As discussed in *Description* on page 68, the augmentation of a thread-safety guarantee for the runtime initialization of function-scope **`static`** objects in C++11 minimizes the effort required to create a thread-safe singleton. Note that, prior to C++11, the simple function-scope **`static`** implementation would not be safe if concurrent threads were trying to initialize the logger; see *Appendix — C++03 double-checked-lock pattern* on page 81.

The Meyers Singleton is also seen in a slightly different form where the singleton type's constructor is made **`private`** to prevent more than just the one singleton object from being created:

```
class Logger
{
private:
    Logger(const char* logFilePath);  // configures the singleton
    ~Logger();                        // suppresses copy construction too

public:
    static Logger& getInstance()
    {
        static Logger localLogger("log.txt");
        return localLogger;
    }
};
```

This variant of the function-scope-**`static`** singleton pattern prevents users from manually creating rogue `Logger` objects; the only way to get one is to invoke the logger's **`static`** `Logger::getInstance()` member function:

```
void client()
{
    Logger::getInstance() << "Hi";  // OK
    Logger myLogger("myLog.txt");   // Error, Logger constructor is private.
}
```

This formulation of the singleton pattern, however, conflates the type of the singleton object with its use and purpose as a singleton. Once we find a use of a singleton object, finding another and perhaps even a third is not uncommon.

Consider, for example, an application on an early model of mobile phone where we want to refer to the phone's camera. Let's presume that a `Camera` class is a fairly involved and sophisticated mechanism. Initially we use the variant of the Meyers Singleton pattern where at most one `Camera` object can be present in the entire program. The next generation of the phone, however, turns out to have more than one camera, say, a front `Camera` and a back `Camera`. Our brittle design doesn't admit the dual-singleton use of the same fundamental `Camera` type. A more finely factored solution would be to implement the `Camera` type separately and then to provide a thin wrapper, e.g., perhaps using the **`strong-typedef`**