Doc No: SC22/WG14/N1335
Date: 8/7/2008
Reply to: Arjun Bijanki (arjun.bijanki@microsoft.com)

Adding Alignment Support to C

Summary
SC22/WG21/N2341 [link] proposes wording to the C++ language and libraries to support alignment, and the language aspects of alignment in that paper are largely applicable to C. This paper proposes their adoption in a semantically equivalent manner (syntax aside, for the moment). The aspects of alignment intended primarily to support generic libraries in C++ are not included for adoption.

For reference, changes voted into the C++ working paper were:

- New: *alignment-specifier* (**alignas**) to declarations
- New: **alignof** expression to retrieve alignment requirements of a type (like **sizeof** for
- size)
- New: alignment arithmetic by library support (**aligned_storage**, **aligned_union**)
- New: standard function (**std::align**) for pointer alignment at run time

Of those changes, the single proposed change to C is the addition of support for specifying stricter alignment:

- New: *alignment-specifier* (**alignas**) to declarations

The other three changes are not proposed for C, since they're largely necessary for template containers and don't seem to have a high degree of utility for C code:

- ~~New: **alignof** expression to retrieve alignment requirements of a type (like **sizeof** for size)~~
- ~~New: alignment arithmetic by library support (**aligned_storage**, **aligned_union**)~~
- ~~New: standard function (**std::align**) for pointer alignment at run time~~

Questions
- Should packing and weaker alignment requirements be considered?
- Does C need aligned allocation (e.g. **posix_memalign**)?

Wording
The proposed wording draws heavily on the wording in WG21/N2341. Note: wording additions are underlined; deletions are in ~~strikethrough~~.

6.4.1

[add **alignas** to the list of keywords]

6.2.8 Alignment of objects [new section]

1. Object types have *alignment requirements* which place restrictions on the addresses at which an object of that type may be allocated. An *alignment* is an implementation-defined integer value representing the number of bytes between successive addresses at which a given object can be allocated. An object type imposes an alignment requirement on every object of that type: stricter alignment can be requested using **alignas**.

2. A *fundamental alignment* is represented by an alignment less than or equal to the greatest alignment supported by the implementation in all contexts.

3. An *extended alignment* is represented by an alignment greater than the greatest alignment supported by the implementation in all contexts. 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.

4. Alignments are represented as values of the type **size_t.** Valid alignments include only those values supported by the implementation for fundamental types, plus an additional implementation-defined set of values, which may be empty. [*Footnote*: It is intended that every valid alignment value is an integral power of two]

5. Alignments have an order from *weaker* to *stronger* or *stricter* alignments. Stricter alignments have larger alignment values. An address that satisfies an alignment requirement also satisfies any weaker valid alignment requirement.

6. The types **char**, **signed char**, and **unsigned char** shall have the weakest alignment requirement.

7. If a request for a specific extended alignment in a specific context is not supported by an implementation, the implementation is allowed to reject the request as ill-formed. The implementation is also allowed to silently disregard the requested alignment.

6.7 Declarations

Syntax

declaration-specifiers:

storage-class-specifier declaration-specifiers$_{opt}$
type-specifier declaration-specifiers$_{opt}$
type-qualifier declaration-specifiers$_{opt}$
function-specifier declaration-specifiers$_{opt}$
alignment-specifier

6.7.9 Alignment specifiers [new section]

1. The alignment specifier has the form

alignment-specifier
alignas *(constant-expression)*

2. The *constant-expression* shall be an integer constant expression.

- If the constant expression evalues to a fundamental alignment, the alignment requirement of the declared object shall be the specified fundamental alignment.

- If the constant expression evaluates to an extended alignment and the implementation supports that alignment in the context of the declaration, the alignment of the declared object shall be that alignment.
- If the constant expression evalues to an extended alignment and the implementation does not support that alignment in the context of the declaration, the program is ill-formed.
- Otherwise, the program is ill formed.

3. If a declaration contains multiple alignment specifiers, the program is ill-formed.

4. An alignment specifier shall not be specified in a declaration of a typedef, or a bit-field, or a function parameter or return type, or an object declared with the **register** storage-class specifier.

5. If the defining declaration of an object has an alignment specifier, any non-defining declaration of that object shall either specify equivalent alignment or have no alignment specifier. No diagnostic is required if declarations of an object have different alignment specifiers in different translation units.

7.20.3/1

The order and contiguity of storage allocated by successive calls to the **calloc**, **malloc**, and **realloc** functions is unspecified. The pointer returned if the allocation succeeds is 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 (until the space is explicitly deallocated).