Adventures in Systems Programming: C++ Local Statics

腾讯课堂 零声学院备注：这篇文章讲述为什么在局部静态变量时使用了递归锁。

原文地址：https://manishearth.github.io/blog/2015/06/26/adventures-in-systems-programming-c-plus-plus-local-statics/

For a while now I've been quite interested in compilers and systems programming in general; and I feel that an important feature of systems programming is that it's relatively easy to figure out what a line of code does (modulo optimizations) at the OS or hardware level[1]. Conversely, it's important to know how your tools work more than ever in systems programming. So when I see a language feature I'm not familiar with, I'm interested in finding out how it works under the hood.

I'm not a C++ expert. I can work on C++ codebases, but I'm not anywhere near knowing all of the features and nuances of C++. However, I am pretty good at Rust and understand a decent portion of the compiler internals. This gives me a great perspective — I've not yet internalized most C++ features to take them for granted, and I'm well equipped to investigate these features.

Today I came across some C++ code similar to the following[2]:

```
void foo() {
    static SomeType bar = Env()->someMethod();
    static OtherType baz = Env()->otherMethod(bar);
}
```

This code piqued my interest. Specifically, the local `static` stuff. I knew that when you have a static like

```
static int FOO = 1;
```

the `1` is stored somewhere in the `.data` section of the program. This is easily verified with `gdb`:

```
static int THING = 0xAAAA;
int main() {
 return 1;
}
```

```
$ g++ test.cpp -g
$ gdb a.out
(gdb) info addr THING
Symbol "THING" is static storage at address 0x601038.
(gdb) info symbol 0x601038
THING in section .data
```

This is basically a part of the compiled program as it is loaded into memory.

Similarly, when you have a `static` that is initialized with a function, it’s stored in the `.bss`section, and initialized before `main()`. Again, easily verified:

```
#include<iostream>
using namespace std;
int bar() {
 cout<<"bar called\n";
 return 0xFAFAFA;
}
static int THING = bar();
int main() {
 cout<<"main called\n";
 return 0;
}
```

```
$ ./a.out
bar called
main called
$ gdb a.out
(gdb) info addr THING
Symbol "THING" is static storage at address 0x601198.
(gdb) info symbol 0x601198
THING in section .bss
```

We can also leave statics uninitialized (`static int THING;`) and they will be placed in `.bss`[3].

So far so good.

Now back to the original snippet:

```
void foo() {
    static SomeType bar = Env()->someMethod();
    static OtherType baz = Env()->otherMethod(bar);
}
```

Naïvely one might say that these are statics which are scoped locally to avoid name clashes. It's not much different from `static THING = bar()` aside from the fact that it isn't a global identifier.

However, this isn't the case. What tipped me off was that this called `Env()`, and I wasn't so sure that the environment was guaranteed to be properly initialized and available before `main()` is called [4].

Instead, these are statics which are initialized the first time the function is called.

```
#include<iostream>
using namespace std;
int bar() {
 cout<<"bar called\n";
 return 0xFAFAFA;
}
void foo() {
 cout<<"foo called\n";
 static int i = bar();
 cout<<"Static is:"<< i<<"\n";
}
int main() {
 cout<<"main called\n";
 foo();
 foo();
 foo();
 return 0;
}
```

```
$ g++ test.cpp
$ ./a.out
main called
foo called
bar called
Static is:16448250
foo called
Static is:16448250
```

```
foo called
Static is:16448250
```

Wait, “the first time the function is called”? *Alarm bells go off...* Surely there’s some cost to that! Let’s investigate.

```
$ gdb a.out
(gdb) disas bar
   // snip
   0x0000000000400c72 <+15>:    test   %al,%al
   0x0000000000400c74 <+17>:    jne    0x400ca4 <_Z3foov+65>
   0x0000000000400c76 <+19>:    mov    $0x6021f8,%edi
   0x0000000000400c7b <+24>:    callq  0x400a00 <__cxa_guard_acqui
re@plt>
   0x0000000000400c80 <+29>:    test   %eax,%eax
   0x0000000000400c82 <+31>:    setne  %al
   0x0000000000400c85 <+34>:    test   %al,%al
   0x0000000000400c87 <+36>:    je     0x400ca4 <_Z3foov+65>
   0x0000000000400c89 <+38>:    mov    $0x0,%r12d
   0x0000000000400c8f <+44>:    callq  0x400c06 <_Z3barv>
   0x0000000000400c94 <+49>:    mov    %eax,0x201566(%rip)
 # 0x602200 <_ZZ3foovE1i>
   0x0000000000400c9a <+55>:    mov    $0x6021f8,%edi
   0x0000000000400c9f <+60>:    callq  0x400a80 <__cxa_guard_relea
se@plt>
   0x0000000000400ca4 <+65>:    mov    0x201556(%rip),%eax
 # 0x602200 <_ZZ3foovE1i>
   0x0000000000400caa <+71>:    mov    %eax,%esi
   0x0000000000400cac <+73>:    mov    $0x6020c0,%edi
   // snip
```

The instruction at `+44` calls `bar()`, and it seems to be surrounded by calls to some `__cxa_guard` functions.

We can take a naïve guess at what this does: It probably just sets a hidden static flag on initialization which ensures that it only runs once.

Of course, the actual solution isn’t as simple. It needs to avoid data races, handle errors, and somehow take care of recursive initialization.

Let’s look at the spec and one implementation, found by searching for `__cxa_guard`.

Both of them show us the generated code for initializing things like local statics:

```
if (obj_guard.first_byte == 0) {
    if ( __cxa_guard_acquire (&obj_guard) ) {
      try {
      // ... initialize the object ...;
      } catch (...) {
        __cxa_guard_abort (&obj_guard);
        throw;
      }
      // ... queue object destructor with __cxa_atexit() ...;
      __cxa_guard_release (&obj_guard);
    }
  }
```

Here, `obj_guard` is our “hidden static flag”, with some other extra data. `__cxa_guard_acquire` and `__cxa_guard_release` acquire and release a lock to prevent recursive initialization. So this program will crash:

```
#include<iostream>
using namespace std;
void foo(bool recur);
int bar(bool recur) {
 cout<<"bar called\n";
 if(recur) {
    foo(false);
 }
 return 0xFAFAFA;
}
void foo(bool recur) {
 cout<<"foo called\n";
 static int i = bar(recur);
 cout<<"Static is:"<< i<<"\n";
}
int main() {
 foo(true);
 return 0;
}
```

```
$ g++ test.cpp
$ ./a.out
```

```
foo called
bar called
foo called
terminate called after throwing an instance of '__gnu_cxx::recursive_init_error'
  what():  std::exception
Aborted (core dumped)
```

Over here, to initialize `i`, `bar()` needs to be called, but `bar()` calls `foo()` which needs `i` to be initialized, which again will call `bar()` (though this time it won't recurse). If `i` wasn't `static` it would be fine, but now we have two calls trying to initialize `i`, and it's unclear as to which value should be used.

The implementation is pretty interesting. Before looking at the code my quick guess was that the following would happen for local statics:

- `obj_guard` is a struct containing a mutex and a flag with three states: "uninitialized", "initializing", and "initialized". Alternatively, use an atomic state indicator.
- When we try to initialize for the first time, the mutex is locked, the flag is set to "initializing", the mutex is released, the value is initialized, and the flag is set to "initialized".
- If when acquiring the mutex, the value is "initialized", don't initialize again
- If when acquiring the mutex, the value is "initializing", throw some exception

(We need the tristate flag because without it recursion would cause deadlocks)

I suppose that this implementation would work, though it's not the one being used. The implementation in bionic (the Android version of the C stdlib) is similar; it uses per-static atomics which indicate various states. However, it does not throw an exception when we have a recursive initialization, it instead seems to deadlock[5]. This is okay because the C++ spec says (Section 6.7.4)

> If control re-enters the declaration (recursively) while the object is being initialized, the behavior is undefined.

However, the implementations in gcc/libstdc++ (also this version of `libcppabi` from Apple, which is a bit more readable) do something different. They use a global recursive mutex to handle reentrancy. Recursive mutexes basically can be locked multiple times by a single thread, but cannot be locked by another thread till the locking thread unlocks them the same number of times. This means that recursion/reentrancy won't cause deadlocks, but we still have one- thread-at-a-time access. What these implementations do is:

- `guard_object` is a set of two flags, one which indicates if the static is initialized, and one which indicates that the static is being initialized ("in use")
- If the object is initialized, do nothing (this doesn't use mutexes and is cheap). This isn't exactly part of the implementation in the library, but is part of the generated code.

- If it isn't initialized, acquire the global recursive lock
- If the object is initialized by the time the lock was acquired, unlock and return
- If not, check if the static is being initialized from the second `guard_object` flag. If it is "in use", throw an exception.
- If it wasn't, mark the second flag of the static's guard object as being "in use"
- Call the initialization function, bubble errors
- Unlock the global mutex
- Mark the second flag as "not in use"

At any one time, only one thread will be in the process of running initialization routines, due to the global recursive mutex. Since the mutex is recursive, a function (eg `bar()`) used for initializing local statics may itself use (different) local statics. Due to the "in use" flag, the initialization of a local static may not recursively call its parent function without causing an error.

This doesn't need per-static atomics, and doesn't deadlock, however it has the cost of a global mutex which is called at most once per local static. In a highly threaded situation with lots of such statics, one might want to reevaluate directly using local statics.

LLVM's libcxxabi is similar to the `libstdc++` implementation, but instead of a recursive mutex it uses a regular mutex (on non-ARM Apple systems) which is unlocked before `__cxa_guard_acquire` exits and tests for reentrancy by noting the thread ID in the guard object instead of the "in use" flag. Condvars are used for waiting for a thread to stop using an object. On other platforms, it seems to deadlock, though I'm not sure.

So here we have a rather innocent-looking feature that has some hidden costs and pitfalls. But now I can look at a line of code where this feature is being used, and have a good idea of what's happening there. One step closer to being a better systems programmer!

Thanks to Rohan Prinja, Eduard Burtescu, and Nishant Sunny for reviewing drafts of this blog post

---

1. Emphasis on *relatively*. This article will show that it's definitely not "easy" all the time.↩
2. This was JNI code which obtained a JNI environment and pulled out method/class IDs from it to be used later↩
3. Unless it has a constructor or otherwise isn't made out of trivially constructible types; in this case it is treated similar to the previous case.↩
4. I checked later, and it was indeed the case that global statics are initialized before `Env()` is ready↩
5. I later verified this with a modification of the crashing program above stuck inside some JNI Android code.↩