.

5. Section 6.5.2.4

Postfix ++ and -- operations on atomic objects are atomic read-modify with sequentially-consistent memory order. [Note: where pointers to atomic objects can be formed, these operations are equivalent, where T is the type of E1, to the code sequence

```
T tmp;
T result = E1;
do {
    tmp = result OP 1;
} while (!atomic_compare_exchange_strong(&E1, &result, tmp));
```

where *result* is the result of the operation.]

6. Section 6.5.3.1

Prefix ++ and -- operations on atomic objects are atomic read-modify with sequentially-consistent memory order. [Note: where pointers to atomic objects can be formed, these operations are equivalent, where T is the type of E1, to the code sequence

```
T result;
T expected = E1;
do {
    result = expected OP 1;
} while (!atomic_compare_exchange_strong(&E1, &expected, result));
```

where *result* is the result of the operation.]

7. Section 6.5.16.2 Compound Assignment, para 3, Semantics

If E1 is atomic, the operations are atomic read-modify with sequentially-consistent memory order. [Note: where pointers to atomic objects can be formed, these operations are equivalent, where T is the type of E1, to the code sequence

```
T result;
T expected = E1;
do {
    result = expected OP E2;
} while (!atomic_compare_exchange_strong(&E1, &expected, result));
```

where *result* is the result of the operation.]

8. Section 6.5.16.3 The ternary assignment operator ?= :

The ternary operator *lvalue ?= candidateExpr : replacementExpr* tests that the *lvalue* has *candidateExpr* value and if so assigns the *replacementExpr* value, yielding a boolean value affirming or denying that the assignment occurred. If lvalue is atomic, the test and assignment must be performed with sequentially-consistent behavior, e.g. the test-and-set

must be done atomically, with memory-load and memory-store fences. The test is done as if the == operator were used, and if equal, will replace the value, as is done with atomic_compare_exchange_strong.

[Example: lazy initialization

```
_Atomic struct x *GlobalPX = NULL;
struct x *getGlobalPX() {
    struct x *returnValue;
    // the read of GlobalPX synchronizes with assignment to member
    if (! (returnValue = GlobalPX)) {
          struct  x *tmp = (struct x *)malloc(sizeof(struct x));
          tmp->member = value;
          if (GlobalPX ?= NULL : tmp)
              returnValue = tmp;
          else
              free(tmp);
    }
    return returnValue;
}
```

9. Section 6.7.2 Type Specifiers

Paragraph 1: Add to declaration productions:

_Atomic (*type-name*) init-declarator-list[opt] ;

Paragraph 2: add _Atomic (*type-name*) to the list

(new paragraph)

The _Atomic production qualifies type-name with _Atomic.

10. Section 6.7.3 Type Qualifiers

Paragraph 1: Add _Atomic to the list.

Paragraph 2: (Constraints)

Access to a member of an _Atomic qualified structure or union results in undefined behavior. [Note: A *data-race* would occur if access to the entire structure in one thread conflicts with access to a member from another thread, where one or more of such accesses is a modification, and such a *data-race* results in undefined behavior].

Paragraph 5: If an attempt is made to refer to an object defined with an _Atomic qualified type through use of an lvalue with non-_Atomic-qualified type, the behavior is undefined.

11. Section 7.16.5 Atomic integer and address types

Paragraph 1: For each line in the following table, atomic typedef names are provided as defined in the direct type column

Atomic typedef	Direct type
atomic_char	_Atomic char

...

Paragraph 2: same transformation as in Paragraph 1.

12. Section 7.16.6 Operations on integer and address types

Re-title section to be “Operations on atomic types”