that of the file manager. But suppose that `FileManager` didn't always log at construction and was created before anything else logged. In that case, we have no reason to think that the logger would be around for the `FileManager` to log during its destruction after `main`.

In the case of low-level, widely used facilities, such as a logger, a conventional Meyers Singleton is contraindicated. The two most common alternatives discussed at the end of *Use Cases — Meyers Singleton* on page 71 involve never ending the lifetime of the mechanism at all. It is worth noting that ~~truly~~ global objects — such as `cout`, `cerr`, and `clog` — from the Standard `iostream` Library are typically not implemented using conventional methods and are in fact treated specially by the runtime system.

Annoyances

Overhead in single-threaded applications

A single-threaded application invoking a function containing a function-scope static storage duration variable might have unnecessary synchronization overhead, such as an **atomic** load operation. For example, consider a program that accesses a simple Meyers Singleton for a user-defined type with a **user-provided** default constructor:

```
struct S  // user-defined type
{
    S() { }  // inline default constructor
};

S& getS()  // free function returning local object
{
    static S local;  // function-scope local object
    return local;
}

int main()
{
    getS();    // Initialize the file-scope static singleton.
    return 0;  // successful status
}
```

Although it is clearly visible to the compiler that `getS()` is invoked by only one thread, the generated assembly instructions might still contain atomic operations or other forms of synchronization, and the call to `getS()` might not be inlined.[7]

[7]Both GCC 11.2 (c. 2021) and Clang 12.0.1 (c. 2021), using the `-Ofast` optimization level, generate assembly instructions for a **memory barrier** and fail to inline the call to `getS`. Using `-fno-threadsafe-statics` reduces considerably the number of operations performed but still does not lead to the compilers' inlining of the function call. Both popular compilers will, however, reduce the program to just two x86 assembly instructions if the user-provided constructor of `S` is either removed or defaulted (see Section 1.1."Defaulted Functions" on page 33); doing so will turn `S` into a **trivially-constructible** type, implying that no code needs to be executed during initialization: