C++ and the Perils of Double-Checked Locking [*]

Scott Meyers and Andrei Alexandrescu

September 2004

> Multithreading is just one damn thing after, before, or simultaneous with another.

1 Introduction

Google the newsgroups or the web for the names of various design patterns, and you're sure to find that one of the most commonly mentioned is Singleton. Try to put Singleton into practice, however, and you're all but certain to bump into a significant limitation: as traditionally implemented (and as we explain below), Singleton isn't thread-safe.

Much effort has been put into addressing this shortcoming. One of the most popular approaches is a design pattern in its own right, the *Double-Checked Locking Pattern* (DCLP) [13, 14]. DCLP is designed to add efficient thread-safety to initialization of a shared resource (such as a Singleton), but it has a problem: it's not reliable. Furthermore, there's virtually no portable way to make it reliable in C++ (or in C) without substantively modifying the conventional pattern implementation. To make matters even more interesting, DCLP can fail for different reasons on uniprocessor and multiprocessor architectures.

This article explains why Singleton isn't thread safe, how DCLP attempts to address that problem, why DCLP may fail on both uni- and multiprocessor architectures, and why you can't (portably) do anything about it. Along the way, it clarifies the relationships among statement ordering in source code, sequence points, compiler and hardware optimizations, and the actual order of statement execution. Finally, it concludes with some suggestions regarding how to add thread-safety to Singleton (and similar constructs) such that the resulting code is both reliable and efficient.

2 The Singleton Pattern and Multithreading

The traditional implementation of the Singleton Pattern [7] is based on making a pointer point to a new object the first time the object is requested:

[*] This is a slightly-modified version of an article that appeared in *Dr. Dobbs Journal* in the July (Part I) and August (Part II), 2004, issues.

```
1  // from the header file
2  class Singleton {
3  public:
4  static Singleton* instance();
5    ...
6  private:
7    static Singleton* pInstance;
8  };
9
10 // from the implementation file
11 Singleton* Singleton::pInstance = 0;
12
13 Singleton* Singleton::instance() {
14   if (pInstance == 0) {
15     pInstance = new Singleton;
16   }
17   return pInstance;
18 }
```

In a single-threaded environment, this generally works fine, though interrupts can be problematic. If you are in `Singleton::instance`, receive an interrupt, and invoke `Singleton::instance` from the handler, you can see how you'd get into trouble. Interrupts aside, however, this implementation works fine in a single-threaded environment.

Unfortunately, this implementation is not reliable in a multithreaded environment. Suppose that Thread *A* enters the `instance` function, executes through Line 14, and is then suspended. At the point where it is suspended, it has just determined that `pInstance` is null, i.e., that no `Singleton` object has yet been created.

Thread *B* now enters `instance` and executes Line 14. It sees that `pInstance` is null, so it proceeds to Line 15 and creates a `Singleton` for `pInstance` to point to. It then returns `pInstance` to `instance`'s caller.

At some point later, Thread *A* is allowed to continue running, and the first thing it does is move to Line 15, where it conjures up *another* `Singleton` object and makes `pInstance` point to it. It should be clear that this violates the meaning of a singleton, as there are now two `Singleton` objects.

Technically, Line 11 is where `pInstance` is initialized, but for practical purposes, it's Line 15 that makes it point where we want it to, so for the remainder of this article, we'll treat Line 15 as the point where `pInstance` is initialized.

Making the classic Singleton implementation thread safe is easy. Just acquire a lock before testing `pInstance`:

```
Singleton* Singleton::instance() {
  Lock lock;       // acquire lock (params omitted for simplicity)
  if (pInstance == 0) {
    pInstance = new Singleton;
  }
  return pInstance;
}                   // release lock (via Lock destructor)
```

The downside to this solution is that it may be expensive. Each access to the Singleton requires acquisition of a lock, but in reality, we need a lock only when initializing `pInstance`. That should occur only the first time `instance` is called. If `instance` is called n times during the course of a program run, we need the lock only for the first call. Why pay for n lock acquisitions when you know that $n - 1$ of them are unnecessary? DCLP is designed to prevent you from having to.

3 The Double-Checked Locking Pattern

The crux of DCLP is the observation that most calls to `instance` will see that `pInstance` is non-null, hence not even try to initialize it. Therefore, DCLP tests `pInstance` for nullness before trying to acquire a lock. Only if the test succeeds (i.e., if `pInstance` has not yet been initialized) is the lock acquired, and after that the test is performed again to make sure `pInstance` is still null (hence the name *double-checked* locking). The second test is necessary, because, as we just saw, it is possible that another thread happened to initialize `pInstance` between the time `pInstance` was first tested and the time the lock was acquired.

Here's the classic DCLP implementation[13, 14]:

```
Singleton* Singleton::instance() {
  if (pInstance == 0) {          // 1st test
    Lock lock;
    if (pInstance == 0) {        // 2nd test
      pInstance = new Singleton;
    }
  }
  return pInstance;
}
```

The papers defining DCLP discuss some implementation issues (e.g., the importance of `volatile`-qualifying the singleton pointer and the impact of separate caches on multiprocessor systems, both of which we address below; as well as the need to ensure the atomicity of certain reads and writes, which we do not discuss in this article), but they fail to consider a much more fundamental problem, that of *ensuring that the machine instructions executed during DCLP are executed in an acceptable order*. It is this fundamental problem we focus on here.

4 DCLP and Instruction Ordering

Consider again the line that initializes `pInstance`:

```
pInstance = new Singleton;
```

This statement causes three things to happen:

Step 1: Allocate memory to hold a Singleton object.

Step 2: Construct a Singleton object in the allocated memory.

Step 3: Make pInstance point to the allocated memory.

Of critical importance is the observation that compilers are not constrained to perform these steps in this order! In particular, compilers are sometimes allowed to swap steps 2 and 3. Why they might want to do that is a question we'll address in a moment. For now, let's focus on what happens if they do.

Consider the following code, where we've expanded pInstance's initialization line into the three constituent tasks we mentioned above and where we've merged steps 1 (memory allocation) and 3 (pInstance assignment) into a single statement that precedes step 2 (Singleton construction). The idea is not that a human would write this code. Rather, it's that a compiler might generate code equivalent to this in response to the conventional DCLP source code (shown earlier) that a human would write.

```
Singleton* Singleton::instance() {
  if (pInstance == 0) {
    Lock lock;
    if (pInstance == 0) {
      pInstance =                       // Step 3
        operator new(sizeof(Singleton)); // Step 1
      new (pInstance) Singleton;        // Step 2
    }
  }
  return pInstance;
}
```

In general, this is not a valid translation of the original DCLP source code, because the Singleton constructor called in step 2 might throw an exception, and if an exception is thrown, it's important that pInstance not yet have been modified. That's why, in general, compilers cannot move step 3 above step 2. However, there are conditions under which this transformation is legitimate. Perhaps the simplest such condition is when a compiler can prove that the Singleton constructor cannot throw (e.g., via post-inlining flow analysis), but that is not the only condition. Some constructors that throw can also have their instructions reordered such that this problem arises.

Given the above translation, consider the following sequence of events:

- Thread *A* enters instance, performs the first test of pInstance, acquires the lock, and executes the statement made up of steps 1 and 3. It is then suspended. At this point pInstance is non-null, but no Singleton object has yet been constructed in the memory pInstance points to.

- Thread *B* enters instance, determines that pInstance is non-null, and returns it to instance's caller. The caller then dereferences the pointer to access the Singleton that, oops, has not yet been constructed.

DCLP will work only if steps 1 and 2 are completed before step 3 is performed, but *there is no way to express this constraint in C or C++.* That's the dagger in the heart of DCLP: we need to define a constraint on relative instruction ordering, but our languages give us no way to express the constraint.

Yes, the C and C++ standards [16, 15] do define *sequence points*, which define constraints on the order of evaluation. For example, paragraph 7 of Section 1.9 of the C++ standard encouragingly states:

> At certain specified points in the execution sequence called sequence points, all side effects of previous evaluations shall be complete and no side effects of subsequent evaluations shall have taken place.

Furthermore, both standards state that a sequence point occurs at the end of each statement. So it seems that if you're just careful with how you sequence your statements, everything falls into place.

Oh, Odysseus, don't let thyself be lured by sirens' voices; for much trouble is waiting for thee and thy mates!

Both standards define correct program behavior in terms of the *observable behavior* of an abstract machine. But not everything about this machine is observable. For example, consider this simple function:

```
void Foo() {
  int x = 0, y = 0;         // Statement 1
  x = 5;                    // Statement 2
  y = 10;                   // Statement 3
  printf("%d,␣%d", x, y);   // Statement 4
}
```

This function looks silly, but it might plausibly be the result of inlining some other functions called by `Foo`.

In both C and C++, the standards guarantee that `Foo` will print *"5,␣10"*, so we know that that will happen. But that's about the extent of what we're guaranteed, hence of what we know. We don't know whether statements 1-3 will be executed at all, and in fact a good optimizer will get rid of them. If statements 1-3 are executed, we know that statement 1 will precede statements 2-4 and—assuming that the call to `printf` isn't inlined and the result further optimized—we know that statement 4 will follow statements 1-3, but we know *nothing* about the relative ordering of statements 2 and 3. Compilers might choose to execute statement 2 first, statement 3 first, or even to execute them both in parallel, assuming the hardware has some way to do it. Which it might well have. Modern processors have a large word size and several execution units. Two or more arithmetic units are common. (For example, the Pentium 4 has three integer ALUs, PowerPC's G4e has four, and Itanium has six.) Their machine language allows compilers to generate code that yields parallel execution of two or more instructions in a single clock cycle.

Optimizing compilers carefully analyze and reorder your code so as to execute as many things at once as possible (within the constraints on observable behavior). Discovering and exploiting such parallelism in regular serial code is the single most important reason for rearranging code and introducing out-of-order execution. But it's not the only reason. Compilers (and linkers) might also reorder instructions to avoid spilling data from a register, to keep the instruction pipeline full, to perform common subexpression elimination, and to reduce the size of the generated executable [4].

When performing these kinds of optimizations, compilers and linkers for C and C++ are constrained only by the dictates of observable behavior on the abstract machines defined by the language standards, and—this is the important bit—those abstract machines are implicitly single threaded. As languages, neither C nor C++ have threads, so compilers don't have to worry about breaking threaded programs when they are optimizing. It should therefore not surprise you that they sometimes do.

That being the case, how can one write C and C++ multithreaded programs that actually work? By using system-specific libraries defined for that purpose. Libraries like Posix threads (pthreads) [6] give precise specifications for the execution semantics of various synchronization primitives. These libraries impose restrictions on the code that library-conformant compilers are permitted to generate, thus forcing such compilers to emit code that respects the execution ordering constraints on which those libraries depend. That's why threading packages have parts written in assembler or issue system calls that are themselves written in assembler (or in some unportable language): you have to go outside standard C and C++ to express the ordering constraints that multithreaded programs require. DCLP tries to get by using only language constructs. That's why DCLP isn't reliable.

As a rule, programmers don't like to be pushed around by their compilers. Perhaps you are such a programmer. If so, you may be tempted to try to outsmart your compiler by adjusting your source code so that `pInstance` remains unchanged until after `Singleton`'s construction is complete. For example, you might try inserting use of a temporary variable:

```
Singleton* Singleton::instance() {
  if (pInstance == 0) {
    Lock lock;
    if (pInstance == 0) {
      Singleton* temp = new Singleton; // initialize to temp
      pInstance = temp;               // assign temp to pInstance
    }
  }
  return pInstance;
}
```

In essence, you've just fired the opening salvo in a war of optimization. Your compiler wants to optimize. You don't want it to, at least not here. But this is not a battle you want to get into. Your foe is wiley and sophisticated, imbued with strategems dreamed up over decades by people who do nothing but think about this kind of thing all day long, day after day, year after year. Unless you write optimizing compilers yourself, they are *way* ahead of you. In this case, for example, it would be a simple matter for the compiler to apply dependence analysis to determine that temp is an unnecessary variable, hence to eliminate it, thus treating your carefully crafted "unoptimizable" code if it had been written in the traditional DCLP manner. Game over. You lose.

If you reach for bigger ammo and try moving temp to a larger scope (say by making it file static), the compiler can still perform the same analysis and come to the same conclusion. Scope, schmope. Game over. You lose.

So you call for backup. You declare temp extern and define it in a separate translation unit, thus preventing your compiler from seeing what you are doing. Alas for you, some compilers have the optimizing equivalent of night-vision goggles: they perform interprocedural analysis, discover your ruse with temp, and again they optimize it out of existence. Remember, these are *optimizing* compilers. They're *supposed* to track down unnecessary code and eliminate it. Game over. You lose.

So you try to disable inlining by defining a helper function in a different file, thus forcing the compiler to assume that the constructor might throw an exception and therefore delay the assignment to pInstance. Nice try, but some build environments perform link-time inlining followed by more code optimizations [5, 11, 4]. GAME OVER. YOU LOSE.

Nothing you do can alter the fundamental problem: you need to be able to specify a constraint on instruction ordering, and your language gives you no way to do it.

5 Almost Famous: The volatile Keyword

The desire for specific instruction ordering makes many wonder whether the volatile keyword might be of help with multithreading in general and with DCLP in particular. In this section, we restrict our attention to the semantics of volatile in C++, and we further restrict our discussion to its impact on DCLP. For a broader discussion of volatile, see the accompanying sidebar.

Section 1.9 of the C++ standard [15] includes this information (emphasis ours):

> The observable behavior of the [C++] abstract machine is its sequence of reads and writes to *volatile* data and calls to library I/O functions.
>
> Accessing an object designated by a *volatile* lvalue, modifying an object, calling a library I/O function, or calling a function that does any of those operations are all *side effects*, which are changes in the state of the execution environment.

In conjunction with our earlier observations that (1) the Standard guarantees that all side effects will have taken place when sequence points are reached and (2) a sequence point occurs at the end of each C++ statement, it would seem that all we need to to do to ensure correct instruction order is to volatile-qualify the appropriate data and to sequence our statements carefully.

Our earlier analysis shows that pInstance needs to be declared volatile, and in fact this point is made in the papers on DCLP[13, 14]. However, Sherlock

Holmes would certainly notice that, in order to ensure correct instruction order, the `Singleton` object *itself* must be also `volatile`. This is *not* noted in the original DCLP papers, and that's an important oversight.

To appreciate how declaring `pInstance` alone `volatile` is insufficient, consider this:

```
class Singleton {
public:
static Singleton* instance();
  ...
private:
  static Singleton* volatile pInstance; // volatile added
  int x;
  Singleton() : x(5) {}
};

// from the implementation file
Singleton* volatile Singleton::pInstance = 0;

Singleton* Singleton::instance() {
  if (pInstance == 0) {
    Lock lock;
    if (pInstance == 0) {
      Singleton* volatile temp = new Singleton; // volatile added
      pInstance = temp;
    }
  }
  return pInstance;
}
```

After inlining the constructor, the code looks like this:

```
if (pInstance == 0) {
  Lock lock;
  if (pInstance == 0) {
    Singleton* volatile temp =
      static_cast<Singleton*>(operator new(sizeof(Singleton)));
    temp->x = 5;           // inlined Singleton constructor
    pInstance = temp;
  }
}
```

Though `temp` is `volatile`, `*temp` is not, and that means that `temp->x` isn't, either. Because we now understand that assignments to non-`volatile` data may sometimes be reordered, it is easy to see that compilers could reorder `temp->x`'s assignment with regard to the assignment to `pInstance`. If they did, `pInstance` would be assigned before the data it pointed to had been initialized, leading again to the possibility that a different thread would read an uninitialized `x`.

An appealing-looking treatment for this disease would be to `volatile`-qualify `*pInstance` as well as `pInstance` itself, yielding a glorified version of `Singleton` where all pawns are painted `volatile`:

```
class Singleton {
public:
static volatile Singleton* volatile instance();
  ...
private:
  // one more volatile added
  static volatile Singleton* volatile pInstance;
};

// from the implementation file
volatile Singleton* volatile Singleton::pInstance = 0;

volatile Singleton* volatile Singleton::instance() {
  if (pInstance == 0) {
    Lock lock;
    if (pInstance == 0) {
      // one more volatile added
      volatile Singleton* volatile temp =
        new volatile Singleton;
      pInstance = temp;
    }
  }
  return pInstance;
}
```

(At this point, one might reasonably wonder why `Lock` isn't also declared `volatile`. After all, it's critical that the lock be initialized before we try to write to `pInstance` or `temp`. Well, `Lock` comes from a threading library, so we can assume it either dictates enough restrictions in its specification or embeds enough magic in its implementation to work without needing `volatile`. This is the case with all threading libraries that we know of. In essence, use of entities (e.g., objects, functions, etc.) from threading libraries leads to the imposition of "hard sequence points" in a program—sequence points that apply to all threads. For purposes of this article, we assume that such "hard sequence points" act as firm barriers to instruction reordering during code optimization: instructions corresponding to source statements preceding use of the library entity in the source code may not be moved after the instructions corresponding to use of the entity, and instructions corresponding to source statements following use of such entities in the source code may not be moved before the instructions corresponding to their use. Real threading libraries impose less draconian restrictions, but the details are not important for purposes of our discussion here.)

One might hope that the above fully `volatile`-qualified code would be guaranteed by the Standard to work correctly in a multithreaded environment, but it may fail for two reasons.

First, the Standard's constraints on observable behavior are only for an abstract machine defined by the Standard, and that abstract machine has no notion of multiple threads of execution. As a result, though the Standard prevents compilers from reordering reads and writes to volatile data *within* a thread, it imposes no constraints at all on such reorderings *across* threads. At least that's how most compiler implementers interpret things. As a result, in practice, many compilers may generate thread-unsafe code from the source above. If your multithreaded code works properly with volatile and doesn't work without, then either your C++ implementation carefully implemented volatile to work with threads (less likely), or you simply got lucky (more likely). Either case, your code is not portable.

Second, just as const-qualified objects don't become const until their constructors have run to completion, volatile-qualified objects become volatile only upon exit from their constructors. In the statement

```
  volatile Singleton* volatile temp = new volatile Singleton;
```

the object being created doesn't become volatile until the expression

```
  new volatile Singleton;
```

has run to completion, and that means that we're back in a situation where instructions for memory allocation and object initialization may be arbitrarily reordered.

This problem is one we can address, albeit somewhat awkwardly. Within the Singleton constructor, we use casts to temporarily add "volatileness" to each data member of the Singleton object as it is initialized, thus preventing relative movement of the instructions performing the initializations. For example, here's the Singleton constructor written in this way. (To simplify the presentation, we've used an assignment to give Singleton::x its first value instead of a member initialization list, as we did in the code above. This change has no effect on any of the issues we're addressing here.)

```
  Singleton()
  {
    static_cast<volatile int&>(x) = 5; // note cast to volatile
  }
```

After inlining this function in the version of Singleton where pInstance is properly volatile-qualified, we get

```
class Singleton {
public:
  static Singleton* instance();
  ...

private:
  static Singleton* volatile pInstance;
  int x;
  ...
};

Singleton* Singleton::instance()
{
  if (pInstance == 0) {
    Lock lock;
    if (pInstance == 0) {
      Singleton* volatile temp =
        static_cast<Singleton*>(operator new(sizeof(Singleton)));
      static_cast<volatile int&>(temp->x) = 5;
      pInstance = temp;
    }
  }
}
```

Now the assignment to x must precede the assignment to pInstance, because both are volatile.

Unfortunately, this all does nothing to address the first problem: C++'s abstract machine is single-threaded, and C++ compilers may choose to generate thread-unsafe code from source like the above, anyway. Otherwise, lost optimization opportunities lead to too big an efficiency hit. After all the discussion, we're back to square one. But wait, there's more. More processors.

6 DCLP on Multiprocessor Machines

Suppose you're on a machine with multiple processors, each of which has its own memory cache, but all of which share a common memory space. Such an architecture needs to define exactly how and when writes performed by one processor propagate to the shared memory and thus become visible to other processors. It is easy to imagine situations where one processor has updated the value of a shared variable in its own cache, but the updated value has not yet been flushed to main memory, much less loaded into the other processors' caches. Such inter-cache inconsistencies in the value of a shared variable is known as the *cache coherency problem*.

Suppose processor A modifies the memory for shared variable x and then later modifies the memory for shared variable y. These new values must be flushed to main memory so that other processors will see them. However, it can be more efficient to flush new cache values in increasing address order, so if y's address precedes x's, it is possible that y's new value will be written to main memory before x's is. If that happens, other processors may see y's value change before x's.

Such a possibility is a serious problem for DCLP. Correct Singleton initialization requires that the Singleton be initialized and that pInstance be updated to be non-null and that *these operations be seen to occur in this order.* If a thread on processor A performs step 1 and then step 2, but a thread on processor B sees step 2 as having been performed before step 1, the thread on processor B may again refer to an uninitialized `Singleton`.

The general solution to cache coherency problems is to use memory barriers (i.e., fences): instructions recognized by compilers, linkers, and other optimizing entities that constrain the kinds of reorderings that may be performed on reads and writes of shared memory in multiprocessor systems. In the case of DCLP, we need to use memory barriers to ensure that `pInstance` isn't seen to be non-null until writes to the `Singleton` have been completed. Here's pseudocode that closely follows an example given in [1]. We show only placeholders for the statements that insert memory barriers, because the actual code is platform-specific (typically in assembler).

```
Singleton* Singleton::instance () {
  Singleton* tmp = pInstance;
  ...                          // insert memory barrier
  if (tmp == 0) {
    Lock lock;
    tmp = pInstance;
    if (tmp == 0) {
      tmp = new Singleton;
      ...                      // insert memory barrier
      pInstance = tmp;
    }
  }
  return tmp;
}
```

Arch Robison (author of [12], but this is from personal communication) points out that this is overkill:

> Technically, you don't need full bidirectional barriers. The first barrier must prevent downwards migration of `Singleton`'s construction (by another thread); the second barrier must prevent upwards migration of `pInstance`'s initialization. These are called ”acquire” and ”release” operations, and may yield better performance than full barriers on hardware (such as Itainum) that makes the distinction.

Still, this is an approach to implementing DCLP that should be reliable, provided you're running on a machine that supports memory barriers. All machines that can reorder writes to shared memory support memory barriers in one form or another. Interestingly, this same approach works just as well in a uniprocessor setting. This is because memory barriers also act as hard sequence points that prevent the kinds of instruction reorderings that can be so troublesome.

7 Conclusions and DCLP Alternatives

There are several lessons to be learned here. First, remember that timeslice-based parallelism on a uniprocessor is not the same as true parallelism across multiple processors. That's why a thread-safe solution for a particular compiler on a uniprocessor architecure may not be thread-safe on a multiprocessor architecture, not even if you stick with the same compiler. (This is a general observation. It's not specific to DCLP.)

Second, though DCLP isn't intrinsically tied to Singleton, use of Singleton tends to lead to a desire to "optimize" thread-safe access via DCLP. You should therefore be sure to avoid implementing Singleton with DCLP. If you (or your clients) are concerned about the cost of locking a synchronization object every time `instance` is called, you can advise clients to minimize such calls by caching the pointer that `instance` returns. For example, suggest that instead of writing code like this,

```
Singleton::instance()->transmogrify();
Singleton::instance()->metamorphose();
Singleton::instance()->transmute();
```

clients do things this way:

```
Singleton* const instance =
  Singleton::instance();          // cache instance pointer

instance->transmogrify();
instance->metamorphose();
instance->transmute();
```

One interesting way to apply this idea is to encourge clients to make a single call to `instance` at the beginning of each thread that needs access to the singleton object, caching the returned pointer in thread-local storage. Code employing this technique thus pays for only a single lock access per thread.

Before recommending caching of the result of calling `instance`, it's generally a good idea to verify that this really leads to a significant performance gain. Use a lock from a threading library to ensure thread-safe Singleton initialization, then do timing studies to see if the cost is truly something worth worrying about.

Third, avoid using a lazily-initialized Singleton unless you really need it. The classic Singleton implementation is based on not initializing a resource until that resource is requested. An alternative is to use eager initialization instead, i.e., to initialize a resource at the beginning of the program run. Because multithreaded programs typically start running as a single thread, this approach can push some object initializations into the single-threaded startup portion of the code, thus eliminating the need to worry about threading during the initialization. In many cases, initializing a singleton resource during single-threaded program startup (e.g., prior to executing `main`) is the simplest way to offer fast, thread-safe singleton access.

A different way to employ eager initialization is to replace use of the Singleton Pattern with the Monostate Pattern [2]. Monostate, however, has different problems, especially when it comes to controlling the order of initialization of the nonlocal static objects that make up its state. *Effective C++* [9] describes these problems and, ironically, suggests using a variant of Singleton to escape them. (The variant is not guaranteed to be thread safe[17].)

Another possibility is to replace a global singleton with one singleton per thread, then use thread-local storage for singleton data. This allows for lazy initialization without worrying about threading issues, but it also means that there may be more than one "singleton" in a multithreaded program.

Finally, DCLP and its problems in C++ and C exemplify the inherent difficulty in writing thread-safe code in a language with no notion of threading (or any other form of concurrency). Multithreading considerations are pervasive, because they affect the very core of code generation. As Peter Buhr pointed out [3], the desire to keep multithreading out of the language and tucked away in libraries is a chimera. Do that, and either (1) the libraries will end up putting constraints on the way compilers generate code (as Pthreads already does) or (2) compilers and other code-generation tools will be prohibited from performing useful optimizations even on single-threaded code. You can pick only two of the troika formed by multithreading, a thread-unaware language, and optimized code generation. Java and the .NET CLI, for example, address the tension by introducing thread-awareness into the language and language infrastructure, respectively [8, 12].

8 Acknowledgements

Pre-publication drafts of this article were reviewed by Doug Lea, Kevlin Henney, Doug Schmidt, Chuck Allison, Petru Marginean, Hendrik Schober, David Brownell, Arch Robison, Bruce Leasure, and James Kanze. Their comments, insights, and explanations greatly improved the presentation of the paper and led us to our current understanding of DCLP, multithreading, instruction ordering, and compiler optimizations. After publication, we incorporated comments by Fedor Pikus, Al Stevens, Herb Sutter, and John Hicken.

9 About the Authors

Scott Meyers has written three *Effective C++* books and is consulting editor for the Addison-Wesley Effective Software Development Series. His current interests focus on identifying fundamental principles for improving software quality. His web site is `http://aristeia.com`.

Andrei Alexandrescu is the author of *Modern C++ Design* and of numerous articles, most of which were written as a *CUJ* columnist. He pursues a Ph.D. degree at University of Washington, specializing in programming languages. His web site is `http://moderncppdesign.com`.

10 [Sidebar] volatile: A Brief History

To find the roots of volatile, let's go back to the 1970s, when Gordon Bell (of PDP-11 fame) introduced the concept of *memory-mapped I/O* (MMIO). Before that, processors allocated pins and defined special instructions for performing port I/O. The idea behind MMIO is to use the same pins and instructions for both memory and port access. Hardware outside the processor intercepts specific memory addresses and transform them into I/O requests; so dealing with ports became simply reading from and writing to machine-specific memory addresses.

What a great idea. Reducing pin count is good—pins slow down signal, increase defect rate, and complicate packaging. Also, MMIO doesn't require special instructions for ports. Programs just use the memory, and the hardware takes care of the rest.

Or almost.

To see why MMIO needs volatile variables, let's consider the following code:

```
unsigned int *p = GetMagicAddress();
unsigned int a, b;
a = *p;
b = *p;
```

If p refers to a port, a and b should receive two consecutive words read from that port. However, if p points to a bona fide memory location, then a and b load the same location twice and hence will compare equal. Compilers exploit this assumption in the *copy propagation* optimization that transforms the last line above into the more efficient:

```
b = a;
```

Similarly, for the same p, a, and b, consider:

```
*p = a;
*p = b;
```

The code writes two words to *p, but the optimizer might assume that *p is memory and perform the *dead assignment elimination* optimization by eliminating the first assignment. Clearly, this "optimization" would break the code. A similar situation can arise when a variable is modified by both mainline code and an interrupt service routine (ISR). What might appear to a compiler to be a redundant read or write might actually be necessary in order for the mainline code to communicate with an ISR.

So when dealing with some memory locations (e.g. memory mapped ports or memory referenced by ISRs), some optimizations must be suspended. volatile exists for specifying special treatment for such locations, specifically: (1) the content of a volatile variable is "unstable" (can change by means unknown to the compiler), (2) all writes to volatile data are "observable" so they must be executed religiously, and (3) all operations on volatile data are executed in the sequence in which they appear in the source code. The first two rules ensure proper reading and writing. The last one allows implementation of I/O protocols that mix input and output. This is informally what C and C++'s `volatile` guarantees.

Java took `volatile` a step further by guaranteeing the properties above across multiple threads. This was a very important step, but it wasn't enough to make `volatile` usable for thread synchronization: the relative ordering of `volatile` and non-`volatile` operations remained unspecified. This ommission forces many variables to be `volatile` to ensure proper ordering.

Java 1.5's `volatile` [10] has the more restrictive, but simpler, *acquire/release semantics*: any read of a `volatile` is guaranteed to occur *prior* to any memory reference (`volatile` or not) in the statements that follow, and any write to a `volatile` is guaranteed to occur *after* all memory references in the statements preceding it. .NET defines `volatile` to incorporate multithreaded semantics as well, which are very similar to the currently proposed Java semantics. We know of no similar work being done on C's or C++'s `volatile`.

References

[1] David Bacon, Joshua Bloch, Jeff Bogda, Cliff Click, Paul Hahr, Doug Lea, Tom May, Jan-Willem Maessen, John D. Mitchell, Kelvin Nilsen, Bill Pugh, and Emin Gun Sirer. The "Double-Checked Locking Pattern is Broken" Declaration. Available at `http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html`.

[2] Steve Ball and John Crawford. Monostate Classes: The Power of One. *C++ Report*, May 1997. Reprinted in *More C++ Gems*, Robert C. Martin, ed., Cambridge University Press, 2000.

[3] Peter A. Buhr. Are Safe Concurrency Libraries Possible? *Communications of the ACM*, 38(2):117–120, 1995. Available at `http://citeseer.nj.nec.com/buhr95are.html`.

[4] Bruno De Bus, Daniel Kaestner, Dominique Chanet, Ludo Van Put, and Bjorn De Sutter. Post-pass Compaction Techniques. *Communications of the ACM*, 46(8):41–46, August 2003. Available at `http://doi.acm.org/10.1145/859670.859696`.

[5] Robert Cohn, David Goodwin, P. Geoffrey Lowney, and Norman Rubin. Spike: An Optimizer for Alpha/NT Executables. Available at `http://www.usenix.org/publications/library/proceedings/usenix-nt97/presentations/goodwin/index.htm`, August 1997.

[6] IEEE Standard for Information Technology. *Portable Operating System Interface (POSIX) — System Application Program Interface (API) Amendment 2: Threads Extension (C Language)*. ANSI/IEEE 1003.1c-1995, 1995.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. Also available as *Design Patterns CD*, Addison-Wesley, 1998.

[8] Doug Lea. *Concurrent Programming in Java*^{TM}. Addison-Wesley, 1999. Excerpts relevant to this article can be found at `http://gee.cs.oswego.edu/dl/cpj/jmm.html`.

[9] Scott Meyers. *Effective C++, Second Edition*. Addison-Wesley, 1998. Item 47 discusses the initialization problems that can arise when using non-local static objects in C++.

[10] Sun Microsystems. J2SE 1.5.0 Beta 1. February 2004. `http://java.sun.com/j2se/1.5.0/index.jsp`; see `http://jcp.org/en/jsr/detail?id=133` for details on the changes brought to Java's memory model.

[11] Matt Pietrek. Link-Time Code Generation. *MSDN Magazine*, May 2002. Available at `http://msdn.microsoft.com/msdnmag/issues/02/05/Hood/`.

[12] Arch D. Robison. Memory Consistency & .NET. *Dr. Dobb's Journal*, April 2003.

[13] Douglas C. Schmidt and Tim Harrison. Double-Checked Locking. In Robert Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern Languages of Program Design 3*, chapter 20. Addison-Wesley, 1998. Available at `http://www.cs.wustl.edu/~schmidt/PDF/DC-Locking.pdf`.

[14] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture, Volume 2*. Wiley, 2000. Tutorial notes based on the patterns in this book are available at `http://cs.wustl.edu/~schmidt/posa2.ppt`.

[15] ISO/IEC 14882:1998(E) International Standard. *Programming languages — C++*. ISO/IEC, 1998.

[16] ISO/IEC 9899:1999 International Standard. *Programming languages — C*. ISO/IEC, 1999.

[17] John Vlissides. *Pattern Hatching: Design Patterns Applied*. Addison-Wesley, 1998. The discussion of the "Meyers Singleton" is on pp. 69ff.