
6.4. Destroying the Singleton

As discussed, `Singleton` is created on demand, when `Instance` is first called. The first call to `Instance` defines the construction moment but leaves the destruction problem open: When should the singleton destroy its instance? The GoF book doesn't discuss this issue, but, as John Vlissides's book Pattern Hatching (1998) witnesses, the problem is thorny.

Actually, if `Singleton` is not deleted, that's not a memory leak. Memory leaks appear when you allocate accumulating data and lose all references to it. This is not the case here: Nothing is accumulating, and we hold knowledge about the allocated memory until the end of the application. Furthermore, all modern operating systems take care of completely deallocating a process's memory upon termination. (For an interesting discussion on what is and is not a memory leak, refer to Item 10 in Effective C++ [Meyers 1998a].)

However, there is a leak, and a more insidious one: a resource leak. `Singleton` 's constructor may acquire an unbound set of resources: network connections, handles to OS-wide mutexes and other interprocess communication means, references to out-of-process CORBA or COM objects, and so on.

The only correct way to avoid resource leaks is to delete the `Singleton` object during the application's shutdown. The issue is that we have to choose the moment carefully so that no one tries to access the singleton after its destruction.

The simplest solution for destroying the singleton is to rely on language mechanisms. For example, the following code shows a different approach to implementing a singleton. Instead of using dynamic allocation and a static pointer, the `Instance` function relies on a local static variable.

Click here to view code image

```
Singleton& Singleton::Instance()
{
   static Singleton obj;
   return obj;
}
```

This simple and elegant implementation was first published by Scott Meyers (Meyers 1996a, Item 26); therefore, we'll refer to it as the Meyers singleton.

The Meyers singleton relies on some compiler magic. A function-static object is initialized when the control flow is first passing its definition. Don't confuse static variables that are initialized at runtime with primitive static variables initialized with compile-time constants. For example:

```
int Fun()
{
   static int x = 100;
   return ++x;
}
```

In this case, `x` is initialized before any code in the program is executed, most likely at load time. For all that `Fun` can tell when first called, `x` has been 100 since time immemorial. In contrast, when the initializer is not a compile-time constant, or the static variable is an object with a constructor, the variable is initialized at runtime during the first pass through its definition.

In addition, the compiler generates code so that after initialization, the runtime support registers the variable for destruction. A pseudo-C++ representation of the generated code may look like the following code. (The variables starting with two underscores should be thought of as hidden, that is, variables generated and managed only by the compiler.)

Click here to view code image

```
Singleton& Singleton::Instance()
{
   // Functions generated by the compiler
   extern void __ConstructSingleton(void* memory);
   extern void __DestroySingleton();
   // Variables generated by the compiler
   static bool __initialized = false;
   // Buffer that holds the singleton
   // (We assume it is properly aligned)
   static char __buffer[sizeof(Singleton)];
   if (!__initialized)
   {
      // First call, construct object
      // Will invoke Singleton::Singleton
      // In the __buffer memory
      __ConstructSingleton(__buffer);
      // register destruction
      atexit(__DestroySingleton);
      __initialized = true;
   }
   return *reinterpret_cast<Singleton *>(__buffer);
}
```

The core is the call to the `atexit` function. The `atexit` function, provided by the standard C library, allows you to register functions to be automatically called during a program's exit, in a last in, first out (LIFO) order. (By definition, destruction of objects in C++ is done in a LIFO manner: Objects created first are destroyed last. Of course, objects you manage yourself with `new` and `delete` don't obey this rule.) The signature of `atexit` is

Click here to view code image

```
// Takes a pointer to function
// Returns 0 if successful, or
// a nonzero value if an error occurs
int atexit(void (*pFun)());
```

The compiler generates the function `__DestroySingleton` —whose execution destroys the `Singleton` object seated in `__buffer` 's memory—and passes the address of that function to `atexit` .

How does `atexit` work? Each call to `atexit` pushes its parameter on a private stack maintained by the C runtime library. During the application's exit sequence, the runtime support calls the functions registered with `atexit` .

We'll see in a short while that `atexit` has important—and sometimes unfortunate—links with implementing the Singleton design pattern in C++. Like it or not, it's going to be with us until the end of this chapter. No matter what solution for destroying singletons we try, it has to play nice with `atexit` or else we break programmers' expectations.

The Meyers singleton provides the simplest means of destroying the singleton during an application's exit sequence. It works fine in most cases. We will study its problems and provide some improvements and alternate implementations for special cases.

Finally, the simplest—albeit inelegant—solution is the "leaky" Singleton that never destroys itself and lets the operating system take care of cleanup.