Querying the Alignment of an Object [Aligning Alignment]

Document #: P2152R1
Date: November 30, 2021
Project: Programming Language C++
Audience: EWGI, SG22
Reply-to: Inbal Levi
<sinbal2l@gmail.com>

1 Introduction

Alignment specification presents a challenge since it is on the junction of a language feature, implementation detail, and the hardware.

The alignment of an object or type can be specified in C++ by using the 'alignas(x)' specifier (where x is a power of 2), currently only a type's alignment can be queried by the 'alignof' operator (7.6.2.6 [expr.alignof]) (issue filed by Steve Clamage)[10].

Since the common compiler extensions' implementations allow alignment attributes to be applied to objects, and since existing practice (with those extensions) permits querying the alignment of objects, this paper suggests to allow it in the Standard of C++ as well. The paper also discusses a few open issues regarding alignment specification.

2 Motivation and Scope

C++11 introduced alignment control and query capabilities through a paper from 2007 [11].

Unfortunately, the current 'alignof' and 'alignas' operators' behaviour is inconsistent between different compilers, and between C's and C++'s struct.

In essence, I suggest the following code be ill-formed (as is its C equivalent, due to Error 2):

```
typedef struct alignas(32){
} U;

typedef struct alignas(16){     // Error 1: weaker alignment of object than its members' alignment
    U u;
} V;

int main() {
    alignas(16) U u;            // Error 2: weaker alignment of object than the alignment of its type
}
```

I suggest that the following code will be well-formed, with alignof(u) == 64, alignof(V) == 32:

```
typedef struct alignas(16) {
} U;

typedef struct alignas(32) {     // OK
    U u;
} V;

int main() {
    alignas(64) U u;
    alignof(u);                 // Not valid in C++20: only alignof(type) is allowed.
    alignof(V);                 // Not addressed in the standard: alignment of type with aligned members.
}
```

3 Definitions

The (relevant) definitions from the C++ standard regarding the alignment attribute are as follows:

1. Fundamental alignment: 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)
2. Extended alignment: An alignment greater than alignof(std::max_align_t).
3. Over-aligned type: A type having an extended alignment requirement.
4. Stricter alignment: An alignment with a greater value.
5. Weaker alignment: An alignment with a lesser value.
6. Trivially-copyable type [class.prop/1][1]
7. Standard-layout class [class.prop/3][2]

4 Proposed Changes: Suggested Design

In order to qualify the alignment of an expression, a few issues described below need to be resolved. The design in this section relies on the following assumptions:

1. The alignment is not part of the type system. As a result, the alignment of an object shouldn't apply as a parameter for the overloading mechanism.
2. The alignment of a type should be resolved by all the different limitations which are applied by its declarations as well as its definition (including hardware limitations, if such exist).
3. The alignment of an object can't be weaker than the alignment of its object type. This results from section 6.7.6 Alignment [basic.align/1]:

   ```
   [...] An object type imposes an alignment requirement on every object of that type; stricter
   alignment can be requested using the alignment specifier (9.12.2)
   ```

Section [4.1] describes incompatibility issues with C. There are additional topics derived from and additional to those changes, all changes are described in section [5]

4.1 The alignment of an expression

Consider the following C code: (https://godbolt.org/z/YnEhz8vxc)

```
struct U {
    _Alignas(32) int x;
};

int main() {
    _Alignas(16) struct U u;    // Gcc, Clang: both compilers throw error
    _Alignas(64) struct U v;    // OK
    _Alignof(v);                // Clang: warning: '_Alignof' applied to an expression is a GNU extension.
                                // Gcc, Clang: alignof(v) == 64
}
```

And its equivalent C++ code: (https://godbolt.org/z/nGzben5Mc)

```
struct U {
    alignas(32) int x;
};

int main() {
    alignas(16) struct U u;     // Gcc ,MSVC: allow specifying weaker alignment than of type (bug), Clang: error
    alignof(u);                 // Gcc: alignof(u) == 16 (bug), Clang: failed in prev line, MSVC: error. [4.1.1]
    alignas(64) struct U v;     // OK (with the existing standard specification)
    alignof(v);                 // Clang: warning: 'alignof' applied to an expression is a GNU extension.
                                // Gcc, Clang: alignof(v) == 64, MSVC: error. [4.1.1]
}
```

The issue presented in the example is as follows:

- Issue [4.1.1]: The standard does not allow alignof(exp) (this is a GNU extension). However, there are multiple compiler implementations allowing alignof(exp). As a result, there is inconsistency between different compilers.

Proposed changes:

- Issue [4.1.1]: Standardize existing practice - allow 'alignof(*id-expression*)' (TODO: Consider only allowing for an object?) and
- Issue [4.1.2]: Proposed values for 'alignof(id-expression)':
  1. For expression t, which yields an object of type T:
     (a) The value x stated in 'alignas(x) T *t*;' if exists, or
     (b) The value of 'alignof(decltype(*t*))'
  2. Unspecified for other expressions which are not objects, i.e. bit-field (*identifier*, *id-expression* which doesn't yield type-id, data class member or a complete object, etc.) TBD

  This will affect additional features such as the alignment of a pointer and of a reference.

5 Alternatives to Proposed Design

5.1 The value of alignment of an expression [4.1.2]

The following alternative could be considered:

- **Option 1**: ‘alignof(obj)‘ equals ‘alignof(decltype(obj))‘, stricker ‘alignas‘ on the object than required by its type is ignored. (Breaking change for C, breaking Gcc’s and Clangs’s behavior).
- **Option 2**: ‘alignof(obj)‘ equals ‘alignof(decltype(obj))‘, stricker ‘alignas‘ on the object than is required by its type will result in an error. (Breaking change for C, breaking Gcc, MSVC, and Clang’s behavior)

Both options are not suggested, with the target of minimizing the behaviour differences from C and from existing practice in mind.

6 Proposed Changes: Impact On the Standard

1. In alignment definition [basic.align]:

   Add a note to [basic.align/1] claiming that the alignment can be affected by hardware: in my opinion, this is beyond the scope of the standard, however, since there is already acknowledgement of this (for example, in section atomic [10]), I suggest adding it explicitly to alignment definition as well.

2. In alignof() definition [expr.alignof]:

   Add in [expr.alignof/1] allowing the alignof(id-expression). This will result in more consistency in the standard, and align the standard with existing practice. The wording aims to limit the *id-expression* to operand type yields type-id of a complete object type and a data class member access.

   I do not, however, suggest adding any changes in the more complex issues related to alignment (for example - I suggest leaving the section on virtual inheritance as is)

7 Proposed Wording

The wording suggested are based on [N4901]

1. Changing Alignment section as follows, allowing ‘alignof(id-expression)‘

   7.6.2.6 Alignof **[expr.alignof]**

   1 An alignof(*id-expression*) yields the alignment requirement of its operand type, if the operand is a type-id representing a complete object type, or a data class member access expression (7.6.1.4) of a complete object or an array thereof, or a reference to one of those types.

8 Changes

8.1 R1

- Implement corrections sent by Jens Maurer and other Core wording experts: Remove section [4.2] to Appendix A. Move wording from [dcl.align] to [basic.align]. Add issue [4.1.2], specifying the value of ‘alignof(id-expression)‘
- Implement corrections sent by Richard Smith: Remove first issue (formerly known as issue (1)) from section (4.1). Remove first issue (formerly known as issue (1)) from section (4.1)
- Remove first issue (formerly known as issue (1)) from section (4.2), Add example remarks and detailed explanations on section (4.2).
- Change "Alternatives to Proposed Design" and "Wording" sections accordingly.

9 Acknowledgements

- **Jens Maurer** For providing useful feedback and wording.
- **Richard Smith** For providing helpful detailed feedback.
- **Daveed Vandevoorde** For reviewing this paper in detail at different stages, providing guidance and useful corrections and directions.
- **Attila Feher** For providing useful and detailed feedback in addition to detailed examples.
- **JF Bastien** For reviewing the draft, adding his directions for a better cohesion with core.
- **Michael Wong** For reviewing the draft, adding his valuable corrections and input.

10 Appendix A: Additional topics with no suggested changes

10.1 The alignment of an object with aligned members (Section [4.2] in R0)

The alignment of a struct in C is resolved to the strictest amongst its members [6.7.5][6].

Consider the following C code: (https://godbolt.org/z/wtJvS_)

```
struct V {} __attribute__((aligned (64)));
struct S {} __attribute__((aligned (32)));
struct U{
    S s;
    V v;
} __attribute__((aligned (16)));        // This alignment is ignored in both gcc and clang

int main() {
   alignof(U);                          // alignof(U) is valid, and equals 64
}
```

And its equivalent C++ code: (https://godbolt.org/z/5Wx-V2)

```
typedef struct alignas(64) V {} V;
typedef struct alignas(32) S {} S;
typedef struct alignas(16) U{           // Gcc: ignored, Clang: error, MSVC: warning. [10.1.1]
    S s;
    V v;
} U;

int main() {
   alignof(U);                          // When compiles, alignof(U) equals 64. [10.1.2]
}
```

A section with the example which is described here exists in the standard [dcl.align/5] [5], yet a few specifications are missing here:

- Issue [10.1.1]: There is no rule suggesting that the alignment of an object, whose members have alignment requirements, is restricted by it. (Although, in section [basic.align/2][4], in the example's explanation, it's assumed that the struct's alignment is restricted by its members' alignment)
- Issue [10.1.2]: There no specification for how the alignment of the members restricts the alignment of the type. (alignment(type) = max(alignment(m_a),alignment(m_b),...) makes sense, but not explicitly specified, and there are other options)

This exposes a broader issue - alignment is a topic which is controlled by three separate domains, and resolved on three different layers:

1. Software: By the 'alignas(x)' attribute, which (assuming we accept the proposed in 4.1) can be specified on both the object type, the object, and its members.
2. (Compiler) Implementation: By the compiler, building the memory storage for the object. (Itanium C++ ABI is addressing this topic, under [Chapter 2: Data Layout][14]

3. Hardware: Alignment specification is used to improve performance by achieving control on the object layout in memory. Naturally, different hardware will affect the resulting performance.

Focusing on the first and second, my view is that an explicitly specified alignment should not be ignored. Therefore, I suggest that specifying an alignment of an object which conflicts with its members' alignment should be ill-formed (behaviour same as clang's). Two possible directions:

1. Specify the alignment of an object is affected by its members and resolved to the strictest among its members, and assume an implementation allocates continuous memory for the object as in Itanium ABI-based implementation ('alignof' evaluated as if it was a complete object [basic.align/2][4]).
2. Specify the alignment of an object is affected by its members and remain vague about implementation, which, in turn, might result with the same code being well-formed or ill-formed depending on different compilers' implementation and hardware.

My preference is to assume the Itanium implementation.

10.1.1 Proposed change:

- Issue [10.1]: None.

  I initially proposed to add that describing a weaker alignment for an object than is required by its members will result in an error. (Stricter than C, aligned with Clang, breaking Gcc's and MSVC's behavior), but got feedback from Core wordsmith this is already covered by [dcl.align/5][5].

10.1.2 Alternative Designs

For issue [10.1], I initially proposed (Option 1), but after consulting with Core wordsmith, I did not suggest a change.

The following alternatives could be considered:

- **Option 1**: Add that describing a weaker alignment for an object than is required by its members will result in an error. (Stricter than C, aligned with Clang, breaking Gcc's and MSVC's behavior). Add that the value of 'alignof(obj)' equals 'alignof(decltype(obj))'.
- **Option 2**: Add that describing a weaker alignment for an object than is required by its members will result in a warning. (Aligned with C (with the addition of a warning), aligned with MSVC, breaking Clang's behavior)
- **Option 3**: Add that describing a weaker alignment for an object than is required by its members will be ignored. (Aligned with C, breaking Clang's behavior)

The incentive for a stricter rule (Option 1) is to avoid specified instructions not executed. Since alignment is a requirement explicitly specified, I believe not implementing the alignment requirement should result in an error. In addition, since the C struct's syntax is different, it will only break C++ code, **where there is already an inconsistency on this topic. This will also increase consistency with the already existing rule - specifying a weaker alignment than the one already applied on an entity is ill-formed.**

11 References

[1] 11.2 Properties of classes [class.prop/1]

```
1 A trivially copyable class is a class:
(1.1) - that has at least one eligible copy constructor, move constructor, copy assignment operator,
or move assignment operator (11.4.4, 11.4.5.3, 11.4.6)
(1.2) - where each eligible copy constructor, move constructor, copy assignment operator, and move
assignment operator is trivial, and
(1.3) - that has a trivial, non-deleted destructor (11.4.7).
```

[2] 11.2 Properties of classes [class.prop/3]

```
3 A class S is a standard-layout class if it:
(3.1) - has no non-static data members of type non-standard-layout class (or array of such types)
or reference,
(3.2) - has no virtual functions (11.7.3) and no virtual base classes (11.7.2),
(3.3) - has the same access control (11.8) for all non-static data members,
(3.4) - has no non-standard-layout base classes,
(3.5) - has at most one base class subobject of any given type,
(3.6) - has all non-static data members and bit-fields in the class and its base classes first
declared in the same class, and
(3.7) - has no element of the set M(S) of types as a base class, where for any type X, M(X) is
defined as follows.97

...
```

[3] 6.7.6 Alignment [basic.align/1]

```
1 Object types have alignment requirements (6.8.1, 6.8.2) which place restrictions on the addresses at
which an object of that type may be allocated. An alignment is an implementation-defined integer value
representing the number of bytes between successive addresses at which a given object can be allocated.
An object type imposes an alignment requirement on every object of that type; stricter alignment can be
requested using the alignment specifier (9.12.2).
```

[4] 6.7.6 Alignment [basic.align/2]

```
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) (17.2).
The alignment required for a type might be different when it is used as the type of a complete object
and when it is used as the type of a subobject. [Example:

struct B { long double d; };
struct D : virtual B { char c; };

When D is the type of a complete object, it will have a subobject of type B, so it must be aligned
appropriately for a long double. If D appears as a subobject of another object that also has B as a
virtual base class, the B subobject might be part of a different subobject, reducing the alignment
requirements on the D subobject.  - end example]

The result of the alignof operator reflects the alignment requirement of the type in the complete-
object case.
```

## [5] 9.12.2 Alignment specifier [dcl.align/5]

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 be required for the entity being declared if all alignment-specifiers appertaining to that entity were omitted. [Example:

```
    struct alignas(8) S{};
    struct alignas(1) U{         // This declaration should not be allowed
    S s;
    };  // error: U specifies an alignment that is less strict than if the alignas(1) were omitted.
```

end example]

## [6] 6.7.5 Alignment specifier [from C standard]

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 type of the object or member being declared.

## [7] 9.12.2 Alignment specifier [dcl.align/6]

6 If the defining declaration of an entity has an alignment-specifier, any non-defining declaration of that entity shall either specify equivalent alignment or have no alignment-specifier.
Conversely, if any declaration of an entity has an alignment-specifier, every defining declaration of that entity shall specify an equivalent alignment. No diagnostic is required if declarations of an entity have different alignment- specifiers in different translation units. [Example:

```
    // Translation unit #1:
    struct S { int x; } s, *p = &s;
    // Translation unit #2:
    struct alignas(16) S; // ill-formed, no diagnostic required: definition of S lacks alignment
    extern S* p;
```

end example]

## [8] 7.6.2.6 Alignof [expr.alignof/3]

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

## [9] 31.7.2 Operations [atomics.ref.ops/1,2]

static constexpr size_t required_alignment;
1 The alignment required for an object to be referenced by an atomic reference, which is at least alignof(T).
2 [Note: Hardware could require an object referenced by an atomic_ref to have stricter alignment (6.7.6) than other objects of type T. Further, whether operations on an atomic_ref are lock-free could depend on the alignment of the referenced object. For example, lock-free operations on std::complex<double> could be supported only if aligned to 2*alignof(double).
  - end note]

[10] CWG closed issues *CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, and SGI. Additional contributions were provided by a variety of individuals*
http://wiki.edg.com/pub/Wg21summer2020/CoreWorkingGroup/cwg_closed.html#1008

[11] Adding alignment support to the C++ programming language / wording (2007) *Attila (Farkas) Fehér, Clark Nelson*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2341.pdf

[12] C and C++ Alignment Compatibility (2010) *Lawrence Crowl, Daveed Vandevoorde*
http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1507.htm

[13] Dynamic memory allocation for over-aligned data (2016) *Clark Nelson*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0035r4.html

[14] Itanium C++ ABI - Chapter 2: Data Layout (March 2017)
https://itanium-cxx-abi.github.io/cxx-abi/abi.html#layout

[15] Current C draft (October 2021)
http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2731.pdf

[16] Current C++ draft (October 2021)
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/n4901.pdf