Document: N1560 Date: 15-March-2011

Harmonize C1x and C++0x atomics

Authors:
Hans Boehm
Lawrence Crowl
Detlef Vollmann
Anthony Williams
Michael Wong
Rajan Bhakta
Raymond Mak
Reply to: michaelw@ca.ibm.com

This paper expands on ballot issues raised by Canada, Switzerland, and the US in response to the C1x 2011 CD ballot.

This will describe issues using Canadian ballot numbers, and they may correspond with US or Switzerland ballot numbers.

Currently, a few aspects of C1x atomics are not compatible with C++0x atomics, such that atomics code legal in C1x may be undefined or yield a diagnostics in C++0x. This will destroy the original goal of having atomics compatible between C and C++.

We believe the current trend of productive syntax for C1x is the right direction. However, we would like to harmonize the C1x and C++0x atomics. We aim to have much more of the C1x atomics syntax work in C++.

We believe we can do this with the following changes.

1. The _Atomic type qualifier should be removed. It is redundant, and its use needlessly hinders C++ compilation of code. For example, we believe this function declaration is ambiguous in the current C draft:

```
int func(_Atomic(int))
```

can mean a func that takes an atomic int or a function that takes a function that returns an atomic int.

We prefer _Atomic not be a qualifier on the function return. If we remove the second meaning, then C++ can define _Atomic as a macro that expands to a template id.

C++ can take the symbols, and promote them to the global namespace.

2. Remove atomic_address in C1x, with a productive syntax, the type is redundant. Furthermore, C++ has removed atomic_address in C++0x. This was removed because it was a base class of the pointer specialization, which leads to its use breaking type safety.

3. The current draft supports too many compound operations, like atomic divide assign for atomic float for arithmetic operations. It is trying to be too general making every compound operator atomic. The original C1x paper (Footnote 113) implies that these operations can be written 'as if' with a compare exchange loop and that might work, but we need to understand it better. These additional operations currently make some C code incompatible with C++. We would prefer a similar phased approach until we understand the implication of such support. C++ selectively narrowed the operations based on what current hardware will not have trouble supporting. This limited support is not permanent, Additional operations can be added at a later time. Until we specify what the extended operations mean, in the presence of traps, we would prefer that C1x limit the supported operations to the same list as C++0x.

Here are the limited set of operations that C++ supports that we would like C to restrict::

	type made atomic			
Operations	bool	T*	integral	others
is_lock_free	Y	Y	Y	Y
load, store, exchange, compare_exchange (weak+strong)	Y	Y	Y	Y
fetch_add, +=, fetch_sub, -= ++,--		Y	Y	
fetch_or, |=, fetch_and, &=. fetch_xor, ^=			Y	

4. There is a current macro that says if you define __STDC_NO_THREADS, then you don't need to provide the stdatomic.h header.

The integer constant **`1`**, intended to indicate that the implementation does not support atomic types (including the **`_Atomic`** type qualifier and the **`<stdatomic.h>`** header) or the **`<threads.h>`** header.

Threads and atomics are different things. Specifically, threads belong to the OS and atomics belong to the hardware. The OS should provide common locking implementations so that different compilers don't do incompatible things. In embedded system, you want hardware support and not have OS come along for the ride. We should separate __STDC_NO_THREADS from stdatomic.h. We suggest __STDCATOMIC__ be defined if the system provides atomics and the atomic header.

5. We would like C1x to remove atomic to atomic assignment as C++0x has removed it. Many programmers are likely to assume that assigning an atomic directly to another

atomic is a single atomic operation, as with += on an atomic. We are not sure about the correct interpretation of the current draft on this point, but this assumption must be incorrect, since very few implementations can reasonably support it. Since direct atomic-to-atomic assignment very rarely appears in correct programs, it should trigger a diagnostic, forcing the programmer to clearly restate it as two separate atomic operations

6. The current mutex API is substantially different from both C++0x and POSIX APIs. It is based on an API which currently has few direct clients. At a minimum, the removal of mtx_try as in N1521 (also WG14 N1437) should be reconsidered.

However, we would much prefer the C mutex type to be the same as the C++ mutex type. C++ mutex types were designed to make that compatibility possible. It will be embarrassing if we don't have the same mutex type. Originally, the syntax was not specified because the C++ committee did not want to assume a C syntax. Now that there is a syntax, this argument is moot. C mutexes are local objects and while C++ may put wrapper around C mutex for member functions, this will make condition variables fail to work. Condition variables only work with the C++ mutex type.

From Lawrence:
As locks become more fine-grained, the time for un-contended lock acquisition can be a significant cost. The C mutex design necessarily adds conditional branches into each operation as it figures out which lock is available. What is supplied by OS facility usually is too slow because it tries to be fair and does not scale well. When I was actively doing a lot of parallel programming, I avoided OS synchronization because nothing would scale well.

From Hans:
The current C mutex design has three unfortunate characteristics when used in a context that also supports C++ code: First, it is difficult to declare a mutex in a header file such that it can be accessed by both C and C++. C++ mutexes are not accessible from C. C mutexes are accessible from C++ only through a wrapper not provided by any standard library. Second, the interfaces differ gratuitously in ways that require significant additional learning time for users of both languages, with no discernable benefit. Third, C mutexes are inherently slower than their directly implemented C++ counterparts, since they require dynamic testing of the mutex type. We believe that, especially with the introduction of mtx_try to differentiate it from established OS APIs, the current C mutex design needlessly diverges from mainstream practice, both in OS APIs and in C++.

We would like to have the C mutex type and the C++ mutex type be exactly the same type, as was done for atomics. We believe the C++ design leads to better performance, especially when we start scaling the system. This is one of the most urgent requirement for compatibility.

7. The locking behavior of I/O functions is not specified. This may result in unexpected behavior in multithread contexts and require explicit locking that will be redundant on most implementations. Require implicit locking or provide for efficient explicit locking.

C++ has taken the minimal step of requiring that I/O functions not introduce data races. They may scramble the contents, but they will not introduce undefined behavior.

8. The library section should be examined for threads incompatibilities. (See WG 14 N1371 – 2009-03-21) Obviously threads-incompatible functions include strtok and rand. Thread safe versions should be included.

9. C and C++ differ in how they indicate optional facilities. C tends to have "#if !defined(NOFOO) #include <foo.h> #else …." whereas C++ tends to just neuter the header file, making it hard for programmers to either adapt to lack of atomic support or to determine the cause of the consequential compilation problems. We would rather see the languages define a positive sense macro indicating that the feature is present, and not provide the header if the feature is not present. Programmers that can adapt will test the macros, programmers that cannot will just include the header and let a failed include indicate lack of support.