Aligned memory: `aligned_allocator` and `aligned_span`

Anabel S. M. Ruggiero

April 12, 2022

1 Introduction

This paper is intended to propose additions to complete a minimum set of essentials to provide

- a safe interface to aligned `operator new` and `operator delete`,
- embed properties of the memory into types without affecting the types of the objects on the aligned memory,
- and leverage the optimizations compilers currently provide for aligned memory loads.

The additions proposed in this paper, `aligned_allocator` and `aligned_span`, along with`vector` and `assume_aligned` and are what the author understands to be the minimum essentials.

2 Motivation

Motivation: SIMD operations Different sets of instructions to handle loading based on alignment x86: SSE - align to 16 bytes or segfault trying AVX - movups No alignment requirement beyond that of `float`, movaps requires alignment to 32 bytes, but can yield better performance dependent on the target processor. Why this matters: load speeds are a typical limiting step in a computation: Put benchmarks here

2.1 What the Standard Already Provides

C++11 introduced overaligned types to the language. Using this feature, a struct wrapping a fixed sized array of 8 `float`s could be used to force alignments, but this requires taking a hit to ergonomics. Instead of accessing an array of floats, you have to work with array of this wrapping type. However, treating the alignment as a property of the memory and not a property of the objects that sit on it resolves the awkwardness of working with wrapper types.

In C++17, alignment aware overloads of `operator new` were added to the language. While C introduced `aligned_alloc` in C11, `aligned_alloc` was not added to the C++ standard library until C++17. While `new` expressions supported over aligned types since C++11, this was the first time a standard interface to the underlying aligned memory allocation was exposed. This also exposed access to form of placement new that would allocate memory, construct an object... and immediately render your program ill-formed. Since the already discouraged new expression does not work for aligning the underlying memory, this leaves a direct, raw call to `::operator new`... or to look to the idiomatic interface for handling memory and construction as disjoint operations: allocators. `std::pmr::polymorphic_allocator` provides an interface for aligned memory allocation.

However, avoiding the runtime overhead of an alignment check either requires part of the semantics of the program to remain implicit, or embedding that information in as part of a static type. One possibility is to have alignment be a property of a container, but embedding the information into an allocator provides this with a minimal addition to the standard.

2.2 The Bare Essentials

[Godbolt link] shows all that the components listed in section 1 together enables GCC and Clang to emit aligned loads for elements in the vectors. `vector<float, allocator<float>>` and `vector<float, aligned_allocator<float, 32>>` both result in only `vmovups` emitted in the compiled assembly. Only calling `EuclideanNorm` with `aligned_span<float, 32>` provide GCC and Clang with the information necessary to emit `vmovaps` instructions. This ensemble effect can still be disrupted by changing a single implementation detail: removing the call to `assume_aligned` in `aligned_span::data()`, which is used in `aligned_span::begin()`.

3 Impact On The Standard

This proposal only proposes additions to the standard library.

4 Design Decisions

As suggested above, element wrappers were explicitly a design that was avoided.

Container based designs were considered, but still requires the container's allocator to provide an interface to some form of aligned memory allocation through 'std::allocator_traits'.

An other possible space for `aligned_span` is a version of `span` that offers more general customization. For example, the currently proposed `mdspan` allows customization of behavior through the `layout` and `accessor` template parameters. The difference in behavior between `aligned_span` and `span` is conceptually similar to using an `aligned_right` or `aligned_left` layout policy in `mdspan`.

The designs of `aligned_allocator` and `aligned_span` closely mirror those of `std::allocator` and `std::span`, respectively.

Remaining design questions

- type of align template parameter, `size_t` is more ergonomic due to not being a scoped enum, but `align_val_t` matches the `operator new` overload.
- contiguous iterator overload of `assume_aligned`?
- expose type traits/concepts for detecting aligned contiguous ranges?
- implementation defined default alignment parameter?
- how should fixed-sized `aligned_span`s be exposed?

Questions regarding direct interaction with containers will be raised in array paper.

5 Technical Specification: `aligned_allocator`

5.1 Additions to Header <memory> synopsis [memory.syn]

- In 20.2.2 [memory.syn], add the following class template and function declarations after the declarations for `allocator`:

```
// [aligned.allocator], the aligned allocator
template<class T> class aligned_allocator;

template<class T, class U>
template<typename T, typename U, align_val_t Align>
  constexpr bool operator==(
    const aligned_allocator<T, Align>&,
    const aligned_allocator<U, Align>&) noexcept;

template<typename T, align_val_t TAlign,
         typename U, align_val_t UAlign>
  requires (TAlign != UAlign)
  constexpr bool operator==(
    const aligned_allocator<T, TAlign>&,
    const aligned_allocator<U, UAlign>&) noexcept;
```

5.2 New Subclauses

In the following subsections, *X*, will be used as a placeholder section number.

5.2.1 20.2.*X* The aligned allocator [aligned.allocator]

No text is proposed for this subclause.

5.2.2 20.2.*X*.1 General [aligned.allocator.general]

All specialization of the aligned allocator meet the allocator completeness requirements ([allocator.requirements.completeness]).

```
namespace std {
  template<class T, align_val_t Align> class aligned_allocator {
   public:
    using value_type                                 = T;
    using size_type                                  = size_t;
    using difference_type                            = ptrdiff_t;
    using propagate_on_container_move_assignment = true_type;

    template<class U> struct rebind{
      using other = aligned_allocator<U, Align>;
    };

    static constexpr align_val_t alignment       = Align

    constexpr aligned_allocator() noexcept;
    constexpr aligned_allocator(const aligned_allocator&) noexcept;
    template<class U> constexpr aligned_allocator(const aligned_allocator<U, Align>&) noex
```

```
    constexpr ~aligned_allocator();
    constexpr aligned_allocator& operator=(const aligned_allocator&) = default;

    [[nodiscard]] constexpr T* allocate(size_t n);
    [[nodiscard]] constexpr T* allocate(size_t n, align_val_t req_align);
    [[nodiscard]] constexpr allocation_result<T*> allocate_at_least(size_t n);
    [[nodiscard]] constexpr allocation_result<T*> allocate_at_least(size_t n, align_val_t
    constexpr void deallocate(T* p, size_t n);
    constexpr void deallocate(T* p, size_t n, align_val_t req_align);
  };
}
```

`allocator_traits<aligned_allocator<T, Align>>::is_always_equal::value` is `true` for any `T` and `Align`.

5.2.3 20.2.*X*.2 Members [aligned.allocator.members]

Except for the destructor, member functions of the aligned allocator shall not introduce data races ([intro.multithread]) as a result of concurrent calls to those member functions from different threads. Calls to these functions that allocate or deallocate a particular unit of storage shall occur in a single total order, and each such deallocation call shall happen before the next allocation (if any) in this order.

```
[[nodiscard]] constexpr T* allocate(size_t n);
```

Mandates: T is not an incomplete type ([basic.types.general]).
Returns: A pointer to the initial element of an array of `n T`. The pointer shall have an alignment of at least `Align` and behave as if the result of a call to `assume_aligned<Align, T>`.
Throws: `bad_array_new_length` if `numeric_limits<size_t>::max() / sizeof(T) < n`, or `bad_alloc` if the storage cannot be obtained.
Remarks: The storage for the array is obtained by calling the aligned overload of `::operator new` ([new.delete]), but it is unspecified when or how often this function is called. This function starts the lifetime of the array object, but not that of any of the array elements.

```
[[nodiscard]] constexpr T* allocate(size_t n, align_val_t req_align);
```

Mandates: T is not an incomplete type ([basic.types.general]).
Returns: A pointer to the initial element of an array of `n T`. The pointer shall have an alignment of at least `max(Align, req_align)` and behave as if the result of a call to `assume_aligned<Align, T>`.
Throws: `bad_array_new_length` if `numeric_limits<size_t>::max() / sizeof(T) < n`, or `bad_alloc` if the storage cannot be obtained.
Remarks: The storage for the array is obtained by calling the aligned overload of `::operator new` ([new.delete]), but it is unspecified when or how often this function is called. This function starts the lifetime of the array object, but not that of any of the array elements.

```
[[nodiscard]] constexpr allocation_result<T*> allocate_at_least(size_t n);
```

Mandates: T is not an incomplete type ([basic.types.general]).
Returns: `allocation_result<T*>ptr, count`, where `ptr` is a pointer to the initial element of an array of `count T` **and** `count >= n`. `ptr` shall have an alignment of at least `Align` and behave as if the result of a call to `assume_aligned<Align, T>`.

Throws: `bad_array_new_length` if `numeric_limits<size_t>::max() / sizeof(T) < n`, or `bad_alloc` if the storage cannot be obtained.
Remarks: The storage for the array is obtained by calling the aligned overload of `::operator new` ([new.delete]), but it is unspecified when or how often this function is called. This function starts the lifetime of the array object, but not that of any of the array elements.

```
[[nodiscard]] constexpr allocation_result<T*> allocate_at_least(size_t n, align_val_t req_
```

Mandates: T is not an incomplete type ([basic.types.general]).
Returns:`allocation_result<T*>ptr, count`, where `ptr` is a pointer to the initial element of an array of `count T` **and** `count >= n`. `ptr` shall have an alignment of at least `max(Align, req_align)` and behave as if the result of a call to `assume_aligned<Align, T>`.
Throws: `bad_array_new_length` if `numeric_limits<size_t>::max() / sizeof(T) < n`, or `bad_alloc` if the storage cannot be obtained.
Remarks: The storage for the array is obtained by calling the aligned overload of `::operator new` ([new.delete]), but it is unspecified when or how often this function is called. This function starts the lifetime of the array object, but not that of any of the array elements.

```
constexpr void deallocate(T* p, size_t n);
```

Preconditions: If `p` is memory that was obtained by a call to `allocate_at_least(size`$_t$`)`, *let*`ret`*be the value*

```
template<class T, class U>
template<typename T, typename U, align_val_t Align>
  constexpr bool operator==(
    const aligned_allocator<T, Align>&,
    const aligned_allocator<U, Align>&) noexcept;
```

Returns: `true`.

```
template<typename T, align_val_t TAlign,
         typename U, align_val_t UAlign>
  requires (TAlign != UAlign)
  constexpr bool operator==(
    const aligned_allocator<T, TAlign>&,
    const aligned_allocator<U, UAlign>&) noexcept;
```

Returns: `false`.

6 Technical Specification: `aligned_span`

6.1 Changes to subclause 24.7.1 [views.general]

- Change ''The header defines the view span.'' to ''The header defines the views span and aligned$_{span}$.''

6.2 Additions to Header synopsis [span.syn]

- In 24.7.2 [span.syn], add the following class template and function declarations after the declarations for span:

```
// [views.aligned.span], class template aligned_span
  template<class ElementType, size_t Extent = dynamic_extent>
    class span;
```

```
template<class ElementType, size_t Extent>
  inline constexpr bool ranges::enable_view<span<ElementType, Extent>> = true;
template<class ElementType, size_t Extent>
  inline constexpr bool ranges::enable_borrowed_range<span<ElementType, Extent>> = t

// [aligned.span.objectrep], views of object representation
template<class ElementType, size_t Extent>
  span<const byte, Extent == dynamic_extent ? dynamic_extent : sizeof(ElementType) *
    as_bytes(span<ElementType, Extent> s) noexcept;

template<class ElementType, size_t Extent>
  span<byte, Extent == dynamic_extent ? dynamic_extent : sizeof(ElementType) * Exten
    as_writable_bytes(span<ElementType, Extent> s) noexcept;
```