Doc No: SC22/WG21/N2341
J16/07-0201
of project JTC1.22.32

Date: 2007-07-18

Phone: +358 40 507 8729 (mobile)
+1-503-712-8433

Reply to: Attila (Farkas) Fehér

Clark Nelson

Email: attila f feher at ericsson com
wolof at freemail hu
clark.nelson@intel.com

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

Short summary

Document status: wording proposal to be considered by CWG and LWG.

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

Problems targeted:

- Allow most efficient implementation of fixed capacity-dynamic size containers
- Allow most efficient implementation of 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 allocators (`new`, `malloc`)

Proposed changes:

- 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

The numbering in this document is based on N2315 Working Draft, Standard for Programming Language C++.

Typographical conventions:

- New paragraphs, notes examples etc. are normally typesetted
- Insertions into existing text are green and double underlined

- ~~Deletions~~ from existing text are green and stricken through
- Any other change to existing text is unintentional and shall be ignored

Special thanks to **Premanand Rao** of HP for his hands with design and wording, **Clark Nelson** of Intel for his effective guidance on managing the task and **Benjamin Kosnik** for his help with Library wording.

Alignment Wording Proposal

Add new keywords to **2.11 Keywords** [lex.key]

Add the words **alignas** and **alignof** before the **asm** keyword.

Add new bullet to **3.2 One definition rule** §4 note [basic.def.odr]

- the type T is the subject of an **alignof** expression (5.3.6) or an **alignas** specifier (7.1.6), or

Update **3.7.3.1 Allocation functions** §2 [basic.stc.dynamic.allocation]

2 The allocation function attempts to allocate the requested amount of storage. If it is successful, it shall return the address of the start of a block of storage whose length in bytes shall be at least as large as the requested size. There are no constraints on the contents of the allocated storage on return from the allocation function. The order, contiguity, and initial value of storage allocated by successive calls to an allocation function ~~is~~ are unspecified. The pointer returned shall be suitably aligned so that it can be converted to a pointer of any complete object type with a fundamental alignment requirement (3.11) and then used to access the object or array in the storage allocated (until the storage is explicitly deallocated by a call to a corresponding deallocation function). Even if the size of the space requested is zero, the request can fail. If the request succeeds, the value returned shall be a non-null pointer value (4.10) p0 different from any previously returned value p1, unless that value p1 was subsequently passed to an operator delete. The effect of dereferencing a pointer returned as a request for zero size is undefined.[37)]

Update **3.9 Types** [basic.types]

Move paragraph 5 to 3.11 Alignment below, as paragraph 1, with the indicated modifications. References to alignment in 3.9 should be changed to refer to 3.11 (there are about seven such references).

Add note to **3.9.2 Compound types** §3 [basic.compound]

[*Note:* Pointers to over-aligned types have no special representation, but their valid value range is restricted by the extended alignment requirement. This international standard specifies only two ways of obtaining such a pointer: taking the address of a valid object with over-aligned type, or using one of the runtime pointer alignment functions. An implementation may provide other means of obtaining a valid pointer value for an over-aligned type. – *end note*]

Add **3.11 Alignment** **[basic.align]**

1 Object types have *alignment requirements* (3.9.1, 3.9.2) which place restrictions on the addresses at which an object of that type may be allocated. ~~The~~ An *alignment* ~~of a complete object type~~ is an implementation-defined integer value representing ~~a~~ the number of bytes between successive addresses at which a given object can be allocated. ~~an object is allocated at an address that meets the alignment requirements of its object type.~~ 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, which is equal to **`alignof(std::max_align_t)`** (18.1).

3 An *extended alignment* is represented by an alignment greater than **`alignof(std::max_align_t)`**. It is implementation-defined whether any extended alignments are supported and the contexts in which they are supported (7.1.6). A type having an extended alignment requirement is an *over-aligned type*. [*Note:* Every over-aligned type is or contains a class type with a non-static data member to which an extended alignment has been applied. – *end note*]

4 Alignments are represented as values of the type **`std::size_t`**. Valid alignments include only those values returned by an **`alignof`** expression for the 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. – *end footnote*]

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 lesser valid alignment requirement.

6 The alignment requirement of a complete type can be queried using an **`alignof`** expression (5.3.6). Furthermore the types **`char`**, **`signed char`** and **`unsigned char`** shall have the weakest alignment requirement. [*Note:* This enables the character types to be used as the underlying type for an aligned memory area (7.1.6).– *end note*]

7 Comparing alignments is meaningful and provides the obvious results:

- Two alignments are equal when their numeric values are equal.
- Two alignments are different when their numeric values are not equal.
- When an alignment is larger than another it represents a stricter alignment.

8 [Note: The run-time pointer alignment functions (20.4.7) can be used to obtain an aligned pointer within a buffer; and aligned-storage support templates in the library can be used to obtain aligned storage (20.6.8).]

9 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. [*Note:*

Additionally, a request for run-time allocation of dynamic memory for which the alignment cannot be honored may be treated as an allocation failure. – *end note*]

Update **5.3 Unary expressions** §1 **[expr.unary]**

1 Expressions with unary operators group right-to-left.

unary-expression:
  postfix-expression
  ++ *cast-expression*
  -- *cast-expression*
  unary-operator cast-expression
  `sizeof` *unary-expression*
  `sizeof ( ` *type-id* **`)`**
  `alignof (` *type-id* **`)`**
  new-expression
  delete-expression

unary-operator: one of
  `* & + - ! ~`

Update **5.3.4 New** §1, §11 **[expr.new]**

1 The *new-expression* attempts to create an object of the *type-id* (8.1) or *new-type-id* to which it is applied. The type of that object is the *allocated type*. This type shall be a complete object type, but not an abstract class type or array thereof (1.8, 3.9, 10.4). It is implementation-defined whether over-aligned types are supported (3.11). [*Note:* because references are not objects, …

11 A *new-expression* passes the amount of space requested to the allocation function as the first argument of type **`std::size_t`**. That argument shall be no less than the size of the object being created; it may be greater than the size of the object being created only if the object is an array. For arrays of **`char`** and **`unsigned char`**, the difference between the result of the *new-expression* and the address returned by the allocation function shall be an integral multiple of the ~~most stringent~~ strictest fundamental alignment requirement (3.9, 3.11) of any object type whose size is no greater than the size of the array being created. [*Note:* Because allocation functions are assumed to return pointers to storage that is appropriately aligned for objects of any type with fundamental alignment, this constraint on array allocation overhead permits the common idiom of allocating character arrays into which objects of other types will later be placed. — *end note*]

Add **5.3.6 Alignof** **[expr. alignof]**

1 An **`alignof`** expression yields the alignment requirement of its operand type. The operand shall be a *type-id* representing a complete object type.

2 The result is an integral constant of type **`std::size_t`**.

3 When alignof is applied to a reference type, the result is the alignment of the referenced type. When alignof is applied to an array type, the result is the alignment of the element type.

4 A type shall not be defined in an **`alignof`** expression.

Update **5.19 Constant expressions** §1 **[expr. const]**

Note: these changes will not be necessary once the `constexpr` proposal has been adopted.

1 In several places, C++ requires expressions that evaluate to an integral or enumeration constant: as array bounds (8.3.4, 5.3.4), as case expressions (6.4.2), as bit-field lengths (9.6), as enumerator initializers (7.2), as static member initializers (9.4.2), and as integral or enumeration non-type template arguments (14.3).

constant-expression:
 conditional-expression

An *integral constant-expression* shall involve only literals of arithmetic types (2.13, 3.9.1), enumerators, non-volatile **`const`** variables and static data members of integral and enumeration types initialized with constant expressions (8.5), non-type template parameters of integral and enumeration types, and **`sizeof`** expressions, and **`alignof`** expressions. Floating literals (2.13.3) shall appear only if they are cast to integral or enumeration types. Only type conversions to integral and enumeration types shall be used. In particular, except in **`sizeof`** and **`alignof`** expressions, functions, class objects, pointers, or references shall not be used, and assignment, increment, decrement, function call (including new-expressions and delete-expressions), comma operators, and throw-expressions shall not be used.

Update **7.1 Specifiers** §1 **[dcl.spec]**

1 The specifiers that can be used in a declaration are

decl-specifier:
 storage-class-specifier
 type-specifier
 function-specifier
 `friend`
 `typedef`
 alignment-specifier

Insert **7.1.6 Alignment specifier** **[dcl.align]**

1 The alignment specifier has the form

alignment-specifier:
 `alignas (` *constant-expression* **`)`**
 `alignas (` *type-id* **`)`**

2 When the alignment specifier is of the form **`alignas(`***constant-expression***`)`**:

- the constant expression shall be an integral constant expression

- if the constant expression evaluates 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 evaluates to an extended alignment and the implementation does not support that alignment in the context of the declaration, the program is ill-formed
- if the constant expression evaluates to zero, the alignment specifier shall have no effect
- otherwise, the program is ill-formed

3 When the alignment specifier is of the form **`alignas(`***type-id***`)`**, it shall have the same effect as **`alignas(alignof(`***type-id***`))`** (5.3.6).

4 When multiple alignment specifiers are specified for an object, the alignment requirement shall be set to the strictest specified alignment.

5 The combined effect of all alignment specifiers in a declaration shall not specify an alignment that is less strict than the alignment that would otherwise be required for the object being declared.

6 An alignment specifier shall not be specified in a declaration of a typedef, or a bit-field, or a reference, or a function parameter or return type, or an object declared with the **`register`** storage-class specifier. [*Note:* In short, the specifier can be used on automatic variables, namespace scope variables, members of class types (as long as they are not bit-fields). In other words it cannot be used in contexts where it would become part of a type so it would affect name mangling, name lookup or ordering of function templates. – *end note.*]

7 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.

8 [*Example:* An aligned buffer, with an alignment requirement of A and an element type T other than `char`, `signed char` or `unsigned char`, can be declared as:

```
T alignas(T) alignas(A) buffer[N];
// where N is the number of T elements making up the buffer
```

Specifying **`alignas(T)`** in the alignment specifier list will ensure that the final requested alignment will not be weaker than **`alignof(T)`**, and therefore the program will not be ill-formed. – *end example*]

9 [*Note*: The alignment of a union type can be strengthened by applying the alignment specifier to any member of the union. – *end note*]

10 [*Note:* The **`aligned_union`** template (20.4.7) can be used to create a union containing a type with a non-trivial constructor or destructor. – *end note*]

Update **8.1 Type names** §1 **[dcl.name]**

1 To specify type conversions explicitly, and as an argument of **`sizeof`**, **`alignof`**, **`new`**, or **`typeid`**, the name of a type shall be specified. This can be done with a *type-id*, which is syntactically a declaration for an object or function of that type that omits the name of the object or function.

The rest of the paragraph is unchanged.

Update **14.6.2.2 Type-dependent expressions** §4 **[temp.dep.expr]**

4 Expressions of the following forms are never type-dependent (because the type of the expression cannot be dependent):

literal

postfix-expression `.` *pseudo-destructor-name*

postfix-expression **`->`** *pseudo-destructor-name*

`sizeof` *unary-expression*

`sizeof (` *type-id* **`)`**

`alignof (` *type-id* **`)`**

`typeid (` *expression* **`)`**

`typeid (` *type-id* **`)`**

`::`$_{opt}$ **`delete`** *cast-expression*

`::`$_{opt}$ **`delete [ ]`** *cast-expression*

`throw` *assignment-expression*$_{opt}$

[Note: For the standard library macro offsetof, see 18.1. —end note]

Update **14.6.2.3 Value-dependent expressions** §2 **[temp.dep.constexpr]**

2 An *identifier* is value-dependent if it is:

- a name declared with a dependent type,
- the name of a non-type template parameter,
- a constant with integral or enumeration type and is initialized with an expression that is value-dependent.

Expressions of the following form are value-dependent if the *unary-expression* is type-dependent or the *type-id* is dependent ~~(even if **`sizeof`** *unary-expression* and **`sizeof (`** *type-id* **`)`** are not type-dependent)~~:

`sizeof` *unary-expression*

`sizeof` **(** *type-id* **)**

`alignof` **(** *type-id* **)**

[*Note:* For the standard library macro **`offsetof`**, see 18.1. —*end note*]

Update **18.1 Types** table 17 **[support.types]**

Table 17: Header <cstddef> synopsis

Type	Name(s)
Macros:	`NULL offsetof`
Types:	`ptrdiff_t size_t max_align_t`

Add **18.1 Types** §5 **[support.types]**

5 `max_align_t` is a POD type whose alignment requirement is at least as great as that of every scalar type, and whose alignment requirement is supported in every context.

Add **20.1.2 Allocator requirements §6** **[allocator.requirements]**

6 If the alignment associated with a specific over-aligned type is not supported by an allocator, instantiation of the allocator for that type is allowed to fail. The allocator is also allowed to silently disregard the alignment. [*Note:* Additionally, `allocate` for that type may fail by throwing `std::bad_alloc`. – *end note*]

Update **20.4.2 Header <type_traits> synopsis** **[meta.type.synop]**

Add **`aligned_union`** synopsis:

```
// [20.4.7] other transformations:
template <std::size_t Len, class ... Types> struct aligned_union;
```

Add to **20.4.7 Other transformations** **[meta.trans.other]**

Table 51: Other transformations

Template	Condition	Comments
`template <template <std::size_t Len, std::size_t Align> struct aligned_storage;`	`Len` is nonzero. `Align` is equal to `alignment_of<T>::value` for some type T.	The member typedef `type` shall be a POD type suitable for use as uninitialized storage for any object whose size is at most *Len* and whose alignment is a divisor of *Align*.
`template < std::size_t Len, class ... Types > struct aligned_union;`	At least one type is provided.	The member typedef **`type`** shall be a POD type suitable for use as uninitialized storage for any object whose type is listed in *Types*; its size shall be at least

		Len. The static member **`alignment_value`** shall be an integral constant of type **`std::size_t`** whose value is the strictest alignment of all types listed in *Types*.

1 [Note: A typical implementation would define **`aligned_storage`** as:

```
  template <std::size_t Len, std::size_t Alignment>
  struct aligned_storage {
    typedef struct {
      alignas(Alignment) unsigned char __data[Len];
    } type;
  };
```

– *end note*]

2 It is implementation defined whether any extended alignment is supported (3.11).

Extend **20.6 Memory** §1 synopsis **[memory]**

```
// 20.6.8 Pointer aligner function
void *align(std::size_t alignment, std::size_t size, void *&ptr, std::size_t&
space);
```

Update **20.6.1.1 Allocator members** §4 **[allocator.members]**

4 *Returns*: a pointer to the initial element of an array of storage of size *n* `* sizeof(T)`, aligned appropriately for objects of type `T`. It is implementation defined whether over-aligned types are supported (3.11).

Update **20.6.3 Temporary buffers** §1 **[temporary.buffer]**

1 *Effects*: Obtains a pointer to storage sufficient to store up to *n* adjacent *T* objects. It is implementation defined whether over-aligned types are supported (3.11).

Add subclause **20.6.8 Align** **[ptr.align]**

```
  namespace std {
    void *align(
      std::size_t alignment,
      std::size_t size,
      void *&ptr,
      std::size_t &space
    );
  }
```

1 *Effects:* If it would be possible to fit *size* bytes of storage aligned by *alignment* into the buffer starting at *ptr* with length *space,* the function updates *ptr* to point to the first possible address of such storage and decreases *space* by the number of bytes used for alignment. Otherwise the function has no effect.

2 *Requires:*

- *alignment* to be a fundamental alignment value or an extended alignment value supported by the implementation in this context
- *ptr* is pointing to at least *space* bytes of contiguous storage

3 *Returns:* A null pointer if an aligned buffer could not be accommodated; otherwise the resulting value of *ptr*.

4 [*Note:* The function updates its *ptr* and *space* arguments so that it can be repeatedly called with possibly different *alignment* and *size* arguments for the same buffer. – *end note*]