91-06-17

Initialization and Destruction of Static Objects

X3J16/91-0053

-- *John Wilkinson, jfw@sgi.com*

Silicon Graphics

1. Initialization of nonlocal static objects

1.1 What should the standard say?

There are two principles which it seems highly desirable to observe:

(1) Nonlocal static objects defined in a translation unit must be initialized in the order that their definitions appear in the translation unit.

(2) A nonlocal static object must be initialized before its first use.

The current draft does not specify (1). It says that constructors for nonlocal static objects are called in the order they occur in a file (Section 12.6.1), but says nothing about other initializations. This needs to be corrected.

The draft goes farther than (2):

(3) The nonlocal static objects in a translation unit must be initialized before the first use of any function or object defined in that translation unit (Section 3.4).

But this will not do, as the following example shows:

```
int x = f() // first use of f()
```

If the use of f() in the initialization of x is its first use, then (3) says that x must be initialized before it is initialized.

To avoid this difficulty, Jonathan suggested changing the wording of (3) to something like

(3a) The nonlocal static objects in a translation unit must be initialized before the first use of any function or object defined in that translation unit, in a thread starting from main().

This however abandons (2) entirely. It allows (though it does not require) an implementation to initialize all nonlocal static objects at startup, without regard to ordering between translation units.

If we want to retain the apparent intent of (3), I suggest the wording

(3b) The nonlocal static objects in a translation unit must be initialized before any object or function defined in that unit is used by any other translation unit; the nonlocal static objects in the translation unit containing main() must be initialized before control enters main().

I suggest then that the draft should include language equivalent to (1), and language equivalent to either (3a), or (3b). (3a) acknowledges the status quo and abandons the attempt to enforce initialization order between translation units. (3b) expresses the apparent intent of the current draft. In the next section, I discuss possible implementations of (3b).

1.2 Implementation

If we are going to put something like (3b) into the standard, we should make sure that the implementation cost is not too great. Even though something similar is in the draft now, everybody has really been getting along without it; in this respect it is like an extension. Nobody is going to want it if, for example, every translation unit in a program has to be recompiled whenever any one of them changes. This means that the necessary information must be present in the compiled code (object files or libraries: I note in passing that these terms are not defined in the draft standard). Again, if too much stuff has to be added to the object files to support this feature, or if the runtime cost is too great, then it probably isn't worth it. I will look at two possible approaches, which are intended to work with information that can reasonably be expected to be available in existing object files. The first approach, static determination of initialization order, unfortunately doesn't quite work. The second seems entirely workable, but entails a slight runtime cost.

I make the following assumptions:

(1) The translator generates a single static initialization function for each translation unit which defines one or more nonlocal static objects requiring initialization, and this initialization function can be recognized: it might, for example, be called __sti__<something>.

(2) We can determine for a given function what symbols it refers to which are not defined in the translation unit where the function is defined.

(3) We can determine for each symbol what translation unit it is defined in.

All these assumptions should be valid, for example, for the output of a UNIX relocatable load.

1.2.1 Static determination of initialization order

We build a dependency graph among the functions in the program as follows:

If f calls g, then f depends on g.

If f refers to a symbol defined in another translation unit, then f depends on the static initialization function for that translation unit.

We ignore the functions that cannot be reached by a path from main() or from the static initializer function for the translation unit containing main(). If any static initialization function belongs to a cycle, complain. Otherwise take any initialization order satisfying the condition that if sti2 depends on sti1, then sti1 is called before sti2.

Finally, we patch the executable so that the static initialization functions are called at startup in the order determined.

Unfortunately this doesn't quite work, as illustrated by the following example:

```
main.C:          X.C:                      Y.C:

extern X x1;     extern Y y1;              Y::Y() {n = 5;}
                                            Y y1;
                 virtual int foo();
                 };

                 struct D: B {
                 virtual int foo();
                 };

                 int D::foo()
                 {return y1.n;}

                 struct X {
                 int n;
                 X();
                 };

                 X::X()
                 {bp = new D;
                 n = bp->foo();}

                 X x1;
```

Here the dependency of X::X() on D::foo() is hidden by the virtual function mechanism, so the need to initialize y1 before x1 is undetected..

1.2.2 Dynamic initialization

For each translation unit with a static initialization function, build a table with one entry for each externally visible symbol defined in that translation unit. Initialize the table with zeros (or any invalid addresses).

For each translation unit, for every reference to a symbol defined in another unit with a static initialization function, replace that reference by an indirect reference through the appropriate address table.

Build a fault-handler to catch the invalid references. The handler should call the appropriate static initialization function and initialize the appropriate table (the table address should be available). The handler should be set up so that the address lookup is retried on return from the handler.

I believe this should work, at the cost of a couple of instructions for every external reference to a symbol defined in a translation unit with nonlocal static objects requiring initialization. This cost could be trimmed somewhat: the translation unit containing main(), for example, has to be initialized at startup anyway, so references into it would not need to become indirect.

2. Destruction of static objects

Destructors for static objects are discussed in the following passages:

In Section 3.4:

"Destructors...for initialized static objects are called when returning from main() and when calling exit().

Destruction is done in reverse order of initialization...If atexit() is to be called, objects initialized before an atexit() call may not be destroyed until after the function specified in the atexit() call has been called."

In Section 6.7:

"The destructor [for a local static object] must be called either immediately before or as part of the calls to the atexit() functions. Exactly when is undefined."

This is confusing in a number of ways:

(1) 3.4 says that a local static object must be destroyed after the calls to the atexit() functions; 6.7 says the opposite.

(2) 3.4 says that local static objects are destroyed in reverse order of initialization; 6.7 implies this order is undefined.

(3) If a function with a local static object with a destructor is registered with atexit(), the rule in 6.7 seems to have problems.

(4) If there is no atexit() call, what happens with local static objects? Presumably they should still get destroyed, but 6.7 doesn't say that.

(5) If 3.4 is supposed to apply just to nonlocal static objects, then 3.4 and 6.7 together say that local static objects should get destroyed before nonlocal static objects if there is an atexit() call. It seems reasonable that this should be the case anyway, in spite of the objection in (3).

(6) If 3.4 applies to local static objects, it seems to say more than is needed; I see no reason why local static objects in different scopes should have to be destroyed in reverse order of initialization.

(7) I have trouble with "objects initialized before an atexit() call..." What difference does it make when an object is initialized in relation to when atexit() is called?

I would suggest rewording these sections with the following effect:

(1) All destructors for static objects should be called after the functions registered with atexit().

(2) Destructors for nonlocal static objects should be called in reverse order of initialization.

(3) Destructors for local static objects should be called before destructors for nonlocal static objects. Destructors for static objects local to a given function should be called in reverse order of initialization. The precise timing of the calls is otherwise undefined.