Note that any memory that the `Logger` itself manages would still come from the global heap and be recognized as memory leaks.[4]

In this final incarnation of a decidedly non-Meyers-Singleton pattern, we first reserve a block of memory of sufficient size and the correct alignment for `Logger` using `std::aligned_storage`. Next we use that storage in conjunction with placement **`new`** to create the logger directly in that static memory. Notice that this allocation is not from the dynamic store, so typical profiling tools will not track and will not provide a false warning when we fail to destroy this object at program termination time. Now we can return a reference to the logger object embedded safely in static memory knowing that it will be there until application exit.

Potential Pitfalls

`static`-storage-duration objects ~~are not guaranteed to be initialized~~

Despite C++11's guarantee that each individual function-scope **`static`** initialization will occur at most once and before control can reach a point where the variable can be referenced, no analogous guarantees are made of nonlocal objects of static storage duration. Absence of this guarantee makes any interdependency in the initialization of such objects, especially across translation units (TUs), an abundant source of insidious errors.

Objects that undergo **constant initialization** have no such issue: Such objects will never be accessible at run time before having their initial values. Objects that are not constant initialized[5] will instead be **zero initialized** until their constructors run, which itself might lead to undefined behavior that is not necessarily conspicuous.

As a demonstration of what can happen when we depend on the relative order of initialization of variables at file or namespace scope used before `main`, consider the **cyclically dependent** pair of source files, `a.cpp` and `b.cpp`:

```
// a.cpp:
extern int setB(int);  // declaration only of setter in other TU
int *a = new int;      // runtime initialization of file-scope variable
int setA(int i)        // Initialize a; then b.
{
    *a = i;            // Populate the allocated heap memory.
    setB(i);           // Invoke setter to populate the other one.
    return 0;          // Return successful status.
}
```

[4] If the global heap is to be entirely avoided, we could leverage a polymorphic-allocator implementation such as `std::pmr` in C++17. We would first create a fixed-size array of memory having static storage duration. Then we would create a **`static`** memory-allocation mechanism, e.g., `std::pmr::monotonic_buffer_resource`. Next we would use placement **`new`** to construct the logger within the static memory pool using our static allocation mechanism and supply that same mechanism to the `Logger` object so that it could get all its internal memory from that static pool as well; a discussion of this topic is planned for **lakos22**.

[5] C++20 added a new keyword, **`constinit`**, that can be placed on a variable declaration to *require* that the variable in question undergo constant initialization and thus can never be accessed at run time prior to the start of its lifetime.