**idiom** (see Section 2.1."Inheriting ~~Ctors~~" on page 535), corresponding to each singleton use:

```
class PrimaryCamera
{
private:
    Camera& d_camera_r;
    PrimaryCamera(Camera& camera)  // implicit constructor
      : d_camera_r(camera) { }

public:
    static PrimaryCamera getInstance()
    {
        static Camera localCamera{/*...*/};
        return localCamera;
    }
};
```

With this design, adding a second and even a third singleton that is able to reuse the underlying `Camera` mechanism is facilitated.

Although this function-scope-**static** approach provides stronger guarantees than the file-scope-**static** one, it does have its limitations. In particular, when one global facility object, such as a logger, is used in the destructor of another function-scope static object, the logger object might possibly have already been destroyed when it is used.[3] One approach is to construct the logger object by explicitly allocating it and never deleting it:

```
Logger& getLogger()
{
    static Logger& l = *new Logger("log.txt");  // dynamically allocated
    return l;  // Return a reference to the logger (on the heap).
}
```

A distinct advantage of this approach is that once an object is created, it *never* goes away before the process ends. The disadvantage is that, for many classic and current profiling tools (e.g., *Purify*, *Coverity*), this intentionally never-freed dynamic allocation is indistinguishable from a **memory leak**. The ultimate workaround is to create the object itself in **static** memory, in an appropriately sized and aligned region of memory:

```
#include <new>  // placement new

Logger& getLogger()
{
    static std::aligned_storage<sizeof(Logger), alignof(Logger)>::type buf;
    static Logger& logger = *new(&buf) Logger("log.txt");  // allocate in place
    return logger;
}
```

[3] An amusing workaround, the so-called *Phoenix Singleton*, is proposed in **alexandrescu01**, section 6.6, "Addressing the Dead Reference Problem (I): The Phoenix Singleton," pp. 137–139.