Further Reading

- For an in-depth discussion of the difficulties of implementing double-checked locking in C++03, see **meyers04a** and **meyers04b**.
- For a discussion of the Singleton pattern and a variety of implementations in C++03, see Chapter 6 of **alexandrescu01**.

Appendix

C++03 double-checked-lock pattern

Prior to the introduction of the function-scope **static** object initialization guarantees discussed in *Description* on page 68, preventing multiple initializations of **static** objects and use before initialization of those same objects was still needed. Guarding access using a `mutex` was often a significant performance cost, so using the unreliable, double-checked-lock pattern was often attempted to avoid the overhead:

```
Logger& getInstance()
{
    static Logger* volatile loggerPtr = 0;  // hack, used to simulate atomics

    if (!loggerPtr)  // Does the logger need to be initialized?
    {
        static ~~std::mutex~~ m;
        ~~std::lock_guard<std::mutex>~~ guard(m);  // Lock the mutex.

        if (!loggerPtr)  // We are first, as the logger is still uninitialized.
        {
            static Logger logger("log.txt");
            loggerPtr = &logger;
        }
    }                    // Either way, the lock guard unlocks the mutex here.

    return *loggerPtr;
}
```

```
    xor eax, eax   ; zero out 'eax' register
    ret            ; return from 'main'
```

A sufficiently smart compiler might, however, not generate synchronization code in a single-threaded context or else provide a flag to control this behavior.

In this example, we are using a **`volatile`** pointer as a partial substitute for an atomic variable, a nonportable solution that is not correct in standard C++ but has historically been moderately effective. The C++11 Standard Library does, however, provide the `<atomic>` header, which is a far superior alternative, and many implementations have historically provided extensions to support atomic types even prior to C++11. Where available, compiler extensions are typically preferable over home-grown solutions.

In addition to being difficult to write, this decidedly complex workaround would often prove unreliable. The problem is that, even though the logic appears sound, architectural changes in widely used CPUs allowed for the CPU itself to optimize and reorder the sequence of instructions. Without additional support, the hardware would not see the dependency that the second test of `loggerPtr` has on the locking behavior of the mutex and would do the read of `loggedPtr` prior to acquiring the lock. This reordering of instructions would then allow multiple threads to acquire the lock while each thinking the **`static`** variable still needs to be initialized.

To solve this subtle issue, concurrency library authors are expected to issue ordering hints, such as **fences** and **barriers**. A well-implemented threading library would provide atomics equivalent to the modern `std::atomic` that would issue the correct instructions when accessed and modified. The C++11 Standard makes the compiler aware of these concerns and provides portable *atomics* and support for threading that enables users to handle such issues correctly. The above `getInstance` function could be corrected by changing the type of `loggerPtr` to `std::atomic<Logger*>`. Prior to C++11, despite being complicated, the same function would reliably implement the Meyers Singleton (see *Use Cases — Meyers Singleton* on page 71) in C++03 on contemporary hardware.