Chapter 1

Safe Features

ch-safe
sec-safe-cpp11 Intro text should be here.

Threadsafe Function-Scope `static` Variables

tion-static-variables

Function-scope **static** objects are now guaranteed to be initialized free of race conditions in the presence of multiple concurrent threads.

Description

iption-functionstatic

When a variable is declared within the body of a function, we say that the variable is declared at **function scope** (a.k.a. **local scope**). A variable not marked **static** is known as an **automatic** variable, with a distinct instance on the **program stack** for each function invocation. **Automatic** objects are allocated on the stack and initialized whenever the **flow of control** passes through the **definition** of that object. An object (e.g., `iLocal`) that is declared **static** within the body of a function (e.g., `f`) will instead be allocated once per program, and will initialized only the first time the **flow of control** passes through the **definition** of that object:

```
#include <cassert>  // standard C assert macro

int f(int i) // function returning the first argument with which it is called
{
    static int iLocal = i;  // object initialized once only, on the first call
    return iLocal;          // the same iLocal value is returned on every call
}

int main()
{
    int a = f(10);  assert(a == 10);  // Initialize and return iLocal.
    int b = f(20);  assert(b == 10);  // Return iLocal.
    int c = f(30);  assert(c == 10);  // Return iLocal.

    return 0;
}
```

In the simple example above, the function, `f`, initializes its **static** object, `iLocal`, with its argument, `i`, only the first time it is called and then always returns the same value (e.g., 10). Hence, when that function is called repeatedly with distinct arguments while initializing, respectively, `a`, `b`, and `c`, all three of these variables are initialized to the same value, 10, used the first time `f` was invoked (i.e., to initialize `a`). Although the function-scope **static** object, `iLocal`, was created after `main` was entered, it will not be destroyed until after `main` exits.

Concurrent Initialization

urrent-initialization

Historically, initialization of **function-scope static storage-duration** objects was not guaranteed to be safe in a **multithreading context** because it was subject to **data races** if the function was called concurrently from multiple threads. These **data races** around initialization can lead to the initializer being invoked multiple times, object construction running concurrently on the same object, and control flow continuing past the variable definition before initialization had completed at all. All of these variations would result in critical software flaws. One common but unreliable pre-C++11 workaround was the *double-checked lock pattern*; see *Appendix: C++03 Double-Checked Lock Pattern* on page 15.

As of C++11, a conforming compiler is now required to ensure that initialization of **function-scope `static` storage-duration** objects is performed safely, and exactly once, before execution continues past the initializer, even when the function is called concurrently from multiple threads.

destruction

Destruction

romeoglossAutomatic objects within a local scope are destroyed when control leaves the scope in which they are declared On the other hand, **`static`** local objects that have been initialized are not destroyed until normal program termination, either after the `main` function returns normally or when the `std::exit` function is called. The order of destruction of these objects will be the reverse of the order in which they completed construction. Note that programs can terminate in several other ways, such as a call to `std::quick_exit`, `_Exit`, or `std::abort`, that explicitly do *not* destroy **`static` storage-duration** objects.

logger-example

Logger example

Let’s now consider a real-world example in which a single object — e.g., `localLogger` in the example below — is used widely throughout a program (see also *Use Cases — Meyers Singleton* on page 6):

```
Logger& getLogger()  // ubiquitous pattern commonly known as "Meyers Singleton"
{
    static Logger localLogger("log.txt");  // function-local static definition
    return localLogger;
}

int main()
{
    getLogger() << "hello";
        // OK, invokes Loggers constructor for the first (and only) time

    getLogger() << "world";
        // OK, uses the previously constructed Logger instance
}
```

Here we hae an example of the “Singleton pattern”[1] being used to create the shared `Logger` instance and provide access to it through the `getLogger()` function. The **`static`** local instance of `Logger`, `localLogger`, will be initialized exactly once and then destroyed after normal program termination. In C++03 this accessor function would be safe to call before `main`, but *not* concurrently. C++11 guarantees that the initialization of `localLogger` will still happen exactly once even when multiple threads call `getLogger` concurrently.
In a large-scale production environment, we would avoid evaluating any expression whose result is intended to be logged unless the logging level for that specific logging statement is

[1] **?**, Chapter 3, section “Singleton”, pp.-127–???

enabled.[2]) All function-local **static** objects, such as `localLogger` in the example above, will be destroyed automatically only on normal program termination,

ultithreaded-contexts

Multithreaded contexts

To illustrate how defects might have been introduced by multithreading *prior* to C++11, suppose that we have a simple type, `MyString`, that always allocates dynamic memory on construction:

```
#include <cstring>  // std::size_t, std::memcpy, std::strlen

class MyString
{
    char* d_string_p;  // pointer holding dynamically allocated memory address

public:
    MyString(const char* s)                                  // (1)
    {                                                        // (2)
        const std::size_t size = std::strlen(s) + 1;         // (3)
        d_string_p = static_cast<char*>(::operator new(size));  // (4)
        std::memcpy(d_string_p, s, size);                    // (5)
    }                                                        // (6)
};
```

Let's say that we want to create a **static** object of this `MyString` class in a function, `f`, that might be invoked concurrently from multiple threads:

```
void f()
{
    static const MyString str("s");  // function-scope, static storage-duration
    // ...
}
```

Let's now imagine that `f` is called from two separate threads concurrently, without having been called before. Suppose that the first thread gets through the `MyString` constructor, in the example above, up to *but not including* line (4) before it is suspended by the operating system. After that, a second thread might begin initializing `str` again — because no guards existed prior to C++11 to prevent this — and make it all the way past line (6) before it too is suspended. When the operating system eventually resumes execution of the first thread, the dynamic allocation and assignment on line (4) **leaks** the memory for the previously constructed `MyString`.

In practice, however, **undefined behavior** (prior to C++11) might have manifested even earlier. When the second thread re-uses the storage claimed by the object in the first thread, it effectively ends the lifetime of one **static** `S` object to start the lifetime of the other one. After that, any attempt to access the original `s` object would be **undefined behavior**, because its lifetime has ended, even though its destructor did not run.

The C++11 Standard Library provides copious utilities and abstractions related to multithreading. One part of that, `std::thread`, is a portable wrapper for a platform-specific

[2]An eminently useful, full-featured logger, known as the `ball` logger, can be found in the `ball` package of the `bal` package group of Bloomberg's open-source BDE libraries (**?**, subdirectory `/groups/bal/ball`).

thread handle provided by the operating system. When constructing an `std::thread` object with a **callable object** a new thread invoking that **callable object** will be spawned. Prior to destroying such `std::thread` objects it is also neccessary to invoke the `join` member function on the thread object, which will block until the background thread of execution completes invoking its **callable object**.

This threading facility from the standard library can be used with our earlier `Logger` example from *Description — Logger example* on page 3, to concurrently attempt to access the `getLogger` function:

```
#include <thread>  // std::thread

void useLogger() { getLogger() << "example"; }  // concurrently called function

int main()
{
    std::thread t0(&useLogger);
    std::thread t1(&useLogger);
        // Spawn two new threads, each of which invokes useLogger.

    // ...

    t0.join();  // Wait for t0 to complete execution.
    t1.join();  // Wait for t1 to complete execution.

    return 0;
}
```

Such use prior to the C++11 thread-safety guarantees (with pre-C++11 threading libraries) could have led to a race condition during the initialization of `localLogger`, which was defined as a local **`static`** object in `getLogger`:
As of C++11, the example above has no data races provided that `Logger::operator<<(const char*)` is designed properly for multithreaded use, even though the `Logger::Logger(const char* logFilePath)` constructor (i.e., the one used to configure the singleton instance of the logger) is not. That is to say, the implicit **critical section** that is guarded by the compiler includes evaluation of the initializer, which is why a recursive call to initialize a function-scope **`static`** variable is undefined behavior and is likely to result in deadlock; see *Dangerous Recursive Initialization* on page 11. Such use of function-scope **`static`**s, however, is not foolproof; see *Potential Pitfalls — Depending on order-of-destruction of local objects after* `main` *returns* on page 12.

The destruction of **function-scope `static`** objects is and always has been guaranteed to be safe *provided* (1) no threads are running after returning from `main` and (2) **function-scope `static`** objects do not depend on each other during destruction; see *Potential Pitfalls — Depending on order-of-destruction of local objects after* `main` *returns* on page 12.

-cases-functionstatic

Use Cases

meyers-singleton

Meyers Singleton

The guarantees surrounding access across **translation units** to runtime initialized objects at file or namespace scope are few and dubious — especially when that access might occur prior to entering `main`. Consider a library component, `libcomp`, that defines a file-scope **static** singleton, `globalS`, that is initialized at run time:

```
// libcomp.h:
#ifndef INCLUDED_LIBCOMP
#define INCLUDED_LIBCOMP

struct S { /*... */ };
S& getGlobalS();  // access to global singleton object of type S

#endif

// libcomp.cpp:
#include <libcomp.h>

static S globalS;
S& getGlobalS() { return globalS; }  // access into this translation unit
```

The interface in the `libcomp.h` file comprises the definition of `S` along with the declaration of an accessor function, `getGlobalS`. Any function wishing to access the singleton `globalS` object sequestered within the `libcomp.cpp` file would *presumably* do so safely via the global `getGlobalS()` accessor function. 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` may 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 non-local 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 at file or namespace scope inside 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 the 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, 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 whose first invocation might be before `main` in some programs and after additional threads have already been started after entering `main` in other programs.

As discussed in *Description* on page 2, 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 vying to initialize the logger; see *Appendix: C++03 Double-Checked Lock Pattern* on page 15.

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);  // Configure 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, *ToasterToothbrush*-like[3] 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 idiom** (see Section 1.1."**??**" on page **??**), 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 is vastly superior to 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 may possibly have already been destroyed when it is used.[4] 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, 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

[3]See **?**, section 0.3, pp. 13–20, specifically Figure 0-9, p. 16.
[4]An amusing workaround, the so-called *Phoenix Singleton*, is proposed in **?**, section 6.6, pp. 137–139.

a **memory leak**. The ultimate workaround is to create the object itself in **static** memory, in an appropriately sized and aligned region of memory[5]:

```
#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;
}
```

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 for all eternity.

itfalls-functionstatic

Potential Pitfalls

nteed-to-be-initialized

SubsubsecCode 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, almost no similar guarantees are made of non-local objects of **static storage-duration** objects. This makes any interdependencies in the initialization of such objects, especially across **translation units** (TUs), an abundant source of insidious errors.

Objects that undergo **constant initialization** have no issue: such objects will never be accessible at run time before having their initial values. Objects that are not constant initialized[6] will instead be **zero initialized** until their constructors run, which itself might lead to conspicuous (or perhaps latent) undefined behavior.

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:
```

[5]Note that any memory that `Logger` itself manages would still come from the global heap and be recognized as memory leaks. If available, 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; see **?**.

[6]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.

```
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 variable; then b.
{
    *a = i;            // Populate the allocated heap memory.
    setB(i);           // Invoke setter to populate the other one.
    return 0;          // Return successful status.
}

// b.cpp:
int *b = new int;      // runtime initialization of file-scope variable
int setB(int i)        // Initialize b
{
    *b = i;            // Populate the allocated heap memory.
    return 0;          // Return successful status.
}

extern int setA(int);  // declaration (only) of setter in other TU
int x = setA(5);       // Initialize a and b.
int main()             // main program entry point
{
    return 0;          // Return successful status.
}
```

These two **translation units** will be initialized before `main` is entered in some order, but — regardless of that order — the program in the example above will wind up dereferencing a null pointer before entering `main`:

```
$ g++ a.cpp b.cpp main.cpp
$ ./a.out
  Segmentation fault (core dumped)
```

Suppose we were to instead move the file-scope **static** pointers, corresponding to both `setA` and `setB`, inside their respective function bodies:

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

// b.cpp: (make analagous changes)
```

Now the program reliably executes without incident:

```
$ g++ a.cpp b.cpp main.cpp
$ ./a.out
$
```

In other words, even though no order exists in which the **translation units** as a whole could have been initialized prior to entering `main` such that the *file*-scope variables would be valid before they were used, by instead making them *function*-scope **`static`**, we are able to guarantee that each variable is itself initialized before it is used, regardless of translation-unit-initialization order.

While on the surface it may seem as though local and non-local objects of **`static`** storage duration are effectively interchangeable, this is clearly not the case. Even when clients cannot directly access the non-local object due to giving it **internal linkage** by marking it **`static`** or putting it in an **unnamed namespace**, the initialization behaviors make such objects behave very differently.

ecursive-initialization

Dangerous Recursive Initialization

As with all other initialization, control flow does not continue *past* the **definition** of a **`static`** local object until after the initialization is complete, making recursive **`static`** initialization — or any initializer that might eventually call back to the same function — dangerous:

```
int fz(int i)  // The behavior is undefined unless i is 0.
{
    static int dz = i ? fz(i - 1) : 0;  // Initialize recursively. (BAD IDEA)
    return dz;
}

int main()  // The program is ill-formed.
{
    int x = fz(5);  // Bug, e.g., due to possible deadlock
}
```

In the ill-fated example above, the second recursive call of `fz` to initialize `dz` has undefined behavior because the control flow reached the same definition again before the initialization of the **`static`** object was completed; hence, control flow cannot continue to the **`return`** statement in `fz`. Given a likely implementation with a nonrecursive mutex or similar lock, the program can potentially deadlock, though many implementations provide better diagnostics with an exception or assertion violation when this form of error is encountered. [7]

otleties-with-recursion

Subtleties with Recursion

Even when not recursing within the initializer itself, the rule for the initialization of **`static`** objects at function scope becomes more subtle for self-recursive functions. Notably, the initialization happens based on when flow of control first passes the variable definition and *not* based on the first invocation of the containing function function. Due to this, when a

[7]Prior to standardization (see **?**, section 6.7, p. 92), C++ allowed control to flow past a **`static`** function-scope variable even during a recursive call made as part of the initialization of that variable. This would result in the rest of such a function executing with a zero-initialized and possibly partially constructed local object. Even modern compilers, such as GCC with `-fno-threadsafe-statics`, allow turning off the locking and protection from concurrent initialization and retaining some of the pre-C++98 behavior. This optional behavior is, however, fraught with peril and unsupported in any standard version of C++.

recursive call happens in relation to the definition of a **static** local variable impacts which values might be used for the initialization:

```
    assert

int fx(int i)  // self-recursive after creating function-static variable, dx
{
    static int dx = i;     // Create dx first.
    if (i) { fx(i - 1); }  // Recurse second.
    return dx;             // Return dx third.
}

int fy(int i)  // self-recursive before creating function-static variable,dy
{
    if (i) { fy(i - 1); }  // Recurse first.
    static int dy = i;     // Create dy second.
    return dy;             // Return dy third.
}

int main()
{
    int x = fx(5);  assert(x == 5);  // dx is initialized before recursion.
    int y = fy(5);  assert(y == 0);  // dy is initialized after recursion.
    return 0;
}
```

If the self-recursion takes place *after* the **static** variable is initialized (e.g., `fx` in the example above), then the **static** object (e.g., `dx`) is initialized on the *first* recursive call; if the recursion occurs *before* (e.g., `fy` in the example above), the initialization (e.g., of `dy`) occurs on the *last* recursive call.

Depending on order-of-destruction of local objects after `main` returns

ts-after-main-returns

Within any given **translation unit**, the relative order of initialization of objects at file or namespace scope having **static storage duration** is well defined and predictable. As soon as we have a way to reference an object outside of the current **translation unit**, before `main` is entered, we are at risk of using the object before it has been initialized. Provided the initialization itself is not cyclic in nature, we can make use of function-scope **static** objects (see *Use Cases — Meyers Singleton* on page 6) to ensure that no such uninitialized use occurs, even across **translation units** before `main` is entered. The relative order of destruction of such function-scope **static** variables — even when they reside within the same **translation unit** — is not clearly known at compile time, as it will be the reverse of the order in which they are initialized, and reliance on such order can easily lead to **undefined behavior** in practice.

This specific problem occurs when a **static** object at file, namespace, or function scope uses (or might use) in its destructor another **static** object that is either (1) at file or namespace scope and resides in a separate **translation unit** or (2) any other function-scope **static** object (i.e., including one in the same **translation unit**). For example, suppose we have implemented a low-level logging facility as a Meyers Singleton:

```
Logger& getLogger()
{
    static Logger local("log.txt");
    return local;
}
```

Now suppose we implement a higher-level file-manager type that depends on the function-scope **static** logger object:

```
struct FileManager
{
    FileManager()
    {
         getLogger() << "Starting up file manager...";
         // ...
    }

    ~FileManager()
    {
        getLogger() << "Shutting down file manager...";
        // ...
    }
};
```

Now, consider a Meyers Singleton implementation for `FileManager`:

```
FileManager& getFileManager()
{
    static FileManager fileManager;
    return fileManager;
}
```

Whether `getLogger` or `getFileManager` is called first doesn't really matter; if `getFileManager` is called first, the logger will be initialized as part of `FileManager`'s constructor. However, whether the `Logger` or `FileManager` object is destroyed first *is* important:

- If the `FileManager` object is destroyed prior to the `Logger` object, the program will have well-defined behavior.
- Otherwise, the program will have **undefined behavior** because the destructor of `FileManager` will invoke `getLogger`, which will now return a reference to a previously destroyed object.

Logging in the the constructor of the `FileManager` makes it certain that the logger's function-local **`static`** will be initialized before that of the file manager; hence, since destruction occurs in reverse relative order of creation, the logger's function-local **`static`** will be destroyed after 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 counter-indicated. The two most common alternatives elucidated at the end of *Use Cases — Meyers Singleton* on page 6 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

annoyances

Overhead in single-threaded applications

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 invokes a free function, `getS`, returning a function-scope **static** object, `local`, of user-defined type, `S`, having a **user-provided** (inline) 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 generated inlined.[8]

[8]Both GCC 10.x and Clang 10.x, using the `-Ofast` optimization level, generate assembly instructions for an **acquire/release memory barrier** and fail to inline the call to `getS`. Using `-fno-threadsafe-statics` reduces the number of operations performed considerably 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."**??**" on page **??**); doing so will turn `S` into a **trivially-constructible** type, implying that no code needs to be executed during initialization:

```
xor eax, eax  ; zero out 'eax' register
ret           ; return from 'main'
```

A sufficiently smart compiler might, however, not generate synchronization code in a single-threaded context or else provide a flag to control this behavior.

see-also

See Also

None so far.

further-reading

Further Reading

- ?
- ?

le-checked-lock-pattern

Appendix: C++03 Double-Checked Lock Pattern

Prior to the introduction of the **function-scope static** object initialization guarantees discussed in *Description* on page 2, preventing multiple initializations of **static** objects and use before initialization of those same objects was still needed. Guarding access using a `mutex` was often a significant performance cost, so using the unreliable, double-checked lock pattern was often attempted to avoid the overhead:

std::mutexstd::lock_guard

```
Logger& getInstance()
{
    static Logger* volatile loggerPtr = 0;  // hack, used to simulate *atomics*

    if (!loggerPtr)  // Does the logger need to be initialized?
    {
        static std::mutex m;
        std::lock_guard<std::mutex> guard(m);  // Lock the mutex.

        if (!loggerPtr)  // We are first, as the logger is still uninitialized.
        {
            static Logger logger("log.txt");
            loggerPtr = &logger;
        }
    }                    // Either way, the lock guard unlocks the mutex here.

    return *loggerPtr;
}
```

In this example, we are using a **volatile** pointer as a weak substitute for an atomic variable, but many implementations would provide nonportable extensions to support atomic types. In addition to being difficult to write, this decidedly complex workaround would often prove unreliable. The problem is that, even though the logic appears sound, architectural changes in widely used CPUs allowed for the CPU itself to optimize and reorder the sequence of instructions. Without additional support, the hardware would not see the dependency that the second test of `loggerPtr` has on the locking behavior of the mutex and would do the read of `loggedPtr` prior to acquiring the lock. By reordering the instructions or whatever, the hardware would then allow multiple threads to acquire the lock, thinking they are threads that need to initialize the **static** variable.

To solve this subtle issue, concurrency library authors are expected to issue ordering hints such as **fences** and **barriers**. A well-implemented threading library would provide atomics equivalent to the modern `std::atomic` that would issue the correct instructions when accessed and modified. The C++11 Standard makes the compiler aware of these concerns and provides portable *atomics* and support for threading that enables users to handle such issues correctly. The above `getInstance` function could be corrected by changing the type of `loggerPtr` to `std::atomic<Logger*>`. Prior to C++11, despite being complicated, the same function would reliably implement the Meyers Singleton in C++98 on contemporary hardware.

So the final recommended solution for portable thread-safe initialization in modern C++ is to simply let the compiler do the work and to use the simplest implementation that gets the job done, e.g., a Meyers Singleton (see *Use Cases — Meyers Singleton* on page 6):

```
Logger& getInstance()
{
    static Logger logger("log.txt");
    return logger;
}
```

sec-safe-cpp14

Chapter 2

Conditionally Safe Features

ch-conditional
sec-conditional-cpp11 Intro text should be here.

sec-conditional-cpp14

Chapter 3

Unsafe Features

ch-unsafe
sec-unsafe-cpp11 Intro text should be here.

sec-unsafe-cpp14