Doc No: SC22/WG21/**N1877**
J16/**05-0137**
of project JTC1.22.32

Date: 2002-05-02 to 2005-08-29

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 Executive summary

Document status: proposal.

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

Related issues not addressed:

- Class-type "packing"
- 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, the `align_union` operator (for discriminated unions)
- New: standard function for pointers for proper alignment runtime

2 The Problems

Dynamic memory allocation might need many CPU cycles and scale badly to multithreaded environments. The alignment related new features make it unnecessary to dynamically allocate memory only to make it well-aligned for a given type T or a list of types. Special alignment requirements cannot be portably specified today. With the proposed features it will be possible to create well-aligned buffers, which are able to hold a given type T (or even a list of types). Close-to-hardware variables – specially aligned by the requirements of the used hardware – are also (optionally) supported by allowing implementation defined alignment-values.

2.1 Fixed capacity, dynamic size containers

Without core-language support for alignment, the library solutions can only deliver partial and partially efficient solutions – and do that at the cost of duplicating knowledge, which is already in the compiler.

2.2 Support for special, aligned types

Programmers dealing with special alignment requirements today have to use non-standard extensions to achieve their goal. The problem with non-standard extensions is that they force programmers to revert to macro magic or duplication of code to make them portable (between

compilers). This kind of code shows up mostly in embedded and close-to-the-hardware code (like drivers).

2.3 Discriminated unions

Need often arises to be able to store a list of types within a special “container”, only holding one kind (type) of object at any given time. Only approximating solutions exist today (some requiring heavy traits support) and they cannot avoid the unneeded dynamic memory allocation all the time.

2.4 Conclusion

The extensions I propose add **support for generic programming, library building** and **systems programming,** since they enable performance without excess memory usage. They also **remove the embarrassment** of not being able to make the most optimal code – with the least effort necessary. These are 4 out of the 7 motivators for change. **This proposal does not affect the type system, name lookup or binary compatibility in any ways.**

3 The Proposal

3.1 Basic Cases

3.1.1 Adding the *alignment-specifier* to the language

The *alignment-specifier* can be used in variable declarations just like a *storage-class-specifier*. It can be used portably to specify the alignment requirement for the variable being declared.

The required alignment can be specified by a type or an integral constant, called *alignment-value*. The former is called *type-based* while the latter is called *value-based alignment-specifier*:

alignment-specifier:
type-based-alignment-specifier
value-based-alignment-specifier

The *alignment-specifier* does not become part of the type (just like the *storage-class-specifier*).

Listing 1)

```
// The alignment-specifier does not change the type:
template <typename T> tfunc( T const &t) { ... }

void func( int  const &i):
void func( long const &i):

// ...

int align_by<long> aligned_var;

func(aligned_var); // Calls funct(int const &)
tfunc(aligned_var); // Instantiates tfunc<int>( int const &)
```

If the *alignment-specifier* would weaken the alignment requirement of the type, the program is ill-formed and diagnostics are required. The alignment requirement relations of related fundamental types have to be defined in the standard, just like as requirements for size relations are in the current standard. Every possible alignment has to be stronger or equal to the alignment requirements of the char types. This ensures that char (unsigned char) buffers can have any alignment specifier.

Listing 2)

```
// The following line will emit diagnostics and the program becomes ill formed
int align_by<char> aligned_var;

// The following line must be OK for any imaginable type T:
char align_by<T> aligned_buffer[sizeof T];
```

See additional requirements for *type-based* aligned variables in the subsection.

The *alignment-specifier* will cause the given variable to be aligned according to the specification.

Listing 3)
```
// Make a buffer to store 50 variables of a type as an array:
unsigned char align_by<T> buffer[50*sizeof(T)];

// Make a type T in the first element:
T *firstT = new (buffer) T;  // Safe, buffer is well-aligned for T

// Make a T in the 42th element of the array
T *raw = static_cast<T*>(buffer);
T *theAnswer = new (raw+42-1) T;
```

The *alignment-specifier* can be applied to any variable declaration including member variables of class types. It cannot be used on declarations of function arguments or catch arguments.

Listing 4)
```
// The following lines are OK
unsigned char align_by<T> buffer[50*sizeof(T)];

template <class T, std::size_t S> class vectorRudiment {
  char align_by<T> buffer_[S*sizeof(T)];
};

if (bool align_by<double>b(1)) {  // Makes little sense, but allowed and will be aligned
  // ...
}

// The following lines are ill-formed
void func( int align_by<double> i); // aligment-specification in parameter-declaration
try {
  doSomething();
} catch (int align_by<double> &i) { // aligment-specification in exception-declaration
  // Handle exception
}
```

The *alignment-specifier* is optional in declarations of variables using the extern *storage-class-specifier*. This gives the opportunity to hide this special requirement from the users of library code.

Listing 5)
```
// Somewhere in a header file I use
extern char force[];

// Somewhere in a library implementation far far away...
class Force : DarkSide {
  Jedi *good; // etc.
};
char align_by<Force> force[sizeof(Force)];
```

The *alignment-specifier* does not become part of the type, but it is possible to create a class type with aligned member variable(s).

Listing 6)
```
// Wrong attempt:
typedef double align_by<0x10000> hwDoubleVector;  // Error!
Void clear(hwDoubleVector &toClear, unsigned size);

// Using C++, why not make a class?
template class <std::size_t S> hwDoubleVector {
 double align_by<0x10000> vec_[S];
 // etc.
};
```

3.1.1.1 Alignment by type

The *type-based-alignment-specifier* takes the form of:

type-based-alignment-specifier:
align_by<*type-id*>

The type used in a *type-based-alignment-specifier* has to be complete.

The variable will be well aligned for the given *type-id*. If the architecture does not require strict alignment the *type-based-alignment-specifier* will use the usual optimal alignment of the given *type-id*.

A variable declared this way must contain enough space to store at least one instance of the type used in the *alignment-specifier*. Otherwise the program is ill-formed and diagnostics are required.[1]

Listing 7)
```
// This will not compile
char align_by<T> buff[sizeof(T)/2];

// But this will
char align_by<alignof(T)> buff[sizeof(T)/2];
```

3.1.1.2 Alignment by alignment-value

The *value-based-alignment-specifier*[2] takes the form of:

value-based-alignment-specifier:
`align_by<`*alignment-value*`>`

alignment-value:
constant-integral-expression-representable-by-std::size_t

The value can be one of an `alignof(T)` expression, 0, 1 or a value defined by the implementation.

- The value of zero does not change the alignment.
- The alignment-value 1 represents the alignment requirements of the type `char`
- A value of the `alignof(T)` expression represents the alignment requirements of type `T`.
- The effect of any other value is implementation defined.[3]

Listing 8)
```
// Aligning the buffer using a number:
template <std::size_t A, std::size_t S> class dyn_array_allocator {
  ...
  char align_by<A> buff_[S];
};
```

Using any other value renders the program ill-formed. Diagnostics is required.

Listing 9)
```
// Aligning the buffer using a number not known by the implementation:
char align_by<11> buff_[S];  // Error!  If 11 is not defined by the impl.
```

3.1.2 Getting the alignment-value

The `alignof` operator retrieves – during compile time - the alignment-value associated with a type. It is a *unary-expression* and takes the following form:

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

Its value is an integral *constant-expression* of type `std::size_t`. That value is 0, 1 or an implementation defined value. It represents the required or – if there is none – the optimal alignment for the given type.

Listing 10)
```
// Saving alignment value of a type for later use whithout the type being known
const std::size_t anAlignment = alignof(int);

// somewhere, in a Galaxy far, far away, a buffer is created to hold the unknown type:
char align_by<anAlignmen> buff_[SomeConst];

// Or in the secret chambers of some template metaprogramming genius:
template <typename T> struct magic {
  enum { value = alignof(T) };
};
```

[1] It makes no sense, so it is better caught at compile time. If it is necessary to make such a declaration, the *value-based alignment-specifier* form can be used.
[2] Value based alignment is added to support alignment-arithmetic and to enable special alignment for HW.
[3] Implementations are encouraged to define this value based on number of bytes – if it makes sense on the given platform.

3.1.3 Alignment arithmetic

The proposed `align_union` operator[1] calculates the union of two alignments, given either as alignment-values or types. The union of two alignments is the smallest alignment-value, which satisfies the alignment requirements represented by both arguments. If such a number cannot be represented using the type `std::size_t`, the program is ill-formed and diagnostics is required.

If both arguments of the operator are either compile-time constants or types, the value of the operator-expression is also a compile-time constant.

```
align_union(TorA1,TorA2) -> alignment-value
```

Examples:

Listing 11)

```
// Base for discriminated union for std::string and std::map
static const std::size_t aval  = align_union(std::string, std::map);
static const std::size_t asize =
  sizeof(std::map)>sizeof(std::string)? sizeof(std::map):sizeof(std::string);

// And the buffer, which can hold both:
char align_by<aval> buff[asize];
```

3.1.4 Runtime pointer alignment

Aligning a pointer's value means increasing it to the closest value that is well aligned for a given type T or for a given alignment-value. I propose a function called std::align for this purpose to be place in the memory standard header. The function does value-based alignment. Type based alignment can be achieved by using the **`sizeof`** and **`alignof`** operators together for specifying the `align_val` and the `size` arguments. It aligns a void pointer within a given buffer. It also checks if the aligned pointer plus the given size will fit into the buffer.

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

Aligns `ptr` using value-based alignment based on `align_val`, to form an aligned `size` bytes buffer.

Parameters:

- `align_val` alignment-value as described in 4.1.1.2 above (*)
- `ptr` the pointer to be aligned
- `space` the number of bytes left in the buffer (**)
- `size` the number of aligned bytes intended to be used after the pointer is aligned

(*) The value of zero and one means no alignment is done.

(**) Set on successful alignment.

The return value of the function is a null pointer if the aligned pointer itself or the `size` bytes of buffer would not fit into the `space` bytes available to use at the address give in the `ptr` pointer. Otherwise the aligned pointer is returned.

The adjustment increases the value of `ptr` – if needed – to make it address a memory area properly or optimally aligned for the given type T, and returns that pointer value if it is valid.

Upon success (not returning NULL) the function updates the `space` argument, it is decreased by the number of bytes used up when moving `ptr`.

[1] The intended purpose is to support discriminated unions.

The function deliberately does not return a pointer to T, since there is no T there yet, until constructed or initialized.

Listing 12)

```
char *ptr;  std::size_t space;
// Want to make 4 doubles there
void *alignedPtr = std::align( alignof(double),
                               ptr, space,
                               4*sizeof(double));
/* ptr == static_cast<char *>(alignedPtr)+size;
   std::size_t tmp = static_cast<char*>(ptr)-static_cast<char*>(orig_ptr);
   space == orig_space - (tmp + size);                                        */
if (!alignedPtr) {
   // realocate, throw, abort, scream...
}
double *dblPtr = static_cast<double*>(alignedPtr); // Safe to do it
dblPtr[0] = 42.00; dblPtr[1] = 3.14; dblPtr[2] = 2.71; dblPtr[3] = 0.00;

// Then 10 of class type T
alignedPtr = std::align( alignof(T),
                         ptr, spaceleft,
                         10*sizeof(T));
if (!alignedPtr) {
   // realocate, throw, abort, scream...
}
```

The function does not check the validity of its arguments. Calling the function with invalid arguments results in undefined behavior.[1]

4 Interactions and Implementability

4.1 Interactions

The proposed features have a loose connection to the rest of the language. Hence they do not require any change in those, and they can be considered independently. However the power of portable alignment features can best be used together with templates.

4.1.1 Effects on syntax of the language

The extensions affect two major syntactic elements of the language: variable declaration and definition (*decl-specifier)* and expressions (*unary-expression*), by adding new elements.

4.1.2 Effects on the type system

4.1.2.1 Not part of the type

An alignment specifier may affect the “placement” of the variable it is applied to, but does not change its size and does not create a new type.

In declarations of variables – which are not also definitions – the *alignment-specifier* can be omitted, as long as it is present in the definition. See section 3.1.1 above for examples.

[1] The pointer is invalid, if space is not really present or if the alignment value is invalid

4.1.2.2 Effects on class types

If an *alignment-specifier* is applied to a non-static member variable declaration in a class declaration, this alignment specification becomes part of the class type. It may change the layout, the size and the alignment requirements of the class type compared to one without that alignment specifier.

Listing 13)

```
class TouristClassSeat {
   Seat seat_;
   // ...
};
class FirstClassSeat {               // sizeof FirstClassSeat >= sizeof TouristClassSeat
   Seat align_by<HumanBeing> seat_;
   // ...
};
```

If an *alignment-specifier* is applied to a static member variable declaration it does not change the layout, size or the alignment requirements for the objects of such class. The alignment specifier can be omitted from the declaration of the static member, as long as it is present in its definition.

Listing 14)

```
class A {  // In the header file
   static char buff[sizeof(double)*42]; // In the header no alignment-specification
};

// In the implementation file
char align_by<double> A::buff[sizeof(double)*42]; // Implementation
```

4.1.3 Strength of alignment relations need to be specified

Strength of alignment is to be defined similarly to how sizes of types are defined in C++. The type `char` has the weakest alignment requirements of all, followed by short and so on.

4.2 Implementability

I feel that most of the core issues needed to implement of this proposal are present in some form in all compilers. If compiler intermediate code contains still types and not only objects of no type (addresses and sizes) the implementers need to add support for expressing different alignment requirements.

Most of the change must come into parsing and code generation. For non-member variables the code generator needs to generate code to align the variable properly according to the specifier. For member variables the class types layout needs to be generated according to the specified alignment requirements.

For the alignment operator compiler implementers need to uncover and document their internal alignment values as well as change them to byte based, if this is possible.

5 *Possible enhancements*

5.1 Provide a unique type for the alignment value?

Should we introduce a new type for this value and the `alignof(T)` expression? It seems that `std::size_t` is just about right for the task, but it leaves me with a bad feeling of coupling two absolutely unrelated things; probably creating even more confusion about alignment. Introducing a new, distinct fundamental type for this task seems to be a good idea (as it would allow overloading), however there is no precedence of that.