Doc No: SC22/WG21/N2165
J16/07-0025
of project JTC1.22.32

Date: 2007-01-14 to 2007.01.15 04:53:00

Reply to: Attila (Farkas) Fehér

Address: LM Ericsson Oy Ab
Hirsalantie 11
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 / 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`** operator to retrieve alignment requirements of a type (like **`sizeof`** for size)
- New: alignment arithmetic by library support (**`aligned_storage`**, **`aligned_union`**)
- New: standard functions for pointer alignment at run time

The numbering in this document is based on N2134 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
- Existing coloring, underlining and strike-through from N2134 is kept for clarity
- Any other change to existing text is unintentional and shall be ignored

Special thanks to **Premanand Rao** of HP for his hands on help with this proposal and for **Clark Nelson** of Intel for his guidance and effective assistance on managing the task.

Alignment Wording Proposal

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

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

Update **3.2 One definition rule** §4 note **[basic.def.odr]**

- the **`typeid`** operator (5.2.8) ~~or~~, the **`sizeof`** operator (5.3.3), the **`alignof`** operator (5.3.6) or the **`alignas`** specifier (8.3.7) is applied to an operand of type T, 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 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 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). Which – if any – extended alignment requirements are fulfilled by the returned pointer is implementation defined. 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 **3.9.1 Fundamental types** §2, §8 **[basic.fundamental]**

2 There are five *standard signed integer types* : "**`signed char`**", "**`short int`**", "**`int`**", "**`long int`**", and "**`long long int`**". In this list, each type provides at least as much storage and has as much alignment requirements as those preceding it in the list. There may also be implementation-defined extended signed integer types. The standard and extended signed integer types are collectively called signed integer types. Plain **`int`**s have the natural size and alignment suggested by the architecture of the execution environment[44)]; the other signed integer types are provided to meet special needs.

8 There are three *floating point* types: **`float`**, **`double`**, **`and long double`**. The type **`double`** provides at least as much precision as **`float`**, and the type **`long double`** provides at least as much precision as **`double`**. The set of values of the type **`float`** is a subset of the set of values of the type **`double`**; the set of values of the type **`double`** is a subset of the set of values of the type **`long double`**. The alignment requirements of floating point types are implementation defined. The value representation of floating-point types is implementation-defined. *Integral* and *floating* types are collectively called *arithmetic* types. Specializations of the standard template **`std::numeric_limits`** (18.2) shall specify the maximum and minimum values of each arithmetic type for an implementation.

Add note to **3.9.2 Compound types** §2 **[basic.compund]**

[*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 only specifies

two ways of obtaining such a pointer: taking the address of a valid object of the over-aligned type, or using one of the runtime pointer alignment functions with a large enough buffer. Other – if any – means of obtaining a valid pointer value for an over-aligned type is implementation defined. – *end note*]

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

1 Alignment is a quality of an address (3.9 §5). An address may satisfy several alignment requirements (also called *well aligned*). [Note: The run-time pointer alignment functions (20.4.8) can be used to obtain an aligned position within a buffer; and aligned-storage support templates in the library can be used to obtain aligned storage (20.6.8).]

1 *Alignment requirements* are implementation defined integer values (3.9 §5), expressed as *alignment values*. This international standard defines two kinds of alignments:

2 *Fundamental alignments* are

- Alignments of fundamental types
- Alignments of any type that is not affected by any **`alignas`** alignment specifier [*Note:* A type can only be affected by the **`alignas`** alignment specifier by applying it to non-static members of class types or members of union types. (8.3.7) – *end note*]
- Alignments of any type that is affected by an **`alignas`** specifier that sets the alignment requirements to any of the previously listed fundamental alignments

3 *Extended alignments* are all the alignments that are not fundamental alignments. What – if any – extended alignments and in what context are supported is implementation-defined (8.3.7). Types having extended alignment requirements are henceforth termed *over-aligned types*. [*Note:* Over-aligned types are meant to support special alignments and their support is consciously not guaranteed in any context. – *end note*]

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

5 Alignments have an order from *weaker* to *stronger* or *stricter* alignments. Stricter alignments have larger alignment values. [*Note:* A larger alignment is not necessarily proper alignment for a weaker alignment. Portable ways of detecting proper alignments involve the use of library facilities (20.4.8, 20.6.8). – *end note*]

6 All complete types have, one and only one, alignment requirement that can be retrieved using the **`alignof`** operator (5.3.6). Furthermore the types **`char`**, **`signed char`** and **`unsigned char`** shall have the weakest possible alignment requirement. [*Note:* This enables the char types to be used as the underlying type for an aligned memory area (8.3.7).– *end note*]

7 An alignment requirement is satisfied by all alignments that are a multiple of it. The remainder operation can be used to detect if an alignment value satisfies an alignment requirement; in which case the remainder of the division by the required alignment value is zero.

8 Comparing alignment values 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 value is larger than another it represents a stricter, but not necessarily compatible, alignment
- the rest of the possible comparisons are self-evident, based on the previous points

Extend **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** §11 and §15 **[expr.new]**

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 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 that is not over-aligned, 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*]

15 [*Note:* when the allocation function returns a value other than null, it must be a pointer to a block of storage in which space for the object has been reserved. The block of storage is assumed to be appropriately aligned and of the requested size. The address of the created object will not necessarily be the same as that of the block if the object is an array. The ways – if any – to obtain dynamically allocated memory with extended alignment is implementation defined. —*end note*]

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

1 The **alignof** operator takes the following form:

expression:
alignof (*type-id* **)**

9 The **alignof** operator yields the alignment requirement of its operand as an alignment value.

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

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

12 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**.

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

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

15 For union types, the strictest alignment requirement of all members is used.

16 For enumerations the alignment of the underlying type is used.

17 For other POD types the alignment of the first non-static data member is used.

18 For non-POD types the alignment requirement is unspecified. [*Note:* Non-POD types may contain padding before their first non-static data member, which may be used to provide proper alignment for that member. The alignof operator will give the proper alignment requirements for such types as well, but it is unspecified what it will be. – *end note*]

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

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

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

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* ~~can~~shall involve only literals of arithmetic types (2.13, 3.9.1), enumerators, non-volatile **const** variables ~~or~~and static data members of integral ~~or~~and enumeration types initialized with constant expressions (8.5), non-type template parameters of integral ~~or~~and enumeration types, and **sizeof** expressions, and **alignof** expressions. Floating literals (2.13.3) ~~can~~shall appear only if they are cast to integral or enumeration types. Only type conversions to integral ~~or~~and enumeration types ~~can~~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*), ~~or~~ comma operators, and *throw-expressions* shall not be used.

Update **8 Declarators** §4 **[dcl.decl]**

4 Declarators have the syntax:

declarator:
direct-declarator
ptr-operator declarator

direct-declarator:
alignment-specifier-list$_{opt}$ *declarator-id*
direct-declarator **(** *parameter-declaration-clause* **)** *cv-qualifier-seq*$_{opt}$ *exception-specification*$_{opt}$
direct-declarator **[** *constant-expression*$_{opt}$ **]**
(*declarator* **)**

ptr-operator:
***** *cv-qualifier-seq*$_{opt}$
&

&&
::$_{opt}$ *nested-name-specifier* ***** *cv-qualifier-seq*$_{opt}$

cv-qualifier-seq:
cv-qualifier cv-qualifier-seq$_{opt}$

cv-qualifier:
const
volatile

declarator-id:
id-expression
::$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name*

alignment-specifier-list:
alignment-specifier

A class-name has special meaning in a declaration of the class of that name and when qualified by that name using the scope resolution operator **::** (5.1, 12.1, 12.4).

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.

Insert **8.3.7 Alignment specifier** **[dcl.align]**

1 The alignment specifier has the form

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

2 The alignment specifiers apply to the name declared by the *declarator-id* that precedes it, and specifies the alignment requirement for the object declared by that name.

3 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 value 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 value 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

4 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).

5 When multiple alignment specifiers are specified for an object, the alignment requirement shall be set to the weakest alignment that meets the alignment requirements of each specifier. If no such alignment can be found, the program is ill-formed.

6 The combined effect of all alignment specifiers shall not specify an alignment that is less strict than the alignment that would otherwise be required for the object being declared; or an alignment that is not compatible with the declared type.

7 The alignment specifier shall not be specified in a **`typedef`** declaration.

8 The alignment specifier shall not be specified for a *bit-field* declaration.

9 The alignment specifier shall not be specified for a reference object, or a function parameter, or a function return type.

10 The alignment specifier shall not be specified for object with the **`register`** storage specifier.

11 The alignment specifier shall not be specified for a function or a member function directly, but may be specified for a pointer to a function or a pointer to a member function.

12 [*Note:* In short, the specifier can be used on automatic, namespace level, namespace level static variables, members of unions, 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 effect name mangling, name lookup or ordering of function templates. – *end note.*]

13 The alignment specifier may be omitted for the declarations that do not obtain storage, even if a later definition of the name has alignment specifiers. However, if a declaration has an alignment specifier, all of its declarations and its definition shall have the same alignment specifier. No diagnostic are necessary if the differing declarations/definition are in different translation units.

14 [Note: For creating aligned buffers it is advisable to use the type unsigned char as underlying type; since that type has the weakest alignment and it represents unsigned bytes of memory. – end note.]

15 [Example: If any other type T than char, signed char or unsigned char is used as underlying type for an aligned buffer for an alignment requirement represented by A (type or integral constant expression) the portable way to define such a buffer is:

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

This is necessary since A might represent a weaker alignment than **`alignof(T)`**, but listing **`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.]

16 [Note: Strengthening alignment of a union type may be done by applying the alignment specifier onto any member of the union. – end note.]

17 [Note: To create a union containing a type with non-trivial constructor/destructor the **`aligned_union`** (20.4.8) can be used. – end note.]

Update **14.5.5.1 Function template overloading** §5 note **[temp.over.link]**

[*Note:* Most expressions that use template parameters use non-type template parameters, but it is possible for an expression to reference a type parameter. For example, a template type parameter can be used in the **`sizeof`** or the **`alignof`** operator. —*end note*]

Extend **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]

Extend **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** SEE ALSO **[support.types]**

SEE ALSO: subclause 5.3.3, Sizeof; subclause 5.3.6, Alignof; subclause 5.7, Additive operators; subclause 12.5, Free store; and ISO C subclause 7.1.6.

(Semicolons added to clarify what belongs together.)

Update **18.5.1.1 Single object forms** §1,§3,§7 **[new.delete.single]**

1 *Effects:* The *allocation function* (3.7.3.1) called by a *new-expression* (5.3.4) to allocate *size* bytes of storage suitably aligned to represent any object of that size that have fundamental alignment requirement. [*Note:* The means – if any – of allocating memory with extended alignment is implementation defined.–*end note*]

3 *Required behavior:* Return a non-null pointer to suitably aligned storage (3.7.3) for any fundamental alignment (3.11), or else throw a **`bad_alloc`** exception. This requirement is binding on a replacement version of this function.

7 *Required behavior:* Return a non-null pointer to suitably aligned storage (3.7.3) for any fundamental alignment (3.11), or else return a null pointer. This nothrow version of **`operator new`** returns a pointer obtained as if acquired from the ordinary version. This requirement is binding on a replacement version of this function.

Update **18.5.1.2 Array forms** §1 **[new.delete.array]**

1 *Effects:* The allocation function (3.7.3.1) called by the array form of a *new-expression* (5.3.4) to allocate *size* bytes of storage suitably aligned to represent any array object – that has fundamental alignment requirement – of that size or smaller.[218] [*Note:* The means – if any – of allocating memory with extended alignment is implementation defined.–*end note*]

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

Change **`aligned_storage`** synopsis to say:

```
// [20.4.8] other transformations:
template <std::size_t Len, std::size_t... Alignments> struct aligned_storage;
```

Add **`aligned_union`** synopsis:

```
// [20.4.8] other transformations:
template <std::size_t Len, class ... Types> struct aligned_union;
```

Update **20.4.4 General requirements** §4 **[meta.requirements]**

4 Table 46 defines ~~a~~two templates that can be instantiated to define ~~a~~ types with ~~a~~ specific alignments and size.

Rewrite **20.4.8 Other transformations** **[meta.trans.other]**

Table 46: Other transformations

Template	Condition	Comments

`template <`
`  std::size_t Len,`
`  std::size_t... Alignments`
`> struct aligned_storage;`
	Len is nonzero.
	
	At least one alignment-value (3.11) is provided.
		The member type **`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 any of the *Alignments*.
		
		The static member **`alignment_value`** shall be an integral constant of type

		std::size_t whose value is the weakest alignment that satisfies all alignment requirements listed in *Alignments*.

`template <`
`  std::size_t Len,`
`  class ... Types`
`> struct aligned_union;`
	At least one type is provided.	The member type **type** shall be a POD type suitable for use as uninitialized storage for any object whose type is listed in *Types*, as long as *Len* provided is zero or the types size is at most *Len* bytes.
		
		The static member **alignment_value** shall be an integral constant of type **std::size_t** whose value is the weakest alignment that satisfies the alignment requirements of all types listed in *Types*.

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

```
template <std::size_t Len, std::size_t... Alignments>
struct aligned_storage
{
  static const std::size_t alignment_value=N;
  struct type {
    unsigned char alignas(N) __data[Len];
  };
};
```

Where *N* is the weakest alignment value that satisfies all the alignment requirements listed in *Alignments*. In other words: *N* is the Least Common Multiple of all *Alignments*.
– *end note*]

2 What – if any – extended alignments are supported by this templates is implementation defined.

3 If **aligned_union** is supplied with zero *Len* it will use the length of the largest of *Types* (**sizeof**) for the size of the member type **type**.

4 [*Note:* A typical implementation of the **aligned_union** template will publicly inherit from an instance of the **aligned_storage** template and use variadic template argument techniques to supply it with a non-zero *Len*, and the *Alignments* list that is created by applying the **alignof** operator to all *Types*. – *end note*]

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);
```

Extend **20.6.1.1 Allocator members** §5 **[allocator.members]**

[*Note:* It is implementation defined what – if any – over-aligned types are supported by the standard allocators and containers. – *end note*]

Extend **20.6.3 Temporary buffers** **[temporary.buffer]**

5 [*Note:* It is implementation-define what – if any – over-aligned types are supported by these function templates. – *end note*]

Update **20.6.7 C Library** §6 **[c.malloc]**

6 The contents are the same as the Standard C library header **<string.h>**, with the change to **memchr()** specified in 21.4, and the addition of **stdalign()** as specified in 20.6.7.1.

SEE ALSO: ISO C clause 7.11.2.

Add subclause **20.6.7.1 Stdalign** **[c.stdalign]**

```
extern "C" stdalign(
  std::size_t alignment,
  std::size_t size,
  void **pptr,
  std::size_t *pspace
);
```

1 The **stdalign()** function is the C signature-compatible counterpart of the **std::align()** function (20.6.8). Calling **stdalign(***alignment*, *size*, *pptr*, *pspace***)** is identical to calling **std::align(***alignment*, *size*, ***pptr*, ***pspace***)**.

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

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

1 *Effects:* If it is possible to fit *size* bytes of storage aligned by *alignment* into the buffer denoted by *ptr* and *space* the function updates *ptr* to point to the first possible address of such a storage and decreases *space* by the amount of bytes used for alignment and *size*. Otherwise the function has no effects.

2 *Requires:*

- alignment to be a fundamental alignment-value or an extended alignment-value supported by the implementation in this context
- *size* and *space* is not larger than the largest representable positive value by the type **std::ptrdiff_t**
- *ptr* is pointing to at least *space* bytes contiguous storage

3 *Returns:* null-pointer if the function had no effect, otherwise the updated 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*]

Extend **A.4 Expressions** **[gram.expr]**

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
*** & + - ! ~**

Extend **A.7 Declarators** **[gram.decl]**

declarator:
direct-declarator
ptr-operator declarator

direct-declarator:
alignment-specifier-list$_{opt}$ *declarator-id*
direct-declarator **(** *parameter-declaration-clause* **)** *cv-qualifier-seq*$_{opt}$ *exception-specification*$_{opt}$
direct-declarator **[** *constant-expression*$_{opt}$ **]**
(*declarator* **)**

ptr-operator:
***** *cv-qualifier-seq*$_{opt}$
&
&&
::$_{opt}$ *nested-name-specifier* ***** *cv-qualifier-seq*$_{opt}$

cv-qualifier-seq:
cv-qualifier cv-qualifier-seq$_{opt}$

cv-qualifier:
const
volatile

declarator-id:
id-expression
::$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name*

alignment-specifier-list:
alignment-specifier