Doc No:	SC22/WG21/N2140 J16/06-0210 of project JTC1.22.32	**Address:**	LM Ericsson Oy Ab Hirsalantie 1 Jorvas 02420
Date:	2006-11-05 to 2006.11.06 21:55:00	**Phone:**	+358 40 507 8729 (mobile)
Reply to:	Attila (Farkas) Fehér	**Email:**	attila f feher at ericsson com wolof at freemail hu

Adding Alignment Support to the C++ Programming Language / Consolidated

1 Short summary

Document status: proposal with the changes suggested by core and library. Proposed wording will be in the mid term mailing.

One-liner: Extending the standard language and library with alignment related features.

Problems targeted:

- Allow most efficient fixed capacity-dynamic size containers and optional elements
- Allow specially aligned variables/buffers for hardware related programming
- Allow building heterogeneous containers at run time
- Allow programming of discriminated unions
- Allow optimized code generation for data with stricter alignment

Related issues not addressed:

- Class-type “packing” (although allowed)
- Requesting specially aligned memory from memory allocators (**`new`**, **`malloc`**)

Proposed changes:

- New: *alignment-specifier* (**`alignas`**) to declarations (type based and value based)
- New: **`alignof`** operator to retrieve alignment value for a type (like **`sizeof`** for size)
- New: alignment arithmetic by library support
- New: standard functions for pointers for proper alignment at run time

Note: The proposed library changes are different from the ones we voted on at the 2006 Portland meeting due to input from the Library Working Group!

Table of Contents

2 Alignment defined

1 Alignment is a quality of an address. An address may satisfy several alignment requirements.

2 *Alignment requirements* are implementation defined integer values, expressed as *alignment values*. Implementations may define alignment requirements (with their corresponding *alignment values*) that are not *alignment requirements* of built-in types.

3 *Alignment values* have the type **`size_t`**. When used as an *alignment value*, the valid value set of **`size_t`** type only includes those values returned by the **`alignof`** operator and those additionally specified by the implementation.

4 The *alignment value* 0 is reserved for future use. All other alignment values are implementation-defined. *Alignment values* that are not powers of 2 are reserved for the implementation to define alignment requirements that are only known at run time. Other *alignment values* (powers of two) represent alignment requirements known at compile time; and are called static alignments.

5 All complete types have, one and only one, static alignment requirement that can be retrieved using the **`alignof`** operator. That alignment will be the optimal alignment for the type.

6 The types **`char`**, **`signed char`** and **`unsigned`** char have the weakest possible alignment.

7 Alignments have an order from weaker to stronger. The strongest alignment of a set of alignments is called the strictest alignment. Stronger alignments have larger alignment values.

8 An alignment requirement is satisfied by all alignments that are a multiple of it.

9 Comparing alignment values of type **`size_t`** is supported and provides the obvious results: two alignment values are equal, if their numeric values are equal; when an alignment value is larger than another it represents a stricter, but not necessarily compatible, alignment. The remainder operation can be used to detect if an alignment satisfies an alignment requirement; in which case the remainder of the division by the required alignment value is zero.

3 The `alignof` operator

1 The **`alignof`** operator returns the alignment requirement of its argument.

expression:

`alignof` (*type-id*)

2 The alignof expression is a constant expression of type **`std::size_t`**.

3 The operand is a *type-id* representing a complete type.

4 When applied to a reference or a reference type, the alignment of the referenced type is used.

5 When applied to a base class subobject is the alignment of the base class type is used.

6 When applied to an array type-id, the alignment of the element type is used.

7 The **`alignof`** operator can be applied to a pointer to a function, but shall not be applied directly to a function.

8 Types shall not be defined in an **`alignof`** expression.

[Note: The operator will return the special alignment value of a type that has aligned members:

```
struct cacheline {  // Assuming cache lines are 128 bytes
  char alignas(128) memory[128];
};

std::size_t a=aligof(cacheline);

// a==128
```

4 The `alignas` alignment specifier

1 The **`alignas`** alignment specifier is:

alignment-specifier:

alignas (*type-id*)
alignas (*constant-expression*)

2 Any number of *alignment-specifier* may appear in a given declaration. In case the requested multiple alignments cannot be solved into a single static alignment the program is ill-formed.

3 If an *alignment-specifier* appears in a declarator (*init-declarator-list*), there can be no **`typedef`** specifier in the same *simple-declaration*.

4 The *alignment-specifier* applies to the name declared by the *init-declarator* that precedes it.

5 The *alignment-specifier* can be applied only to names of objects, names of data members. [Note: In particular, function parameters cannot be declared with an alignment-specifier.]

6 Declarations that are not definitions (do not allocate storage to objects) may omit the *alignment-specifier* as long as the definition contains it. [Note: it is recommended to use the *alignment specifier* in the declarations too, as that provides an opportunity for better code generation.] If a declaration contains *alignment-specifiers* it has to specify the same alignment as the *alignment-specifiers* of the corresponding definition, otherwise the program is ill-formed.

7 An *alignment-specifier* used in the declaration of an object declares the object to satisfy the alignment requirements defined by its argument.

8 A *type-id* argument shall denote a complete type.

A constant expression (alignment value) argument shall represent a static alignment (shall be a power of 2). A *constant-expression* argument of the *alignment-specifier* shall only be an *alignment-value* that represents an alignment requirement known at compile time. [Note: In other words only powers of 2, see **Table of Contents**

9 Alignment defined).

10 An *alignment-specifier* shall not weaken the alignment of the object it is applied to.

11 An implementation shall support alignments of all built-in types in alignment specifiers. If an implementation supports specification of stricter alignments it may specify different limits to the supported alignments in different contexts. [Note: For example it may limit alignments applicable to automatic variable.]

5 (Compatible) changes to `aligned_storage`

1 **`aligned_storage`** becomes:

```
template <std::size_t Len, std::size_t... Alignments>
struct aligned_storage {
  static const std::size_t alignment_value=N;
  typedef integral_constant<alignment_value> alignment_of;
  implementation-defined-POD type;
};
```

2 The template is backwards compatible with the TR1 **`aligned_storage`**.

3 The restrictions on the supported alignments are lifted. Now it is implementation defined what "special" alignments are supported in different contexts.
[Note: So it is possible to create an aligned storage that is aligned on page boundary (let's say 8096 bytes) but an implementation may stop you from using it as an automatic (stack stored) variable.)]

4 The template now supports variadic arguments and thus we are able to specify many alignments. All of them must (still) be static alignments; powers of 2.

5 The template first calculates the weakest alignment that satisfies all the requested alignments. (Basically binary-ors all alignments, then picks the largest digit that is set.) The inner constant **`alignment_value`** will contain the resulting number. The inner type **`alignment_of`** is the same number as a TR1 **`alignment_of`** template metafunction applied to this type.

6 The typical implementation of the inner POD type will be:

```
template <std::size_t __S, std::size_t __A>
struct __aligned_type {
  char alignas(A) __arr[__S];
};
typedef __aligned_type<Len,alignment_value> type;
```

6 New template `aligned_union`

1 **`aligned_union`** becomes:

```
template <std::size_t Len, T... Types>
struct aligned_union {
  static const std::size_t alignment_value=N;
  typedef integral_constant<alignment_value> alignment_of;
  implementation-defined-POD type;
};
```

2 The template is same as the TR1 **`aligned_storage`**, but it takes types instead of alignment values, and uses the alignment requirements of all the types.

3 Passing 0 for **`Len`** will cause the template to calculate it. In such case Len will be the maximum of all the sizes of all the type arguments.

4 The template first calculates the weakest alignment that satisfies the alignment requirements of all types. The inner type **`alignment_of`** is the same number as a TR1 **`alignment_of`** template metafunction applied to this type.

5 The typical implementation of **`aligned_union`** will be done by publicly inheriting from **`aligned_storage`** and using variadic template techniques together with the **`alignof`** operator to pass the alignment values of each template argument to the base.

7 Run-time pointer alignment

1 The run-time pointer alignment functions forward-adjust (increase) the value of a pointer within a buffer to the closest value that satisfies the alignment requirement specified by a given alignment value. The alignment value may be a run-time alignment (non-power of 2) value as well. The run-time pointer alignment functions come in C and C++ flavors: the **`stdalign`** function in the **`<cstdlib>`** headers; and the **`std::align`** function in the **`<memory>`** header.

2 The functions work on buffers defined by a pointer to the first available byte and the size that is the number of the remaining bytes in the buffer. In addition to the required alignment, the callers need to specify the size (in bytes) of the element(s) to be stored at the aligned pointer. If the alignment cannot be done within the remaining buffer, or there would not remain enough space for the user defined size the functions have no effect and returns a NULL pointer.

3 In case the alignment is done, the functions adjust the pointer (into the buffer) to point after the aligned area (with the user defined size); and decrease the value of remaining buffer space by the number of bytes used up for alignment plus the number of bytes the caller indicated to use.

4 The functions return the aligned pointer value; or the NULL pointer value if the alignment was not done.

[Note: The run-time alignment functions can be used to test a pointer against an alignment requirement by passing 0 for buffer size and required size. The function will return the NULL pointer value, if the pointer passed in is not well-aligned.]

7.1 The `stdalign` function

```
void *stdalign( size_t align_val, size_t size, void **pptr, ptrdiff_t *pspace);
```

1 *Effects:* The value of the pointer pointed by ***`pptr`*** is increased by the minimum amount necessary to satisfy the alignment requirements represented by ***`align_val`***. The value pointed by ***`pspace`*** is decreased by the amount of bytes used up during the pointer value increase plus the value of the ***`size`*** argument; but only if it is possible to do so within the buffer denoted by the original value of the pointer pointed by ***`pptr`*** and space in bytes pointed by the ***`pspace`*** argument. If the alignment cannot be done within the mentioned buffer, or there would not remain at least ***`size`*** bytes in the buffer after the alignment is done, calling the function has no effects. If the value of the pointer pointed by ***`pptr`*** is the NULL pointer value, or the buffer denoted by it and the value (as size in bytes) pointed by the ***`pspace`*** argument is not allocated to the application the effect of the function are undefined.

2 *Returns:* The modified value of the pointer pointed by ***`pptr`*** or the NULL pointer value if the function had no effect.

7.2 The `std::align` function

```
void *
align(
  std::size_t align_val,
  std::size_t size,
  void *&ptr,
  std::ptrdiff_t &space
  );
```

1 Same as **`stdalign`**, but it takes references (not pointers) to the arguments it changes.