From: Willem Wakker
Date: May 2008

Issue: initializing static or external variables that are not initialized explicitly.

Discussion (from email SC22WG14.11416):

In the (good?) old days there was K&R C which stated (appendix A, para 8.6):

> Static and external variables which are not initialized are guaranteed to start off as 0; automatic and register variables which are not initialized are guaranteed to start off as garbage.

Earlier in the K&R book (chapter 4, page 84) an example is given where external variables are explicitly initialized with 0 and the text says:

> These initializations are actually unnecessary since all are zero, but it's good form to make them explicit anyway.

So, one might expect that many K&R C programs will rely on the fact that static and external variables that are not initialized explicitly are implicitly set to zero.

Then comes C89. Subclause 6.5.7 says:

> If an object that has static storage duration is not initialized explicitly, it is initialized implicitly as if every member that has arithmetic type were assigned 0 and every member that has pointer type were assigned a null pointer constant.

This is exactly as what was written in K&R C; note the use of the term 'null pointer constant' here: a null pointer constant is 'an integral constant expression with the value 0, or such an expression cast to type void *' (see 6.3.2.#3). I do not think that at this time the possibility that a null pointer (result of assigning a null pointer constant to a pointer) might be represented by something that does not have all bits zero was taken into account, and that still the full K&R intentions were honored.

Then comes DR_016 which (in question 2) starts of with:

> This one is relevant only for hardware on which either null pointer or floating point zero is */not/* represented as all zero bits.

It deals with

> union { char *p; int i; } x;

and then states:

> If the null pointer is represented as, say, 0x80000000, then there is no way to implicitly initialize this object. Either the p member contains the null pointer, or the i member contains 0, but not both. So the behavior of this translation unit is undefined. This is a bad state of affairs. I assume it was not the Committee's intention to prohibit a large class of implicitly initialized unions; this would render a great deal of existing code nonconforming.

The issue here is about the representation of the null pointer which is unspecified (6.2.6.1#1: The representation of all types are unspecified except as stated in this subclause). The claim of non-conformance here is false (I think): if a program depends on the unspecified representation of pointer types then it is non-conforming anyway. There is however a problem with the cited union in the context 6.5.7 of C89 related to 'what does it means that everything is cleared, i.e. all bits are set to zero'. My inclination in handling this problem would have been rather than stepping forward (defining what we think all those null bits should mean for the values of the types, or, even worse: define

what values there should be for the various types) to take a step backward and say something to the effect: 'the data space of the objects is cleared (all bits set to zero); it is implementation defined(??) what this means for the values of the objects'.

However the committee decided otherwise (the even worse part of above) using 'null pointer' rather than 'null pointer constant' and decided on the union issue that only the first member was to be initialized.

This may have a silent effect on a whole class of programs that worked correctly under the K&R/C89 specifications, namely those programs where the 2nd (3rd, ...) member of a union has a bigger size than the first member: for the union

union { int i; short sa[20] s; } u;

only the i is initialized leaving a hole in the rest of the union; programs may no longer rely on the initialization to 0 of the s. To make things even worse: on 'normal' systems uninitialized data is commonly allocated in a BSS or COMMON segment which is usually cleared at start-up time; on these systems the effect is not noted. Only when you port a such a program to a system whereby the compiler has to generate explicit initializations and is strictly following the C99 rules the problem occurs.

Summary:

- the committee solution, as implemented in C99, introduces a silent change from C89 to C99 that is not even mentioned in the rationale, and diverts explicitly from what I would call the 'K&R/C89 spirit of C';
- the working of a program now may depend on the order in which the members of a union are defined, which is an unwanted effect (think of generated C programs);
- the analogy (from DR #016) with the fact that for explicit initializations also only the first element of a union is initialized is false: there the programmer knows and sees what he is doing;
- one could imagine that the uninitialized holes in the data space might be exploited and cause a security risk.

Action item resulting from discussion at Delft (April 2008) WG14 meeting:

Submit changes to C1x to flood static memory with zero bytes, then patch floats and pointers as needed.

Proposed change to C1X:

Change the 2nd sentence of 5.1.2p1 to:

Before program startup first the storage area that holds all objects with static storage shall be cleared (all bytes are set to zero), then all objects with static storage duration shall be *initialized* (set to their initial values).