Doc No: SC22/WG21/**N1971**
J16/**06-0041**
of project JTC1.22.32

Date: 2002-05-02 to 2006-02-26

Reply to: Attila (Farkas) Fehér

Address: LM Ericsson Oy Ab
Hirsalantie 1
Jorvas 02420

Phone: +358 40 507 8729 (mobile)

Email: attila f feher at ericsson com
wolof at freemail hu

Adding Alignment Support to the C++ Programming Language

1 Short summary

Document status: proposal for standards wording.

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 runtime
- 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* to declarations (type based and value based)
- New: **`alignof`** operator to retrieve alignment value for a type (like **`sizeof`** for size)
- New: alignment arithmetic, **`alignof`** as operator to create a union of alignments (for discriminated unions)
- New: standard functions for pointers for proper alignment runtime

This document is based on ISO/IEC 14882:1998(E) version of the standard.

2 Wording proposal

2.1 Alignment

Alignment is a quality of an address. The alignment can be expressed as an implementation-defined integer value representing a number of bytes (see 3.9 §5), called the *alignment value*. The alignment value is specified to have the type **`align_t`**; which is an implementation defined signed integer type defined in the standard header **`<cstddef>`** (18.1) and **`<cstdlib>`**. The *alignment value* 0 is reserved for future use. All other alignment values are implementation-defined. Negative *alignment values* are reserved for the implementation to define alignment requirements that are only known runtime. Positive *alignment values* represent alignment requirements (and alignments) known compile time; and are called fixed alignments. All complete types have fixed alignment requirements that can be retrieved using the unary **`alignof`** operator (3.5.6).

Alignments have an order, from weaker to stronger. Stronger alignments have larger alignment values. An alignment requirement is satisfied by all alignments that have alignment values divisible without remainder by the alignment value representing the alignment requirement in question.

Implementations may define further alignments that are not related to any type. Those may be fixed-alignments or – when their alignment value is negative – alignments that are only known runtime.

The type **`align_t`** is a signed integral type; however its valid value range only includes those values specified by the implementation when used as an alignment-value. No implementations shall return negative alignment-values from any form of the **`alignof`** operator. [Note: Or in other words, the **`alignof`** operator only returns (and accepts) fixed alignment values that can be used in alignment specifications. Portable applications shall not depend on other values than those returned by the **`alignof`** operator applied to types; and should not mutate these values afterwards.] Comparing alignment values of type **`align_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 alignment. The remainder operation can be used to detect if one alignment (represented by an alignment value) satisfies the alignment requirement (represented by an alignment value); in that case the remainder is zero.

2.2 Add the `alignof` keyword to 2.11 Keywords [lex.key]

Add the word **`alignof`** before the **`asm`** keyword.

2.3 The `alignof` operator

2.3.1 5.3 § and grammar to be changed as (change is underlined):

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*)

alignof (*constant-expression*)

new-expression

delete-expression

unary-operator: one of

* & + ! ~

2.3.2 New section 5.3.6 unary `alignof` expression [expr.unary.alignof] to be added

5.3.6 alignof [expr.unary.alignof]

The unary **`alignof`** operator yields the alignment value representing the alignment requirements represented by its operand. The parenthesized operand is either a constant-expression that represents a valid alignment-value, or a type-id representing a complete type. If the operand is an alignment-value that value is returned. If the operand is a type-id of a complete type, the alignment value representing the alignment requirements of that type is returned.

The **`alignof`** operator shall not be applied to an expression that yields to a negative alignment value, or the behavior is undefined.

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

The result of applying **`alignof`** to a base class subobject is the alignment of the base class type.

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

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

The lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions are not applied to the operand of **`alignof`**.

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

The result is a constant of type **`align_t`**. [Note: **`align_t`** is defined in the standard header **`<cstddef>`** (18.1) and **`<cstdlib>`**.]

2.3.3 New section 5.19 `alignof` operator [expr. alignof] to be inserted

The **`alignof`** operator has two forms. The unary-expression form is discussed in 5.3.6. This section discusses the form with more than one argument.

expression:

alignof (*alignment-list*)

alignment-list:

alignment-argument

alignment-list, alignment-argument

alignment-argument:

type-id

constant-expression

The non-unary **`alignof`** operator yields the smallest alignment value representing the (weakest) alignment that fulfills the alignment requirements represented by all of its operands. The operands

are either a constant-expressions that represent a valid fixed alignment-value, or a type-ids representing complete types.

The non-unary **alignof** operator works as if its arguments first are converted to alignment-values using the unary **alignof** operator (see 5.3.6). All restrictions and listed in 5.3.6 apply.

The result is a constant of type **align_t**. [Note: **align_t** is defined in the standard header **<cstddef>** (18.1) and **<cstdlib>**.]

2.3.4 Change section 5.19 (becomes 5.20) Constant expressions [expr. const] to be changed in the listed paragraphs, as indicated by the underlined text

In § 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), and **alignof** operator arguments (5.3.6, 5.19).

(grammar unchanged)

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

In § 3:

An arithmetic constant expression shall have arithmetic or enumeration type and shall only have operands that are integer literals (2.13.1), floating literals (2.13.3), enumerators, character literals (2.13.2) and **sizeof** (5.3.3) or **alignof** (5.3.6, 5.19) expressions ~~(5.3.3)~~. *(Rest remains unchanged)*

2.4 The (`alignof`) alignment specifier

2.4.1 Insert 7.1.2 Alignment specifier [dcl.align] (and renumber the following 7.1.x items)

The alignment specifier is:

alignment-specifier:

alignof (*alignment-list*)

alignment-list:

alignment-argument

alignment-list, alignment-argument

alignment-argument:

type-id

constant-expression

At most one *alignment-specifier* shall appear in a given declaration. If an *alignment-specifier* appears in a *to-be-decided*, there can be no **`typedef`** specifier in the same *simple-declaration*.

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

The *alignment-specifier* can be applied only to names of objects and to anonymous unions (9.5). There can be no aligned function parameters.

An *alignment-specifier* used in the declaration of an object declares the object to satisfy the alignment requirements defined by the elements of the *alignment-argument-list* (?.?.?). An *alignment-specifier* can be used in declarations of class members; ?.? describes its effect.

The object being declared shall have large enough size (as sizeof) to store any type specified by any of the *type-id*s in the *alignment-specifier*, otherwise the program is ill-formed. This restriction also applies to class member declarations. That restriction is not applied to declarations of names with no known size, such as extern declarations of arrays.

[Note: If alignment of a smaller entity is to be set to the alignment requirements of a larger type, the **`alignof`** operator shall be used to turn the type-id of the large type into an alignment value:

```
int i alignof(double); // Ill-formed if sizeof(double)>sizeof(int)

int i alignof( alignof( double ) ); // Turned type-id into alignment value: valid

extern int ia[] alignof(double); // Allowed as long as the definition defines a large
  // enough array to store a double. Implementations may use the knowledge of
  // double-alignment to optimize access to the array.
```

end note]

The *alignment-arguments-list* of the *alignment-specifier* shall only contain *alignment-values* that represent alignment requirements known compile time (see ?.?.?), or in other words only positive *alignment-values*, otherwise the program is ill-formed.

Declarations that are not definitions (do not allocate storage to objects) may omit the *alignment-specifier* as long as the definition contains it.

2.5 Runtime pointer alignment

The runtime pointer alignment functions forward-adjust (increase) the value of a pointer within a buffer to the closest value that satisfies an alignment requirement specified by a given alignment value. The alignment value may be runtime alignment (negative) value as well. The runtime 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.

2.5.1 The `stdalign` function

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

Effects:

The value of the pointer pointed by ***`pptr`*** is increased by the minimum amount necessary for it to satisfy the alignment requirements represented by ***`align_val`*** and 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.

Returns:

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

2.5.2 The `std::align` functions

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

Effects:

The value of the pointer referenced by ***`ptr`*** is increased by the minimum amount necessary for it to satisfy the alignment requirements represented by ***`align_val`*** and the value referenced by ***`space`*** 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 referenced by ***`ptr`*** and space in bytes referenced by the ***`space`*** 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 referenced by ***`ptr`*** is the NULL pointer value, or the buffer denoted by it and the value (as size in bytes) referenced by the ***`space`*** argument is not allocated to the application the effect of the function are undefined.

Returns:

The modified value of the pointer referenced by ***`ptr`*** or the NULL pointer value if the function had no effect.