Document: N1731
Date: 2013/07/23

Issues with alignment in C11

Joseph Myers

There are various deficiencies in the C11 text about alignment requirements.

Issue 1: Existence of over-aligned types

6.2.8#3 defines the concept of an over-aligned type, with a footnote saying "Every over-aligned type is, or contains, a structure or union type with a member to which an extended alignment has been applied.". But there is no way in the syntax to apply such an alignment to a member. _Alignas appears in the syntax for alignment-specifier, which in turn appears in that for declaration-specifiers (6.7#1). But structure and union members instead use struct-declaration which uses specifier-qualifier-list which doesn't include a case for alignment-specifier at all. So for the reference to over-aligned types, and the reference in 6.7.5#6 to the "declared object or member", to be meaningful, something needs adding to the syntax for struct-declaration. (Note that specifier-qualifier-list is also used in the syntax for type-name, and it seems less likely that a type-name was intended to be able to include alignment-specifiers.)

Issue 2: Contexts in which alignments are supported

6.2.8#2 defines "fundamental alignment": "A fundamental alignment is represented by an alignment less than or equal to the greatest alignment supported by the implementation in all contexts, which is equal to _Alignof (max_align_t)."

6.2.8#3 defines "extended alignment": "An extended alignment is represented by an alignment greater than _Alignof (max_align_t). It is implementation-defined whether any extended alignments are supported and the contexts in which they are supported. A type having an extended alignment requirement is an over-aligned type."

6.2.8#4 defines "valid alignment", saying "Alignments are represented as values of the type size_t. Valid alignments include only those values returned by an _Alignof expression for fundamental types, plus an additional implementation-defined set of values, which may be empty. Every valid alignment value shall be a nonnegative integral power of two.".

max_align_t is specified in 7.19#2 as "an object type whose alignment is as great as is supported by the implementation in all contexts".

The memory management functions in 7.22.3 are defined to return a pointer "suitably aligned so that it may be assigned to a pointer to any type of object with a fundamental alignment requirement and then used to access such an object or an array of such objects in the space allocated". In the case of aligned_alloc, there may be a stricter requirement given by the alignment passed to the function, but the alignment passed to the function can't result in memory any less-aligned than a fundamental alignment requirement. The alignment requirement still applies even if the size is too small for any object requiring the given alignment (see the response to C90 DR#075).

There are various problems with the above:

* The term "fundamental type" is not defined in C11.

* There is also no definition of what a "context" is in which an alignment might or might not be supported. In common implementation practice, separate contexts might be by the storage duration of the object (static, thread, automatic, allocated, with the last referring to the alignments guaranteed by calloc, malloc and realloc).

* A "valid alignment" may not be a "fundamental alignment". Thus, whatever interpretation is adopted for "fundamental type", nothing in the standard requires the alignment of a "fundamental type" to be a "fundamental alignment". For example, say "long double" is a "fundamental type"; it would seem nonsensical if declaring "long double" objects (in any context) failed to work, but nothing seems to require malloc to return objects sufficiently aligned for long double.

* Given these gaps in the definition, nothing in the normative text appears to imply footnote 57 "Every over-aligned type is, or contains, a structure or union type with a member to which an extended alignment has been applied.", although no doubt it reflects the intent.

* If "fundamental type" is interpreted to mean "basic type", that is not sufficient to resolve these lacunae. For example, if

  ```
  struct s { long double ld; }
  ```

  has an alignment requirement bigger than long double, it should still be possible to allocate memory for it with malloc, and the same applies to any typedef from a standard header that might also have a bigger alignment requirement than any basic type.

The following principles seem natural for any fix for this issue:

* C99 referred to "any type of object" in the alignment requirements for calloc, malloc and realloc. As a matter of compatibility, this means that any type that could be constructed within C99 (including one using types from standard headers) should have an alignment required by C11 to be supported in all contexts, and the same applies to types from C extensions originally specified as extensions to C99. Most of the following principles follow to a greater or lesser extent from this compatibility principle.

* All basic types have alignments supported in all contexts.

* All enumerated types have alignments supported in all contexts.

* All pointer types have alignments supported in all contexts (even if the type pointed to does not).

* All types from standard headers specified as complete object types in the definitions of those headers have alignments supported in all contexts. (This includes both types specified as typedefs and types

specified as structs or unions with a given tag.)

* If a type has an alignment supported in all contexts, so do arrays of that type, qualified versions of that type, and atomic versions of that type.

* If all the members of a structure or union have types with alignments supported in all contexts, and none of them use an _Alignas specifier specifying an alignment bigger than supported in all contexts, then that structure or union has an alignment supported in all contexts.

* Where C extensions such as TS 18661-2 and 18661-3 are proposed that define new types, or existing such extensions such as TR 18037 are revised and updated for C11, care should be taken that the new types are covered under the above, whether through being basic types or through being defined in standard headers. (If SIMD vector types, as mentioned at <http://www.open-std.org/pipermail/cplex/2013-June/000010.html>, were to end up in any such extension, it would probably be appropriate to define them in a way that does *not* require calloc, malloc and realloc to return memory suitably aligned for them; such types often require alignments bigger than needed for any other type, so imposing such requirements on memory allocation functions would result in undue inefficiency.)