WHEN YOU HIT „ERROR: MEMORY IS NOT ALIGNED”

Mateusz Nowak @ code::dive 2018

Hello !

Mateusz Nowak

Software Engineer @ Intel

Work on: Audio FW dev, preparing and giving speeches and trainings
Interested in: Team work organization, management topics

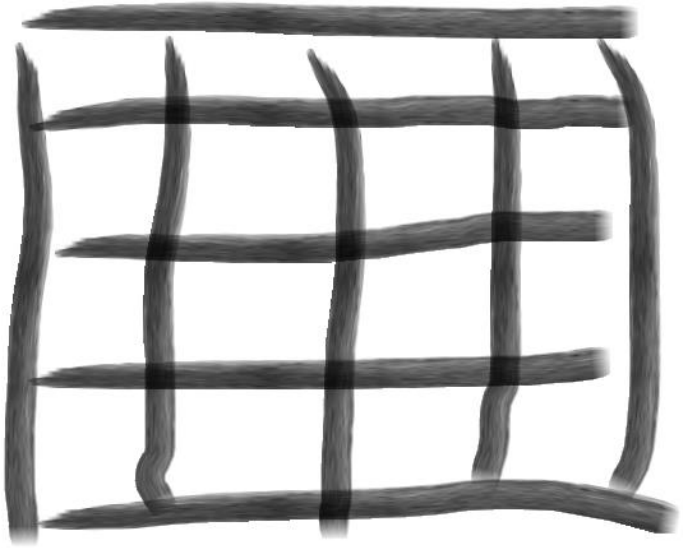
Agenda

1. What is alignment and why do we care
2. Enforcing alignment
   a. Compiler specific attributes and functions
   b. `alignas`, `alignof`, `std::align`, `std::aligned_storage`, and `aligned_alloc`
   c. Custom aligned allocator
   d. `std::align_val_t` and aligned new
   e. Polymorphic resources

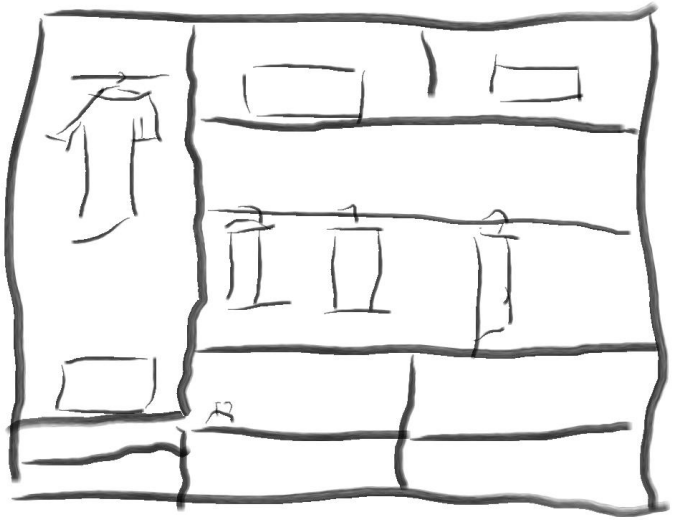
WHAT IS ALIGNMENT

And why do we care

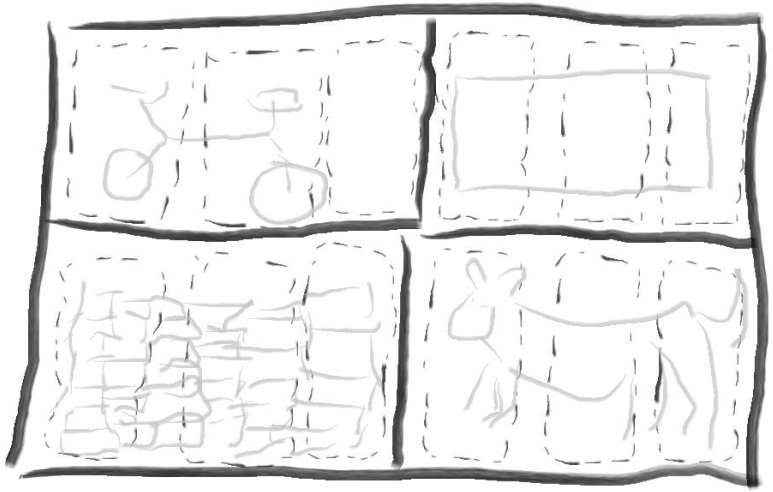
Memory



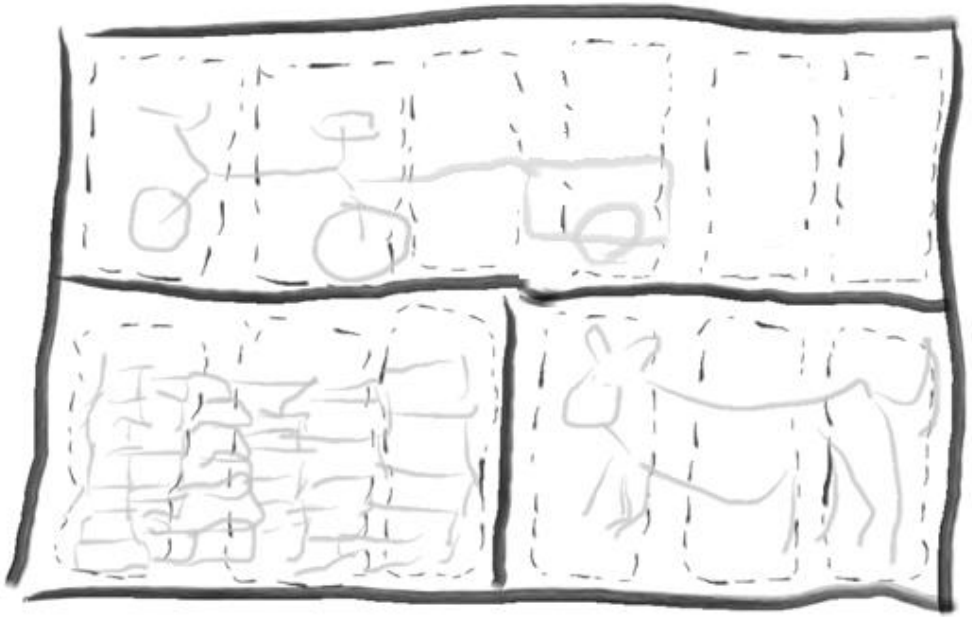
Usually we don't care about alignment



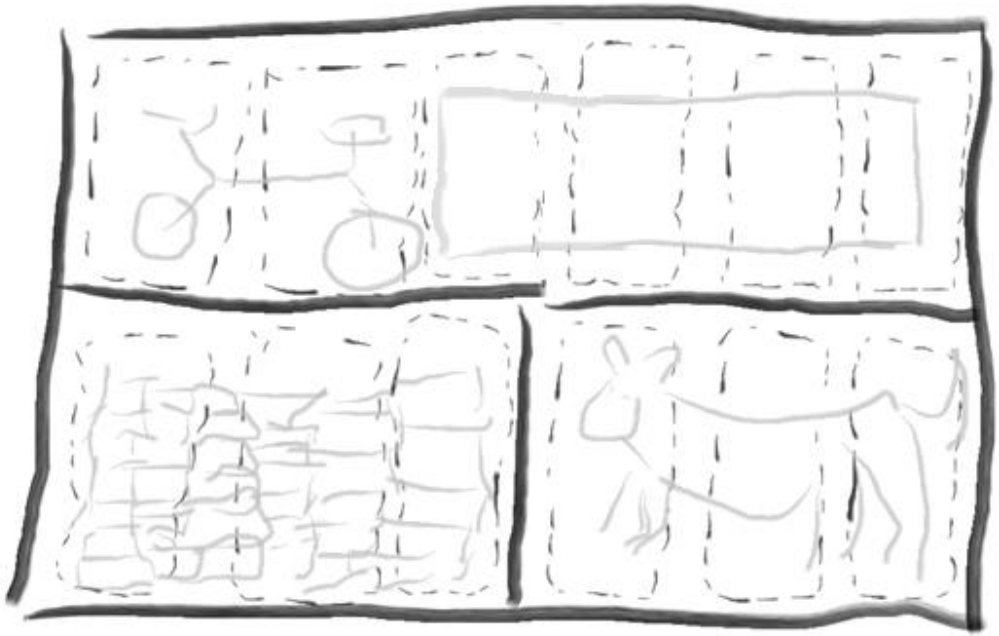
When shelves need to have specific size



When shelves need to have specific size



When it needs to be packed



Default rules for alignment

- 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)`. 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. [C++17 N4700 6.11:2]
- Storage is always apropriately aligned to fundamental alignment.
- **Padding is added** in structures to make sure all fields are properly aligned (unless packed).
- Data is usually read and write in chunks with architecture specific alignment (word size).

It may matter for:

Specific CPU instructions (like SSE)

Sharing data between CPUs

Explicit cache handling

HW registers mapping

Performance

ENFORCING ALIGNMENT

Different methods with different standards and tools

Compiler specific attributes

```
__attribute__((aligned(X)))
```
(gcc, clang)

```
__declspec(align(X))
```
(msvc)

```
__attribute__((packed))
```
(gcc, clang)

```
#pragma pack([push], [pop], [n])
```
(msvc)

Compiler specific functions

```
int posix_memalign(void** ptr, size_t alignment, size_t size)
```
(POSIX, gcc, clang)

```
void* _mm_malloc(int size, int alignment)
```
and
```
_mm_free(void* ptr)
```
(Intel specific)

```
void* _aligned_malloc(size_t size, size_t alignment)
```
and
```
_aligned_free(void* ptr)
```
(MSVC)

alignas and alignof

alignas defines alignment for type or object

```
struct alignas(16) S { int a; };
alignas(128) float buffer[20];
```

alignof and std::alignment_of allow to query alignment for the type

```
alignof(char); std::alignment_of<char>::value;
alignof(S); std::alignment_of<S>();
std::alignment_of_v<S>; // since C++17
```

Bonus: GNU extension alignof expression

```
alignof(buffer);
alignof(S::a);
```

Beware of alignas

```
alignas(64) std::vector<std::string> aligned_vector_of_unaligned_strings;

alignas(64) int * aligned_pointer_to_unaligned_int;

#ifndef __cplusplus
#define alignas(size) _Alignas(size) // Or #include <stdalign.h>
#endif

struct alignas(16) {
 uint8_t a;
 uint64_t b;
} some_struct;
if (((uint64_t)&some_struct.b) % 16 != 0) std::cout << "Members do not align.\n";
```

std::align

Obtain aligned pointer from given buffer.

```
size_t size = 160;
void * ptr_32_al;
posix_memalign(&ptr_32_al, 32, size);
void * ptr_64_al = std::align(64, 128, ptr_32_al, size);
```

std::aligned_storage


```
constexpr size_t data_size = 2;
constexpr size_t data_size_bytes = data_size * sizeof(float);
constexpr size_t alignment = 16;
std::aligned_storage<data_size_bytes, alignment>::type
// with C++14: std::aligned_storage_t<data_size_bytes, alignment>
    data_storage;
(...)
auto * sse_data = reinterpret_cast<__m128*>(&data_storage);
size_t sse_size = sizeof(data_storage) / sizeof(float) / 4;
for (size_t i{ 0 }; i < sse_size; ++i) {
  sse_data[i] = _mm_sqrt_ps(sse_data[i]);
}
```

```
// In MSVC remember to set /D "_ENABLE_EXTENDED_ALIGNED_STORAGE
```

aligned_alloc

Available in C11 and C++17.

If not strict, some compilers allow to use for older standards.

Need to be aware of size requirement.

Free with `free(void* ptr)`

```
size_t aligned_size(size_t size, size_t align)
{
 assert(align > 0);
 if (size % align == 0) return size;
 return (size + align - 1) / align * align;
}

float * array = aligned_alloc(8,
 aligned_size(20 * sizeof(float), 8));
```

Not available in msvc: https://blogs.msdn.microsoft.com/vcblog/2017/12/19/c17-progress-in-vs-2017-15-5-and-15-6/

Custom allocators

```
template <typename T>
struct custom_allocator {
 template <typename> friend struct custom_allocator;
 typedef T value_type;
 custom_allocator() noexcept {}
 template <typename U>
 constexpr custom_allocator(const custom_allocator<U>& other) noexcept :
   custom_allocator() {}
 T* allocate(std::size_t n);
 void deallocate(T * p, std::size_t n) noexcept;
};
template <class T, class U>
bool operator==(const custom_allocator<T>&, const custom_allocator<U>&) { return true; }
template <class T, class U>
bool operator!=(const custom_allocator<T>&, const custom_allocator<U>&) { return false; }
```

Custom aligned allocator

```
template <typename T> struct custom_allocator {
  ...
  custom_allocator(std::size_t alignment = 8) noexcept : alignment_(alignment) {}
  template <typename U> constexpr custom_allocator(const custom_allocator<U>& other) noexcept :
    custom_allocator(other.alignment_) {}
  T* allocate(std::size_t n) {
    if (n > (std::size_t(-1) / sizeof(T))) return nullptr; // Or throw std::bad_alloc
    return static_cast<T*>(aligned_alloc(alignment_, aligned_size(n * sizeof(T))));
  }
  void deallocate(T * p, std::size_t) noexcept { free(p); }
  private:
  std::size_t alignment_;
  std::size_t aligned_size(std::size_t size)
  { return size % alignment_ == 0 ? size : (size + alignment_ - (size % alignment_)); }
};

template <typename T> using aligned_allocator = custom_allocator<T>;
```

Aligned allocator – usage examples

```
std::vector<int, aligned_allocator<int>> vec(5, aligned_allocator<int>(64));
// Be aware this align .data(), but not every vector element
```

```
using aligned_string = std::basic_string<char, std::char_traits<char>, aligned_allocator<char>>;
aligned_allocator<char> allocator(16);
aligned_string s("some text", allocator);
```

```
struct C {...};
aligned_allocator<C>(64) a;
auto * v = new (a.allocate(1)) C;
v->~C();
a.deallocate(v);
```

Aligned allocator – custom deleter

```
template<typename T, typename A>
struct deleter {
 deleter(A & allocator) : allocator_(allocator) {}
 void operator()(T * ptr) {
 if (ptr == nullptr) return;
 ptr->~T();
 allocator_.deallocate(ptr);
 }
private:
A & allocator_;
};

aligned_allocator<C>(64) a;
auto * v = new (a.allocate(1)) C;
deleter<C, aligned_allocator<C>>(a)(v);
```

Aligned allocator – unique_ptr example

```
aligned_allocator<int> allocator(32); // 32 bytes alignment
std::unique_ptr<int, deleter<int, aligned_allocator<int>>> ptr(new (allocator.allocate(1)) int,
  deleter<int, aligned_allocator<int>>(allocator));

template<typename T, typename A>
using aligned_unique_ptr = std::unique_ptr<T, deleter<T, A>>;
template<typename T, typename A>
aligned_unique_ptr<T, A> make_aligned_unique(A & allocator) {
  return aligned_unique_ptr<T, A>(new (allocator.allocate(1)) T, deleter<T, A>(allocator));
}

template<typename T, typename A, typename D>
unique_ptr<T, D> make_aligned_unique2(A & allocator, D & deleter) {
  return unique_ptr<T, D>(new (allocator.allocate(1)) T, deleter);
}
```

unique_ptr<T[]> trap

```
struct Data { ... };

aligned_allocator<Data> a(128);
using del = deleter<Data, aligned_allocator<Data>>;
del d(a);

std::unique_ptr<Data[], del> array(new (a.allocate(20)) Data[20], d);
```

unique_ptr<T[]> - Bind size to deleter function

```
auto deleter = [](Data * ptr, size_t size) {
 if (!ptr) return;
 for (size_t i = 0; i < size; ++i) ptr[i].~Data();
 free(ptr);
};
std::unique_ptr<Data[], std::function<void(Data*)>> array(
 new (aligned_alloc(20 * sizeof(Data)) Data[20],
 std::bind(deleter, std::placeholders::_1, 20));
```

unique_ptr<T[]> - Deleter instance

```
struct Deleter {
 Deleter(size_t size) : size_(size) {}
 template<typename T>
 void operator()(T * ptr) {
  if (!ptr) return;
  for (size_t i = 0; i < size_; ++i) ptr[i].~T();
  free(ptr);
 }
private:
size_t size_;
};

std::unique_ptr<int[], Deleter> p(new (aligned_alloc(20 * sizeof(int)) int[20], Deleter(20));
```

Custom array_ptr<T, D>

```
template<typename T, typename D>
struct array_ptr {
 array_ptr(T * ptr, size_t size, D && deleter) : ptr_(ptr), size_(size), deleter_(deleter) {}
 ~array_ptr() { deleter_(ptr_, size_); }
 array_ptr(array_ptr &) = delete;
 array_ptr& operator=(array_ptr &) = delete;
 array_ptr(array_ptr && other);
 array_ptr& operator=(array_ptr && other);
 void reset(T * ptr, size_t size);
 T * get() { return ptr_; }
 T & operator[](size_t index);
private:
 D & deleter_;
 T * ptr_;
 size_t size_;
};
```

Custom array_ptr<T, D> and array deleter

```
struct Deleter {
  template<typename T>
  void operator()(T * ptr, size_t size = 1) {
    if (!ptr) return;
    for (size_t i = 0; i < size; ++i) ptr[i].~T();
    free(ptr);
  }
};

array_ptr<int, Deleter> a(new int[20], 20, Deleter());
```

Why array_ptr<T, D> instead of std::vector<T, D>

- Explicit control over memory block size
- Clear ownership
- ???
- Same as `std::unique_ptr<T[]>` vs `std::vector<T>` ?

std::align_val_t and aligned new


```
__STDCPP_DEFAULT_NEW_ALIGNMENT__
auto * const ptr = new (std::align_val_t(128)) int;
/Zc:alignedNew (msvc)
-faligned-new=N (gcc)
```

delete goes crazy

```
#include <cstdlib>
#include <new>

struct a {
 void * operator new(std::size_t count) {
 return ::new a;
 }
 void * operator new(std::size_t count, std::align_val_t alignment) {
 return ::new (alignment) a;
 }
 void operator delete(void * pointer) {
 ::operator delete(pointer);
 }
};

auto * aa = new (std::align_val_t{64}) a;
::operator delete(aa, std::align_val_t{64});
// or (if there was a::operator delete(void*, std::align_val_t) defined):
// aa->operator delete(aa, std::align_val_t{64});
```

pmr::


```
std::array<std::byte, 1024*10> buffer;
std::pmr::monotonic_buffer_resource underlying
    { buffer.data(), buffer.size(), std::pmr::null_memory_resource() };
std::pmr::synchronized_pool_resource pool { &underlying };
std::pmr::vector<std::pmr::string> vector_of_strings{ &pool };
```

pmr:: Custom aligned memory resource


```
template <size_t Alignment>
struct aligned_resource : public std::pmr::memory_resource {
    explicit aligned_resource(
        std::pmr::memory_resource * const us = std::pmr::get_default_resource()) :
        upstream_ { us } {}
private:
    std::pmr::memory_resource * const upstream_;
    void * do_allocate(size_t bytes, size_t alignment) override {
        size_t final_alignment = std::max(Alignment, alignment);
        return upstream_->allocate(bytes, final_alignment);
    }
    void do_deallocate(void * ptr, size_t bytes, size_t alignment) override {
        size_t final_alignment = std::max(Alignment, alignment);
        upstream_->deallocate(ptr, bytes, final_alignment);
    }
    bool do_is_equal(const std::pmr::memory_resource& other) const noexcept override {
        return this == &other || dynamic_cast<const aligned_resource*>(&other) != nullptr;
    }
};
```

pmr:: aligned


```
std::array<std::byte, 1024*10> buffer;
std::pmr::monotonic_buffer_resource underlying
    { buffer.data(), buffer.size(), std::pmr::null_memory_resource() };
aligned_resource<64> underlying2{ &underlying };
std::pmr::synchronized_pool_resource pool { &underlying2 };
std::pmr::vector<std::pmr::string> vector_of_strings{ &pool };
```

Q & A

If you have more question please catch me on LinkedIn:

Mateusz Nowak https://www.linkedin.com/in/mateusz-nowak-b2006083/

Or via mail: mateusz.nowak@intel.com

Feedback is appreciated ☺