Document	P1413R3
Date	2021-11-22
Author	CJ Johnson <johnsoncj@google.com>
Audience	Library Evolution Working Group (LEWG), Library Working Group (LWG)

Deprecate std::aligned_storage and std::aligned_union

Changes over Revision 2

- Added wording suggested by Jonathan Wakely
- Removed extraneous pie chart

Changes over Revision 1

- Fixed incorrect change log
- Minor edits
- At Cologne 2019, LEWG-I voted to forward the paper to LEWG

SF	F	N	A	SA
8	5	1	0	0

Changes over Revision 0

- Changed the title (Previously: "A safer interface for std::aligned_storage") and removed `std::aligned_storage_for`
- Merged the content from "Additional usage" into "Existing usage" and removed the "Additional usage" section
- Added "Suggested replacement" section
- Added "Wording" section
- Major edits
- At Kona 2019, LEWG-I voted to update the paper by removing the new proposed interface (`std::aligned_storage_for`) and instead deprecating `std::aligned_storage` and `std::aligned_union`

  Deprecate `std::aligned_storage`?

  Unanimous Consent

SF	F	N	A	SA
--	-	-	-	--

  Deprecate `std::aligned_union`?

SF	F	N	A	SA
10	0	1	0	0

Preface

For simplicity, this paper will use the term `aligned_*` to refer to all of the types that fall under the same category. This includes standard types such as `std::aligned_storage` and `std::aligned_union`, their `*_t` counterparts, Boost's many implementations from which the standard types were inspired and other utilities such as `absl::aligned_storage_t` [1] and `folly::aligned_storage_for_t` [2].

Additionally, `std::aligned_*` is used to specifically refer to those from the standard.

Proposal

`std::aligned_*` should be deprecated in the standard to make clear to users of the standard library that they are not to be used and may be removed in the future.

Background

`aligned_*` are harmful to codebases and should not be used. At a high level:

- Using `aligned_*` invokes undefined behavior (The types cannot provide storage.)
- The guarantees are incorrect (The standard only requires that the type be *at least* as large as requested but does not put an upper bound on the size.)
- The API is wrong for a plethora of reasons (See "On the API".)
- Because the API is wrong, almost all usage involves the same repeated pre-work (See "Existing usage".)

On the API

`std::aligned_*` suffer from many poor API design decisions. Some of these are shared, and some are specific to each. As for what is shared, there are three main problems:

- Using `reinterpret_cast` is required to access the value
- `::type` is not automatically resolved
- There is no upper bound on the size of `::type`

The first of these is pretty self explanatory. There is no `.data()` or even `.data` on `std::aligned_*` instances. Instead, the API requires you to take the address of the object, call `reinterpret_cast<T*>(...)` with it, and then finally indirect the resulting pointer giving you a `T&`. Not only does this mean that it cannot be used in constexpr, but at runtime it's much easier to accidentally invoke undefined behavior. `reinterpret_cast` being a requirement for use of an API is unacceptable.

The second of these problems has already been somewhat addressed with the creation of `std::aligned_storage_t` and `std::aligned_union_t` in C++14, but that doesn't stop programmers from making mistakes. Writing `std::aligned_storage<t_size, t_align> t_storage;` looks and feels right, but it's **very** wrong (missing `typename` and `::type`). In fact, because of the first problem mentioned, it's easy to not catch this second problem when it happens. `reinterpret_cast` will do what you tell it to do, even if that means taking a type that is the wrong size (`std::aligned_storage<t_size, t_align>`) and reading it through its address as if it was the type you intended (`std::aligned_storage<t_size, t_align>::type`).

The third is more of a defect in the standard, but it applies to both types and thus should be noted here. The language of the standard only says `::type` needs to be at least as big as requested. It fails to actually limit what the size can be. This has the potential to result in more memory being used than is needed (especially for arrays of `std::aligned_*` where the extra unused bytes are multiplied by the number of elements in the array).

As for what specifically plagues `std::aligned_storage`, there are two key problems:

- The template has no type arguments
- The second template argument is defaulted

`std::aligned_storage` takes in, as nontype template parameters, two arguments of `size_t`. These denote the size of the storage and the alignment of the storage. Except in very rare cases, the alignment parameter should always be in lock-step with the size parameter. If you're passing in `sizeof(T)`, you should be passing in `alignof(T)` (see "Existing usage"). In fact, this is so important, Facebook released `folly::aligned_storage_for_t` [2] for this *exact* purpose.

Not only are the template arguments not what they probably should be, but the second one actually has a default value. Nothing in the compiler is going to stop you from writing `std::aligned_storage_t<sizeof(T)>`. The second parameter, the alignment, has an implementation-defined default value that may or may not be sufficient for `T` (it's not required to support over-aligned types!). If the programmer happens to know that the default is correct for their use-case, then I suppose this is ok. But quite frankly, allowing the alignment parameter to be silently incorrect is not good API design and was a mistake.

The problems specific to `std::aligned_union` are less awful, but still not desirable:

- The leading `size_t` parameter is pointless
- Size and alignment deduction is inconsistent with `std::aligned_storage`

`std::aligned_union` takes in its template parameters a single `size_t` and then a variadic list of types. From there, it deduces the size and alignment of every type and then uses the maximum of those values for the actual storage. In addition to deducing the max size of the types provided, the leading `size_t` is the minimum size for the storage. Even if all types are smaller, the `std::aligned_union` will be no smaller than that provided minimum.

The thing is, needing a minimum size is uncommon, at best. Most of the usage is of the form `std::aligned_union_t<0, Ts...>`. Having to pass in a `0` just to meet the API is strange. If readers happen to have context and have worked with `aligned_*` before, they can likely understand why the author put it there. But if they're new to these types and aren't familiar with the API of `std::aligned_union`, it is puzzling to see. It isn't immediately clear what the programmer was intending. Do they *want* a zero-sized storage object or are they simply passing in a value to meet the awkward API?

Additionally, deducing the size and alignment automatically is nice, but it's inconsistent with `std::aligned_storage`. `std::aligned_*` types should almost certainly not have such radically different APIs. It leads to interesting uses such as `std::aligned_union_t<0, T> t_storage;` (passing in only a single type). Did the author use `std::aligned_union` because of the automatic size and alignment deduction? Or did they forsee a future where `t_storage` would be used to hold one of many possible types and so far only added the first one? Since the APIs of `std::aligned_*` are different, the intent of the author in this example becomes unclear.

Existing usage

NOTE: For this section, `aligned_*` will just refer to `std::aligned_storage` and friends (not `std::aligned_union`).

`aligned_*` types are niche, being most useful for container types provided by utility libraries. Thus, to find how they are used in practice, data was scraped from the implementation details of Folly, Boost and Abseil.

Across these three libraries, there are 95 distinct uses. Nearly **73%** (69/95) of those uses are of the form `aligned_storage<sizeof(T), alignof(T)>` (or some alternate spelling thereof). The vast majority of `aligned_*`'s use is simply repeating the same process of taking a type and deducing its size + alignment manually to meet the API. [3]

By library it's **89%** (8/9) for Abseil, **58%** (11/19) for Folly, and **75%** (50/67) for Boost (though notably this number increases to **88%** (23/26) for Boost.Container). All three libraries slightly or overwhelmingly favor the sizeof-alignof-T pattern when using `aligned_*`. [3]

In addition to the main utility libraries of the world, the sizeof-alignof-T pattern appears in most of `aligned_*`'s usage across the internet. The CppReference `static_vector` example for `std::aligned_storage` features the fullly spelled out version as `typename std::aligned_storage<sizeof(T), alignof(T)>::type`. [4] Comments on StackOverflow, when using `aligned_*`, commonly follow the pattern, as well. [5][6][7][8] The pattern appears in wiki pages [9], blog posts [10][11][12] and in the source of other libraries [13] and languages [14].

It seems that the reasonable conclusion to draw from this data is that the API was designed incorrectly. The primary usecase of "I need a type that can provide storage for this `T`" is not being well satisfied by `std::aligned_*`.

Suggested replacement

The easiest replacement for `aligned_*` is actually not a library feature. Instead, users should use a properly-aligned array of `std::byte`, potentially with a call to `std::max(std::initializer_list<T>)`. These can be found in the `<cstddef>` and `<algorithm>` headers, respectively (with examples at the end of this section).

Unfortunately, this replacement is not ideal. To access the value of `aligned_*`, users must call `reinterpret_cast` on the address to read the bytes as `T` instances. Using a byte array as a replacement does not avoid this problem. That said, it's important to recognize that continuing to use `reinterpret_cast` where it already exists is not nearly as bad as newly introducing it where it was previously not present.

It's reasonable to then conclude that perhaps the standard library should provide a typedef of such a byte array, making it easier to switch. Such as this:

```
namespace std2 {
template <typename T>
using aligned_storage = alignas(T) std::byte[sizeof(T)];
}
```

Unfortunately, the `alignas` attribute is ignored on typedefs, since it can only be applied to objects. To demonstrate this:

```
template <typename T>
using unaligned_storage = std::byte[sizeof(T)];

// They're the same type!
static_assert(
  std::is_same_v<
    unaligned_storage<double>,
    std2::aligned_storage<double>
  >
);
```

Since users *must* apply `alignas` themselves, there doesn't seem to be much value in providing a typedef like `unaligned_storage` in the standard, seeing as all it can deduce is the correct size.

With that, here are example replacement diffs in line with the suggestion of this paper:

```
  // To replace std::aligned_storage...

  template <typename T>
  class MyContainer {
    // [...]

   private:
-   std::aligned_storage_t<sizeof(T), alignof(T)> t_buff;
+   alignas(T) std::byte t_buff[sizeof(T)];

    // [...]
  };
```

```
  // To replace std::aligned_union...

  template <typename... Ts>
  class MyContainer {
    // [...]

   private:
-   std::aligned_union_t<0, Ts...> t_buff;
+   alignas(Ts...) std::byte t_buff[std::max({sizeof(Ts)...})];

    // [...]
  };
```

Wording

Modify the synopsis in **[meta.type.synop]**:

```
  // [meta.trans.other], other transformations
  template<class T> struct type_identity;
- template<size_t Len, size_t Align = default-alignment> // see [meta.trans.other]
-  struct aligned_storage;
- template<size_t Len, class... Types> struct aligned_union;
  template<class T> struct remove_cvref;
```

```
  template<class T>
   using type_identity_t = typename type_identity<T>::type;
- template<size_t Len, size_t Align = default-alignment> // see [meta.trans.other]
-  using aligned_storage_t = typename aligned_storage<Len, Align>::type;
- template<size_t Len, class... Types>
-  using aligned_union_t = typename aligned_union<Len, Types...>::type;
  template<class T>
   using remove_cvref_t = typename remove_cvref<T>::type;
```

Remove the `std::aligned_storage` and `std::aligned_union` rows from table **[tab:meta.trans.other]**.

Modify the synopsis in **[depr.meta.types]**:

```
  namespace std {
    template<class T> struct is_pod;
    template<class T> inline constexpr bool is_pod_v = is_pod<T>::value;
+   template<size_t Len, size_t Align = default-alignment> // see below
+     struct aligned_storage;
+   template<size_t Len, class... Types> struct aligned_union;
```

```
+   template<size_t Len, size_t Align = default-alignment> // see below
+     using aligned_storage_t = typename aligned_storage<Len, Align>::type;
+   template<size_t Len, class... Types>
+     using aligned_union_t = typename aligned_union<Len, Types...>::type;
  }
```

Add to the end of **[depr.meta.types]**:

```
+ template<size_t Len, size_t Align = default-alignment>
+  struct aligned_storage;
+
+ -?- The value of default-alignment shall be the most stringent alignment
+     requirement for any object type whose size is no greater than Len
+     ([basic.types]).
+ -?- Mandates: Len is not zero. Align is equal to alignof(T) for some type T or
+     to default-alignment.
+ -?- The member typedef type shall be a trivial standard-layout type suitable for
+     use as uninitialized storage for any object whose size is at most Len and
+     whose alignment is a divisor of Align.
+ -?- [Note 1: Uses of aligned_storage<Len, Align>::type can be replaced by an
+     array std::byte[Len] declared with alignas(Align).  --end note]
+
+ template<size_t Len, class... Types> struct aligned_union;
+
+ -?- Mandates: At least one type is provided. Each type in the template parameter
+     pack Types is a complete object type.
+ -?- The member typedef type is a trivial standard-layout type suitable for use
+     as uninitialized storage for any object whose type is listed in Types; its
+     size shall be at least Len. The static member alignment_value is an integral
+     constant of type size_t whose value is the strictest alignment of all types
+     listed in Types.
```

References

[1] `absl::aligned_storage_t`

[2] `folly::aligned_storage_for_t`

[3] Gathered data

This data was gathered in January of 2019 for revision 0 of this paper. `std::aligned_union` was not part of the paper at the time, so data about its usage was not collected. No effort was made to update the data for revision 1.

Library	Is of the form aligned_storage<sizeof(T), alignof(T)>?	Depends on default alignment?	Source
Folly	YES	NO	HazptrHolder.h:220
Folly	YES	NO	HazptrHolder.h:338
Folly	YES	NO	LifoSem.h:125
Folly	YES	NO	small_vector.h:1142
Folly	YES	NO	Replaceable.h:640
Folly	YES	NO	dynamic.h:782
Folly	YES	NO	F14Table.h:567

Library	Is of the form aligned_storage<sizeof(T), alignof(T)>?	Depends on default alignment?	Source
Folly	YES	NO	F14Policy.h:1208
Folly	YES	NO	UnboundedQueue.h:761
Folly	YES	NO	MPMCQueue.h:1442
Folly	YES	NO	AtomicUnorderedMap.h:366
Folly	NO	YES	Fiber.h:124
Folly	NO	YES	Fiber.h:164
Folly	NO	YES	Function.h:255
Folly	NO	YES	PolyDetail.h:384
Folly	NO	NO	ExceptionWrapper.h:223-224
Folly	NO	NO	small_vector.h:1154-1155
Folly	NO	NO	F14Table.h:330-332
Folly	NO	NO	Memory.h:109
Boost.Accumulators	NO	YES	droppable_accumulator.hpp:249
Boost.Asio	NO	YES	allocator.hpp:49
Boost.Beast	NO	NO	type_traits.hpp:88-90
Boost.Container	YES	NO	advanced_insert_int.hpp:131
Boost.Container	YES	NO	advanced_insert_int.hpp:154
Boost.Container	YES	NO	advanced_insert_int.hpp:291
Boost.Container	YES	NO	advanced_insert_int.hpp:401
Boost.Container	YES	NO	flat_tree.hpp:925
Boost.Container	YES	NO	flat_tree.hpp:937
Boost.Container	YES	NO	flat_tree.hpp:948
Boost.Container	YES	NO	flat_tree.hpp:960
Boost.Container	YES	NO	flat_tree.hpp:997
Boost.Container	YES	NO	flat_tree.hpp:1008
Boost.Container	YES	NO	flat_tree.hpp:1019
Boost.Container	YES	NO	flat_tree.hpp:1030
Boost.Container	YES	NO	node_handle.hpp:114-116
Boost.Container	YES	NO	varray.hpp:228-231
Boost.Container	YES	NO	varray.hpp:1072-1073

Library	Is of the form aligned_storage<sizeof(T), alignof(T)>?	Depends on default alignment?	Source
Boost.Container	YES	NO	varray.hpp:1115-1116
Boost.Container	YES	NO	varray.hpp:1625-1628
Boost.Container	YES	NO	small_vector.hpp:385-386
Boost.Container	YES	NO	tree.hpp:137-138
Boost.Container	YES	NO	list.hpp:79
Boost.Container	YES	NO	slist.hpp:84
Boost.Container	YES	NO	stable_vector.hpp:150-151
Boost.Container	YES	NO	string.hpp:176-177
Boost.Container	NO	NO	copy_move_algo.hpp:1024-1025
Boost.Container	NO	NO	copy_move_algo.hpp:1054-1055
Boost.Container	NO	NO	static_vector.hpp:73
Boost.Coroutine2	YES	NO	pull_control_block_cc.hpp:33
Boost.Coroutine2	YES	NO	pull_control_block_cc.hpp:74
Boost.Fiber	YES	NO	shared_state.hpp:160
Boost.Fiber	YES	NO	buffered_channel.hpp:41
Boost.Fiber	YES	NO	buffered_channel.hpp:536
Boost.Fiber	YES	NO	unbuffered_channel.hpp:585
Boost.Flyweight	YES	NO	archive_constructed.hpp:70
Boost.Flyweight	NO	NO	key_value.hpp:157-164
Boost.Foreach	NO	YES	foreach.hpp:611
Boost.Geometry	YES	NO	varray.hpp:165-168
Boost.Geometry	YES	NO	varray.hpp:1052
Boost.Geometry	YES	NO	varray.hpp:1101
Boost.Geometry	YES	NO	varray.hpp:1613-1616
Boost.Geometry	YES	NO	serialization.hpp:50
Boost.Hana	NO	YES	traits.hpp:176
Boost.Hana	NO	NO	traits.hpp:169
Boost.Hof	NO	YES	construct.hpp:106
Boost.Interprocess	NO	NO	offset_ptr.hpp:69-72
Boost.Iostreams	YES	NO	optional.hpp:108

Library	Is of the form aligned_storage<sizeof(T), alignof(T)>?	Depends on default alignment?	Source
Boost.Lockfree	NO	NO	spsc_queue.hpp:429-431
Boost.Log	YES	NO	thread_id.cpp:131
Boost.Log	YES	NO	threadsafe_queue.hpp:78
Boost.MultiIndex	YES	NO	archive_constructed.hpp:74
Boost.MultiIndex	YES	NO	seq_index_ops.hpp:134-137
Boost.MultiIndex	YES	NO	seq_index_ops.hpp:141-149
Boost.Optional	YES	NO	old_optional_implementation.hpp:87
Boost.Optional	YES	NO	optional.hpp:120
Boost.Parameter	NO	YES	maybe.hpp:37-39
Boost.PolyCollection	YES	NO	value_holder.hpp:61
Boost.Pool	YES	NO	singleton_pool.hpp:196
Boost.Python	NO	YES	referent_storage.hpp:59-61
Boost.Serialization	YES	NO	stack_constructor.hpp:38-41
Boost.Signals2	NO	NO	auto_buffer.hpp:1061-1063
Boost.Spirit	YES	NO	static.hpp:103-104
Boost.Unordered	YES	NO	implementation.hpp:751-752
Boost.Unordered	YES	NO	implementation.hpp:2753-2754
Boost.Utility	YES	NO	value_init.hpp:93
Boost.Variant	NO	NO	variant.hpp:366-369
Abseil	YES	NO	mutex_nonprod.inc:255
Abseil	YES	NO	fixed_array.h:399-400
Abseil	YES	NO	inlined_vector.h:1248-1249
Abseil	YES	NO	inlined_vector.h:1250-1251
Abseil	YES	NO	low_level_alloc.cc:222-223
Abseil	YES	NO	raw_hash_set.h:1156-1157
Abseil	YES	NO	raw_hash_set.h:1626-1627
Abseil	YES	NO	raw_hash_set.h:542-543
Abseil	NO	NO	symbolize_win32.inc:54-55

[4] `std::aligned_storage` on CppReference.com

[5] StackOverflow comment by user743382

[6] StackOverflow post by Andrew Tomazos

[7] StackOverflow post by nwp

[8] StackOverflow comment by Andrew

[9] "More C++ Idioms/Nifty Counter" on WikiBooks.org

[10] "Pimpl idiom without dynamic memory allocation" blog post

[11] "À propos de l'alignement en mémoire" blog post

[12] "Strange behavior of std::aligned_storage" blog post

[13] `WTF::NeverDestroyed` on opensource.apple.com

[14] "Multiple Producer Single Consumer Lockless Q" on haskell.org