X3J16/96-0016
WG21/N0834
November 9, 1995
J. Stephen Adamczyk
Edison Design Group
jsa@edg.com

Static versus dynamic initialization

This paper provides some background for a motion at the Tokyo meeting to change the rules regarding dynamic initialization. The change allows implementations to optimize some dynamic initializations by rendering them as static initializations. [Note: the motion passed. This paper is in the mailing to get the reasons behind the change on the record.]

Two-phase initialization

The Working Paper, in 3.6.2 [basic.start.init], specifies that initialization of non-local objects is done in two phases. First, certain initializations are done statically (typically, this means at load time). Then, all the remaining initializations are done dynamically (i.e., by executed code at runtime), in the order of their definitions within a translation unit.

The current Working Paper indicates a small set of initializations that are done statically. 3.6.2 [basic.start.init] paragraph 1 indicates that objects initialized with constant expressions are done statically; 8.5.1 [dcl.init.aggr] paragraph 13 says that for aggregates that are initialized with both constant and nonconstant expressions, the initializations involving constant expressions can be done in either the static phase or the dynamic phase.

The problem

The core working group got into this topic by discussing whether certain specific initializations should be added to the set of static initializations, namely

- Aggregates for which all the initializers are constant expressions.
- References initialized with simple lvalues.

and whether some should be removed from that set, namely

- Initializations for which the initializer is a constant expression, but the expression is actually the argument of a constructor or conversion function call.

This led to discussion of other similar cases, and it became clear that (a) it would be very difficult to pin down, for every initialization case, whether the initialization must be static or not (to pick just one example, is "`const int&r = 1`" a static initialization?), and (b) even if we could do it, it probably isn't a good idea. There must be some latitude left to implementations. In particular, it seemed desirable to allow implementations to do static initialization for something like

```
struct complex {
  float r;
  float i;
  complex(float rp, float ip) : r(rp), i(ip) {}
};
complex x(1,0, 2.0);
```

The problem is that the current promise made to programmers is that all static initializations are done first, and then all dynamic initializations; that means that to fulfill the promise, a compiler has to know precisely which initializations are static. Presumably, all the rest are dynamic.

An idea

The working group's idea is to change the promise made to programmers to "when a variable is initialized, all of the variables whose definitions precede it in the compilation unit will already be initialized, and all of the variables whose definitions follow it in the compilation unit and whose initializations are required to be static will already be initialized."

How does this differ from the original promise? It means that if the initializer of a potentially-dynamic initialization refers to the value of a potentially-dynamically-initialized variable that appears later in the same compilation unit, the results aren't guaranteed. The later variable might be initialized (if the implementation chose to do static initialization for that variable), or it might be merely zero-initialized (if the implementation chose to do dynamic initialization). For example:

```
extern complex y;
complex x(1.0, y.i);  // Unspecified value for y.i
complex y(1,0, 2.0);
```

The working group felt that such references could be viewed as poor programming practice, and therefore making them unspecified is reasonable.

The details

[Note: this paper was prepared at the Tokyo meeting, and the wording below matches the motion that passed. The motion was incorporated into the WP with slightly different wording.]

In 3.6.2 [basic.start.init], paragraph 1, insert at the end of the sentence "Objects with static storage duration initialized with constant expressions ...":

> ; see also _dcl.init.aggr_ for additional initialization cases for which initialization must be done statically.

In 3.6.2 [basic.start.init], insert after paragraph 1, as a new paragraph:

> An implementation is permitted to implement an initialization of a nonlocal object with static storage duration as a static initialization even when such initialization is not required to be done statically, provided that (a) the dynamic version of the initialization would not cause the value of any other nonlocal object with static storage duration to change prior to its initialization, and (b) the static version of the initialization produces the same value in the initialized object as would be produced if all initializations not required to be done statically were done dynamically. [Note: as a consequence, it is unspecified whether a reference, in an initialization, to a variable potentially requiring dynamic initialization, and defined later in that compilation unit, will obtain the value fully initialized or merely zero-initialized. Example:
>
> ```
> struct A {
>  float x;
>  A(float p) : x(p) {}
> };
> extern A b;
> A a(b.x);   // unspecified forward reference to b.x
> A b(1.0);
> ```
>
> -- end note]

In 8.5.1 [dcl.init.aggr], add at the end of paragraph 13:

> If all of the member initializer expressions are constant expressions, and the aggregate is a POD type, the initialization is done during the static phase of initialization."