Document Number: WG14 N1131
Date: 2005-08-26
Reply to: Hans Boehm
Hans.Boehm@hp.com
1501 Page Mill Rd., MS 1138
Palo Alto CA 94304 USA

# Implications of C++ Memory Model Discussions on the C Language

Hans Boehm Doug Lea Bill Pugh

**Abstract**

A number of us (Andrei Alexandrescu, Hans Boehm, Peter Dimov, Kevlin Henney, Ben Hutchings, Doug Lea, Bill Pugh, Alexander Terekhov, and others) have been working on the semantics of multithreaded programs in C++. We feel that many of the same issues apply to the C language, and that it is important to keep the solutions as similar as possible.

Although we would like to concentrate our effort on one language at a time, we would like to be made aware of any C-specific issues that are likely to arise, or any other insights from members of the C committee. To that end, this is a very quick overview of the issues that drive the C++ work, and a pointer to the relevant C++ documents.

## 1 Overview

Mainstream desktop and server machines increasingly require explicitly concurrent programs to achieve full performance, due to the increasing prevalence of both single-chip multiprocessors and hardware support for multiple threads.

Presently a common way to write such programs is to program in C or C++, with the aid of a threads library, such as a Pthreads implementation, to provide concurrency. This is also an established technique for handling multiple concurrent event streams, even on single-threaded single processor machines.

Unfortunately, this approach has turned out not to be completely sound, primarily because reliable multi-threaded execution requires certain guarantees about the language and compiler that cannot easily be provided a library or library specification[1]. Some of the associated issues have been understood for many years. The second half of this paper briefly outlines a symptom of this issue which appears to not have been well-recognized.

As a result, several of us have started an effort to address these problems by directly defining the meaning of multithreaded programs in the underlying programming language. Initially this is being done in the context of C++, building on some earlier work in the context of Java [3, 2].

This has resulted in three papers in C++ committee mailings: WG21/N1680=J16/04-0120, WG21/N1777=J16/05-0037, and WG21/N1876=J16/05-0136. The issue was discussed at the Redmond and Lillehammer meetings. There appears to be a consensus that this should be addressed in the next revision of the C++ standard.

Current proposals and discussions can be reached from `http://www.hpl.hp.com/personal/Hans_Boehm/c++mm`. The primary purpose of this short paper is to call attention to those discussions and related materials.

In the context of C++ we are addressing three somewhat separable issues:

1. Defining the meaning of existing programs in the presence of threads. Our current approach largely follows Pthreads and leaves the semantics undefined if there is a *data race*, i.e. if a program modifies a location while another thread is accessing it.

   This approach appears to be the only plausible one for C and C++. However, it can only succeed if the definition of a data race is made precise enough for programmers, compiler writers, and hardware to know when data races occur and how to avoid them. Currently, it is not defined at all. Among other consequences, unexpected compiler transformations regularly break multithreaded programs (as illustrated in the example below).

2. Defining an atomic operations library to allow the construction of correct multithreaded programs without locks. This does not directly affect most existing application-level programs, though a significant number of them should be modified to use this library in order to ensure correctness. Such a library is necessary for development of portable core libraries and infrastructure code that increasingly use lock-free techniques to implement high-performance synchronization support. Defining an atomics library relies critically on the semantics of memory operations and data races.

3. Designing a threads API that meshes better with the rest of the C++ language.

We expect that the first two issues and their solutions also apply, with minor modifications, to C. And compatibility would be greatly desirable. I expect the last issue is mostly C++-specific, though there are likely to be exceptions, such as support for thread-local storage.

As a result we would like to encourage members of the C committee to follow our discussions, and to provide input, particularly if they see aspects of our approach that would make it less palatable to the C committee, and hence lead to unnecessary divergence between C and C++.

## 2 A Simplified Example

We illustrate some of the problems addressed by this work with a simple case in which the current language specifications for C and C++ are clearly inadequate for multithreaded programs. This is only one among many possible examples. It helps demonstrate that the problems are in fact profound, and must be addressed by the language specification and compilers. It also points out that the expected impact on compilers is likely to be nontrivial.

Consider the following declarations and function definition:

```
int global_positive_count = 0;

typedef struct list_struct {
    struct list_struct *next;
    double val; } * list;

void count_positives(list l)
{
    list p;

    for (p = l; p != 0; p = p -> next)
        if (p -> val > 0.0)
            ++global_positive_count;
}
```

Now consider the case in which thread *A* performs

```
count_positives(<list containing only negative values>);
```

while thread *B* performs

```
          ++global_positive_count;
```

This should be perfectly correct, since `count_positives`, in this specific case, does not update `global_positive_count`, and hence the two threads operate on distinct global data, and require no locking.

But some existing optimizing compilers (including gcc, which tends to be relatively conservative) will "optimize" `count_positives` to something similar to:

```
void count_positives(list l)
{
    list p;
    register int r = global_positive_count;

    for (p = l; p != 0; p = p -> next)
        if (p -> val > 0.0) ++r;
    global_positive_count = r;
}
```

This transformation is clearly consistent with the C language specification, which addresses only single-threaded execution. In a single-threaded environment, it is indistinguishable from the original.

The Pthread specification also contains no clear prohibition against this kind of transformation. And since it is a library and not a language specification, it is not clear that it could.

However, in a multithreaded environment, the transformed version is quite different, in that it assigns to `global_positive_count`, even if the list contains only negative elements. Our original program is now broken, since the update of `global_positive_count` by thread $B$ may be lost, as a result of thread $A$ writing back an earlier value of `global_positive_count`. By Pthread rules, a thread-unaware compiler has turned a perfectly legitimate program into one with undefined semantics.

Some more realistic examples, including some that have been encountered in practice, can be found in [1]. (It also reviews yet another reason why programs like the above may fail.) But I hope this example has served as a brief introduction to the kind of problems we are trying to address, and hopefully encouraged others to follow the discussion.

## References

[1] Hans Boehm. Threads cannot be implented as a library. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 26–37, 2005.

[2] JSR 133 Expert Group. Jsr-133: Java memory model and thread specification. `http://www.cs.umd.edu/~pugh/java/memoryModel/jsr133.pdf`, August 2004.

[3] Jeremy Manson, William Pugh, and Sarita Adve. The java memory model. In *Conference Record of the Thirty-Second Annual ACM Symposium on Principles of Programming Languages*, January 2005.