Querying the Alignment of an Object [Aligning Alignment]

Document #: P2152R0
Date: June 15, 2020
Project: Programming Language C++
Audience: EWGI
Reply-to: Inbal Levi
<sinbal2l@gmail.com>

1 Introduction

The alignment requirements of an object or type can be specified in C++ by using the alignas(x) specifier (where x is a power of 2).

The current specification of the alignof operator (7.6.2.5 [expr.alignof]) allows it to be applied only to types, not to objects. Since the alignment attribute may be applied to objects, and since existing practice permits querying the alignment of objects, it should be considered whether to allow this in Standard C++ as well.[9]

The following paper discusses design issues that need to be considered **in order to allow consistently querying the alignment of an object (vs. the object's type) using the alignof operator.**

2 Motivation and Scope

C++11 introduced alignment control and query capabilities through a paper from 2007 [10].

Unfortunately, the current alignof operator is inconsistent between different compilers, and inconsistent within the standard itself.

Further research has exposed a divergence between C's objects and C++'s objects, which I believe is the result of the similar yet different syntax for struct in the languages, resulting in the current alignment section in the standard not resolving critical issues sufficiently. These issues, as well as suggested fixes, are described in this paper.

In essence, I suggest the following code is ill-formed (as is its C equivalent, thanks 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 be well-formed, with alignof(u) == 64 and 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:

- 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)
- Extended alignment: An alignment greater than alignof(std::max_align_t).
- Over-aligned type: A type having an extended alignment requirement.
- Stricter alignment: An alignment with a greater value.
- Weaker alignment: An alignment with a lesser value.

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. As has been discussed before, as well as for aligning the behaviour with C, 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)
   ```

Sections [4.1], [4.2] describe 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 object type's declaration vs. object definition

Consider the following C code: (https://godbolt.org/z/kv9NkF)

```
typedef struct U U;
struct U {
}__attribute__((aligned (32)));

int main() {
    _Alignas(16) U u;       // Gcc, Clang: both compilers throw error
    _Alignas(64) 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/HqoFnw)

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

int main() {
    alignas(16) U u;        // Gcc ,MSVC: allow specifying weaker alignment than of type, Clang: error. (1)
    alignof(u);             // Gcc: alignof(u) == 16, Clang, MSVC: error
    alignas(64) U v;        // OK
    alignof(v);             // Clang: warning: 'alignof' applied to an expression is a GNU extension.
                            // Gcc, Clang: alignof(v) == 64, MSVC: error. (2)
}
```

The issues presented in the example are as follows:

- Issue (1): As described above, the C++ standard [14] (as well as the C standard [13]) specifies that an object type's alignment restricts the object's alignment [1], resulting in: an object can't be defined with alignas() specifier which describes alignment that is weaker than its type. However, the standard does not state the result of describing such an alignment.
- Issue (2): There are multiple references to functionality derived from allowing alignof(exp), however, the standard does not allow alignof(exp). As a result, there is inconsistency between different compilers, and between C and C++ standards.

Proposed changes:

- Issue (1): Add that describing a weaker alignment for an object than required by its object type will resolve with an error. (Aligned with C, Aligned with Clang, Breaking Gcc's and MSVC's behaviour)
- Issue (2): Standardize the existing practice - allow alignof(exp). this will affect additional features such as the alignment of a pointer and of a reference.

4.2 The alignment of an object with aligned members

The alignment of an object in C is resolved to the strictest amongst its members [6.7.5][4].

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

```
typedef struct V V;
typedef struct S S;
typedef struct U U;

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. (1)
    S s;
    V v;
} U;

int main() {
   alignof(U);                          // When compiles, alignof(U) equals 64. (2)
}
```

A section with the example which is described here exists in the standard [3], yet the wording diverges from C [6.7.5][4]. As a result, two specifications are missing here:

- Issue (1): There is no rule defining whether describing an alignment of an object which is weaker than the alignment required by its members should result in an error, a warning, or be ignored.
- Issue (2): There is no rule defining what is the alignment of an object, whose members (in the broad sense) have alignment requirements, and so this is an open issue. (Although, in section [basic.align][2], in the example's explanation, it's assumed that the struct's alignment is restricted by its members' alignment)

Proposed changes:

- Issue (1): Add describing a weaker alignment for an object than is required by its members will resolve with an error. (Stricter than C, Aligned with Clang, Breaking Gcc's and MSVC's behaviour)
- Issue (2): Restore the rule from C (replacing "type" with "entity"), this will result in: The alignment of an entity should be stricter than or equal to its members' alignment.

5 Proposed Changes: Alternative Design

5.1 The alignment of an object type's declaration vs. object definition [4.1]

For issue (1), I suggested:
Add that describing a weaker alignment for an object than required by its object type will resolve with an error. (Aligned with C, Aligned with Clang, Breaking Gcc's and MSVC's behaviour)

the following alternatives could be considered:

- **Option 1:** Add that describing a weaker alignment for an object than required by its object type will be ignored. (Breaking C behaviour, aligned with Gcc's)
- **Option 2:** Add that describing a weaker alignment for an object than required by its object type will resolve with a warning. (Breaking C behaviour, contain change for all compilers)

In both cases the suggestions are for behaviour weaker than of C's struct, and will increase the difference between the languages regarding the struct keyword, as well as ignore the alignment explicit demand, therefore I don't recommend it.

5.2 The alignment of an object with aligned members [4.2]

For issue (1), I suggested:
Add describing a weaker alignment for an object than is required by its members will resolve with an error. (Stricter than C, Aligned with Clang, Breaking Gcc's and MSVC's behaviour)

the following alternatives could be considered:

- **Option 1**: Add describing a weaker alignment for an object than is required by its members will be ignored. (Aligned with C, Breaking Clang's behaviour)
- **Option 2**: Add describing a weaker alignment for an object than is required by its members will resolve with a warning. (Aligned with C (with the addition of a warning), Aligned with MSVC, Breaking Clang's behaviour)

The incentive for a stricter rule suggested in section [4.2] is to avoid specified instructions not executed. Since alignment is a requirement explicitly specified, I believe not implementing the alignment requirement should result with an error. In addition, since the C struct's syntax is different, it will only break C++ code, **in which there is already an inconsistency on this topic.**

6 Proposed Changes: Impact On the Standard

1. In alignment definition [basic.align]:
   - Specify in [basic.align/1] that the alignment can be affected by hardware: in our opinion, this is beyond the scope of the standard, however, since there is already acknowledgement of this (for example, in section atomic [8]) I suggest adding it to alignment definition as well.

- Move the sentence from the end of [basic.align/1] referring to alignment of type vs. object type to a different bullet. Add describing a weaker alignment will result in an error. (**Aligned with C, Aligned with Clang, Breaking Gcc's and MSVC's behaviour**)

2. In alignof() definition [expr.alignof]:
   - Add in [expr.alignof/1] allowing the alignof(exp). This will result in additional standard features becoming valid (such as alignof reference [7], currently unsupported).
   - Add querying the alignment of an object declared with different alignment values in different translation units is UB. (It is already specified under [dcl.align], but it is also relevant to the alignof() expression)
3. In alignas() definition [dcl.align]:
   - Restore the [dcl.align/5] section, aligning it with C. Add the combined effect of all alignment-specifiers in a declaration shall not specify an alignment less strict than of its members (in the broad sense).
   - Moreover, I suggest to specify explicitly, that describing such an alignment will resolve with and error. (**Stricter than C, Aligned with Clang, Breaking Gcc's and MSVC's behaviour**)
4. Add minor fix: add existing practice of alignment in reinterpret_cast [6].

7 Proposed Wording

- Adding a restriction regarding stating alignment of an object weaker than its members:

6.7.6 Alignment **[basic.align]**

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. In addition to alignment requirements, the alignment of an entity can be affected by its memory location as well as the hardware type.

2 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). describing a weaker alignment will result in an error.

3 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. [...]

- Changing Alignment section as follows, allowing alignof(exp), and specifying UB for querying alignment of an entity declared with different alignments in different translation units:

7.6.2.5 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, or an unparenthesized id-expression or an unparenthesized data class member access (7.6.1.4) of a complete object, or an array thereof, or a reference to one of those types.

2 The result is a prvalue of type std::size_t. [*Note*: An alignof expression is an integral constant expression (7.7). The type std::size_t is defined in the standard header <cstddef> (17.2.1, 17.2.4). — *end note*]

3 Querying the alignment of an object type declared with different alignments in different translation units will result in undefined behavior.

4 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.

- Fix the wording in alignment specifier section, define the alignment value of an object with aligned members, and **add specifying otherwise yields an error**:

9.12.2 Alignment specifier **[dcl.align]**

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, specifying such an alignment will result in an error. Moreover, 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 {
S s;
}; // error: U specifies an alignment that is less strict than if the alignas(1) were omitted.
```

— *end example*]

- Add existing practice of alignment in reinterpret_cast [6]:

7.6.1.9 Reinterpret cast **[expr.reinterpret.cast]**

7 [...] [Note: Converting a prvalue of type pointer to T 1 to the type pointer to T 2 (where T1 and T2 are object types and where the alignment requirements of T2 are no strcter than those of T1) and back to its original type yields the original pointer value, as well as its alignment. end note]

8 Future Topics To Be Examined

Consistency of std::allocate::allocate and std::get_temporary_buffer support for object with smaller alignment than the type.

9 Acknowledgements

- **Daveed Vandevoorde** Many thanks for reviewing this paper in detail at different stages, providing guidance and useful corrections and directions.
- **Attila Feher** For reviewing initial ideas, provide very useful and detailed feedback in addition to very helpful 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.

- **Everyone on EWG, CWG** for contributing to a knowledgeable discussion on this topic.

10 References

[1] 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).
```

[2] 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 align-
ment requirement of the type in the complete-object case.
```

[3] 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]
```

[4] 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.
```

[5] 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]

[6] 7.6.1.9 Reinterpret cast [expr.reinterpret.cast/7]

7 An object pointer can be explicitly converted to an object pointer of a different type.65 When a prvalue v of object pointer type is converted to the object pointer type pointer to cv T , the result is static_cast<cv T*>(static_cast<cv void*>(v)). [Note: Converting a prvalue of type pointer to T 1 to the type "pointer to T2" (where T1 and T2 are object types and where the alignment requirements of T2 are no stricter than those of T1) and back to its original type yields the original pointer value. end note]

[7] 7.6.2.5 Alignof [expr.alignof/3]

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

[8] 31.7.1 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]

[9] 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

[10] 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

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

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

[13] Current C draft (2019)
http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2454.pdf

[14] Current C++ draft (2020)
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/n4861.pdf