The interface in the `libcomp.h` file comprises the definition of `S` along with the declaration of an accessor function, `getGlobalS`. Code outside the `libcomp.cpp` file can access the singleton object `globalS` only by calling the free function `getGlobalS()`. Now consider the `main.cpp` file in the example below, which implements `main` and also makes use of `globalS` prior to entering `main`:

```
// main.cpp:
#include <cassert>    // standard C assert macro
#include <libcomp.h>  // getGlobalS()

bool globalInitFlag = getGlobalS().isInitialized();

int main()
{
    assert(globalInitFlag);   // Bug, or at least potentially so
    return 0;
}
```

Depending on the compiler or the link line, the call initializing `globalInitFlag` might occur and return *prior* to the initialization of `globalS`. C++ does not guarantee that objects at file or namespace scope in separate translation units will be initialized just because a function located within that translation unit happens to be called.

An effective pattern for helping to ensure that a nonlocal object *is* initialized before it is used from a separate translation unit — especially when that use might occur prior to entering `main` — is simply to move the **`static`** object from file or namespace scope to the scope of the function accessing it, making it a function-scope **`static`** instead:

```
S& getGlobalS()  // access into this translation unit
{
    static S globalS;  // singleton is now function-scope static
    return globalS;
}
```

Commonly known as the **Meyers Singleton** for author Scott Meyers who popularized it, this pattern ensures that the singleton object will *necessarily* be initialized on the first call to the accessor function that envelopes it, irrespective of when and where that call is made. Moreover, that singleton object will also live past the end of `main`. The Meyers Singleton pattern also gives us a chance to catch and respond to exceptions thrown when constructing the **`static`** object, rather than immediately terminating the program, as would be the case if declared as a **`static`** global variable. Much more importantly, however, since C++11, the Meyers Singleton pattern automatically inherits the benefits of effortless race-free initialization of *reusable* program-wide singleton objects. The Meyers Singleton can be safely used both in the programs where the singleton initialization might happen before `main` and those where it might happen after additional threads have already been started.