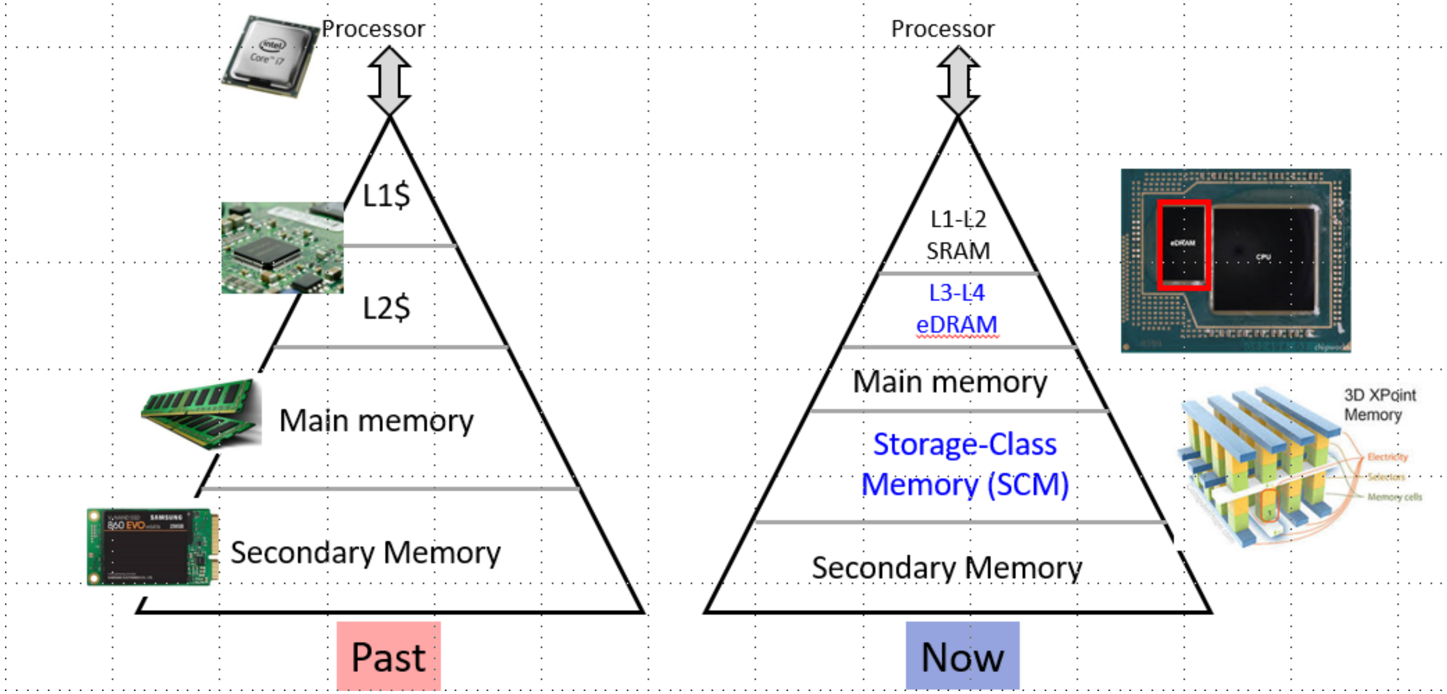
内存管理
Memory Management

现代C++基础
Modern C++ Basics

Jiaming Liang, undergraduate from Peking University

- Low-level Memory Management
- Smart Pointers
- Allocators
  - PMR

- The real structure of memory is quite complex…



Credit: Jie Zhang @ PKU Arch.

- However, OS has abstracted them as *virtual memory* by page table, so in most cases users can view memory as a large contiguous array.
  - When such abstraction causes performance bottleneck, programmers need to dig into that further;
    - C++ also has some utilities to solve some common problems.

Memory Management

Low-level Memory Management

Memory Management

- Low-level Memory Management
  - Object layout
  - `operator new/delete` in detail

Object layout

- Object will occupy a contiguous segment of memory that:
  - Starts at some address that matches some **alignment**;
  - And ends at some address that matches some **size**.
- A *complete object* may have many *subobjects* as members or elements (e.g. array or `class`).
  - `sizeof` reflects size of the type when it forms a complete object, which is always `>0`.
  - For example:

```
struct Empty {};
static_assert(sizeof(Empty) > 0);
```

- In most cases, subobjects just occupy memory in the same way:

```
struct Empty {};
struct NonEmpty
{
    int a;
    Empty e;
};
// Padding may exist so we use '>='
static_assert(sizeof(NonEmpty) >= sizeof(int) + sizeof(Empty));
```

Object layout

- However, some subobjects as class member can have 0 size…
  - Formally called "potentially-overlapping objects".

1. For a class, if it fulfills:
   - No non-static data members;
   - No virtual methods or virtual base class;
   - It's a base class.

   Then it's **allowed** to have 0 size.

   Moreover, it's **forced** to have 0 size if
   - The derived class is a standard-layout class.

```cpp
struct Empty {};
struct NonEmpty : Empty
{ // Standard-layout
    int a;
};
static_assert(sizeof(NonEmpty) == sizeof(int));
```

- Also called "*Empty Base (Class) Optimization*" (EBO/EBCO).

Object layout

- So now we can understand `static_cast` / `reinterpret_cast`…

> - For `static_cast`, besides inheritance-related pointer conversion, it also processes `void*`.
>   - You can convert any **object** pointer to `void*` (this is also implicit conversion).
>   - You can also convert explicitly `void*` to any object pointer.
>   - **BUT**, this requires the underlying object of type `U` and the converted pointer `T*` (ignoring cv) to have the relationship (called *pointer-interconvertible*) as:
>   - `T == U`.
>   - `U` is a union type, while `T` is type of its member (though using it still needs this member to be in its lifetime).
>   - `U` is standard-layout, while `T` is type of its **first** member or its base class.
>   - Or all vice versa/transitivity (i.e. you can swap `T` and `U` above; after all, "inter").

In lecture 5, *Lifetime & Type Safety*.

FYI, this can be checked by [`std::is_pointer_interconvertible_with_class`]() and [`std::is_pointer_interconvertible_base_of`]() since C++20.

Object layout

- Empty base will be collapsed so conversion is safe.

```cpp
struct Empty {};
struct NonEmpty : Empty
{ // standard-layout
    int a;
};

NonEmpty obj;
// ptr points to the base class of obj.
Empty* ptr = reinterpret_cast<Empty*>(&obj);
// ptr2 points to obj.a.
int* ptr2 = reinterpret_cast<int*>(&obj);

static_assert(std::is_pointer_interconvertible_with_class(&NonEmpty::a));
static_assert(std::is_pointer_interconvertible_base_of_v<Empty, NonEmpty>);
```

- A class is said to be **standard-layout**, if:
  - All non-static data members have the same accessibility and are also standard-layout.
    - This is because the layout of members that have different accessibility are unspecified (before C++23); e.g. as the sequence of declaration or first all public members and second all private members.
  - No virtual functions or non-standard-layout base classes.
  - The base class is not the type of the first member data.
  - There is at most one class in the inheritance hierarchy that has non-static member variable.
    - That's because layout of inheritance is not regulated.

Object layout

[1]: Strictly speaking, it should be "similar types", e.g. adding cv-qualifiers is allowed. See [conv.qual] for details.
[2]: Except for potentially non-unique object like string literals.

2. Since C++20, for a member subobject that is marked with attribute `[[no_unique_address]]`, it's **allowed** to have 0 size.
   - Particularly, msvc will ignore this attribute for backward compatibility; instead, it respects `[[msvc::no_unique_address]]`.
   - For example:

```
struct Y
{
    int i;
    [[no_unique_address]] Empty e;
};
```

In gcc/msvc/clang, `sizeof(Y) == 4`.

   - Note: C++ regulates two objects of the same type[1] must have **distinct addresses**[2].
     - For example:

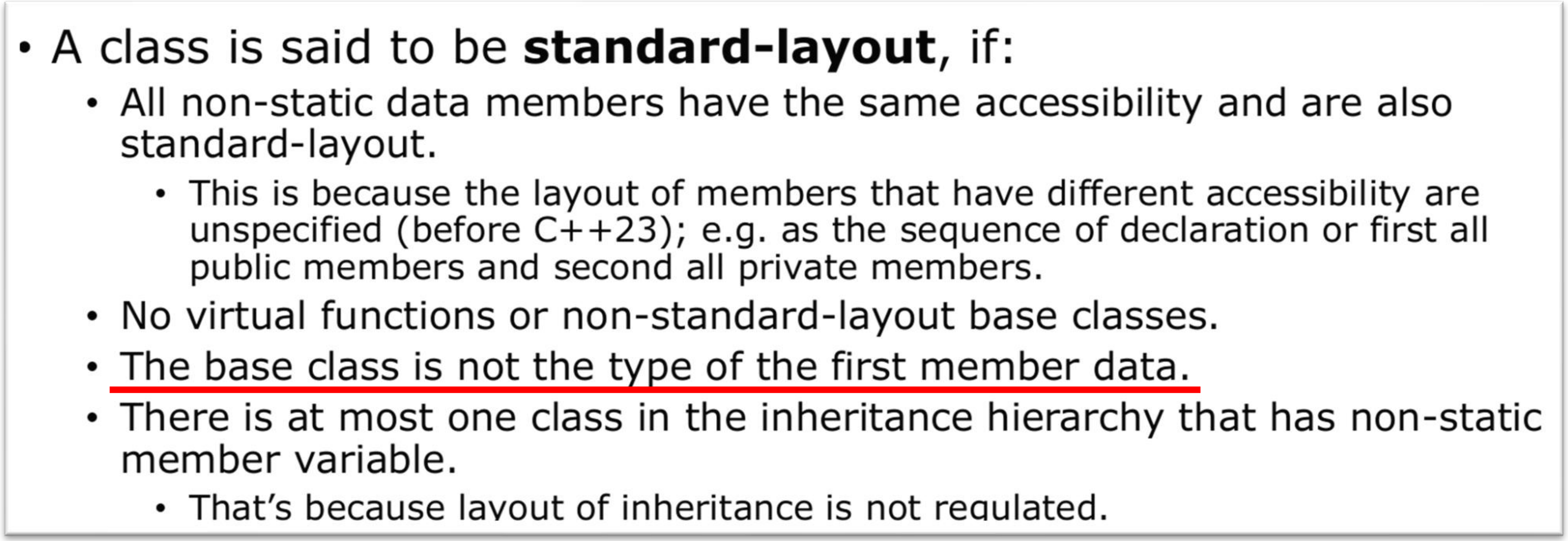
```
struct Z
{
    char c;
    // e1 and e2 cannot share the same address because they have the
    // same type, even though they are marked with [[no_unique_address]].
    // However, either may share address with 'c'.
    [[no_unique_address]] Empty e1, e2;
};
```

All three compilers make `sizeof(Y) == 2`.

Object layout

```
struct W
{
    char c[2];
    // e1 and e2 cannot have the same address, but one of
    // them can share with c[0] and the other with c[1]:
    [[no_unique_address]] Empty e1, e2;
};
```

- Theoretically, this can be optimized as `sizeof(W) == 2`; however, all three compilers make `sizeof(W) == 3`.
- And again, we can understand in standard layout…



Object layout

- Now EBCO doesn't guarantee to happen:

```cpp
struct Empty {};
struct NonEmpty : Empty
{ // Not standard-layout
    Empty e;
    int a;
};

NonEmpty obj;
// ptr doesn't necessarily point to e.
Empty* ptr = reinterpret_cast<Empty*>(&obj);
```

  - In ABI, base class may be put first;
  - As subject of base class must be distinguished from the first member, then base class may be not really "empty".
  - And a non-empty base leads to non-standard-layout*.

*: there may be some defects in current definitions. See [SO question](#).

Layout Compatible*

- **This part is optional**.
- Finally we fix our claim before:

> - Similarly, for union type, it's **illegal** to access an object that's not in its lifetime (it's only allowed in C)!
>   - Here `u.a` is in its lifetime, while `u.b` is not.
>   - You should use `std::memcpy` or `std::bit_cast` since C++20 to make them bitwise equivalent.
>
> ```
> union U { int a; float b; };
>
> int main()
> {
>     U u; u.a = 1; std::cout << u.b;
> ```

- Rigorously, when types have *common initial sequence*, it's legal to access out of lifetime:

```
struct T1 { int a, b; };
struct T2 { int c; double d; };
union U { T1 t1; T2 t2; };
int f() {
  U u = { { 1, 2 } };   // active member is t1
  return u.t2.c;        // OK, as if u.t1.a were nominated
}
```

Layout Compatible*

- Formally, we say two types are layout compatible if:
  - Naïve cases:
    - They are of the same type, ignoring cv qualifier; or,
    - They are enumerations with the same underlying integer type.
  - Otherwise,
    1. They are both standard-layout; and,
    2. Their common initial sequence covers all members.
- where common initial sequence means the longest sequence of non-static data members and bit-fields in declaration order that:
  1. corresponding entities are layout-compatible; and,
  2. corresponding entities have the same alignment requirements; and,
  3. either both entities are bit-fields with the same width or neither is a bit-field.

Layout Compatible*

- For example:

```cpp
struct A { int a; char b; };
struct B { const int b1; volatile char b2; };
// A and B's common initial sequence is A.a, A.b and B.b1, B.b2

struct C { int c; unsigned : 0; char b; };
// A and C's common initial sequence is A.a and C.c

struct D { int d; char b : 4; };
// A and D's common initial sequence is A.a and D.d

struct E { unsigned int e; char b; };
// A and E's common initial sequence is empty
```

A and B are layout-compatible.

- Since C++20, you can use `std::is_layout_compatible`* and `std::is_corresponding_member` to check it.

```cpp
struct T1 { int a, b; };
struct T2 { int c; double d; };
struct T3 { int a, b; };

static_assert(std::is_corresponding_member(&T1::a, &T2::c));
static_assert(!std::is_corresponding_member(&T1::b, &T2::d));
static_assert(std::is_layout_compatible_v<T1, T3>);
```

*: Strictly speaking, `std::is_layout_compatible` will tolerate non-struct-type, while the standard only regulates struct-type.

Alignment

- To maximize efficiency, data should be aligned properly.
  - For example, on some platform:

```
// 00 // long long, char, int can live here
// 01 // char
// 02 // char
// 03 // char
// 04 // char, int can live here
// 05 // char
// 06 // char
// 07 // char
// 08 // long long, char, int can live here
```

- In C++, it can be checked by `alignof(T)`;
  - Platform-dependent, return `std::size_t`, quite like `sizeof`.

```
std::println("{} {} {}", alignof(char), alignof(int), alignof(long long));
```

```
Program returned: 0
  1 4 8
```

*Or using type traits `std::alignment_of`.

Alignment

- When wrapping data in class, every object will be aligned to its own alignment, leading to padding.
  - For example:

```
struct S {  // begins at:
    char a; // 0
    // 3 padding bytes to match alignof(i)
    int i;  // 4
    char b; // 8
    // 3 padding bytes to match alignof(j)
    int j;  // 12
    char c; // 16
    // 7 padding bytes to match alignof(l)
    long long l; // 24
    // Possible padding bytes to match alignof(S)
}; // sizeof(S): at least 32.
```

Each element in C array should be suitably aligned, thus `sizeof(X)` must be multiple of `alignof(X)`.

Alignment

- Naturally, all scalar types will have alignment not greater than `alignof(std::max_align_t)` (in `<cstddef>`).
  - And allocation will align to this alignment by default.
- However, sometimes you may want *over-aligned* data.
  - Then you can use `alignas(N)` to make alignment `N`.
    - Ignored when `N == 0`, compile error if `N` is not power of 2.
  - For example, to match OpenGL uniform layout:

These three members are all aligned to 16.

```
struct BasicParams
{
    alignas(16) glm::vec3 cameraPos;
    int randOffset;

    alignas(16) glm::vec3 cameraForward, cameraRight, cameraUp;
    float g;
```

Alignment

- Note 1: you can also use `alignas(T)` to have alignment same as `T`.
- Note 2: when using multiple `alignas`, the largest one will be selected.
  - So our previous code segment can be rewritten:

```
alignas(std::max(alignof(float), alignof(int))) std::byte arr[20];
float* ptr = reinterpret_cast<float*>(arr);
*ptr = 1.0f;
int* ptr2 = reinterpret_cast<int*>(arr);
// std::cout << *ptr2; // -> illegal
```

```
alignas(float) alignas(int) std::byte arr[20];
```

- Note 3: you can do pack expansion in `alignas`, which is same as `alignas(arg1) alignas(arg2) … alignas(argN)`.
  - i.e. select the largest alignment among N arguments.

Alignment

```
alignas(1) int a = 2; ❌
```

- Note 4: over-align only: if `alignas` is weaker than its natural alignment (i.e. alignment without `alignas`), compile error.
  - Some compilers will ignore or only warn.
- Note 5: alignment is NOT part of the type, so you cannot alias it in `using` or `typedef`.

```
struct C {
    long long x;
    int y;
};

using T = alignas(16) C;
```

❌

Attributes are added after `struct`.

```
struct alignas(16) C {
    long long x;
    int y;
};
```

✅

```
struct C {
    long long x;
    int y;
};

struct D
{
    alignas(16) C c;
};
```

✅

- Note 6: function parameter and exception parameter are not allowed to use `alignas`.

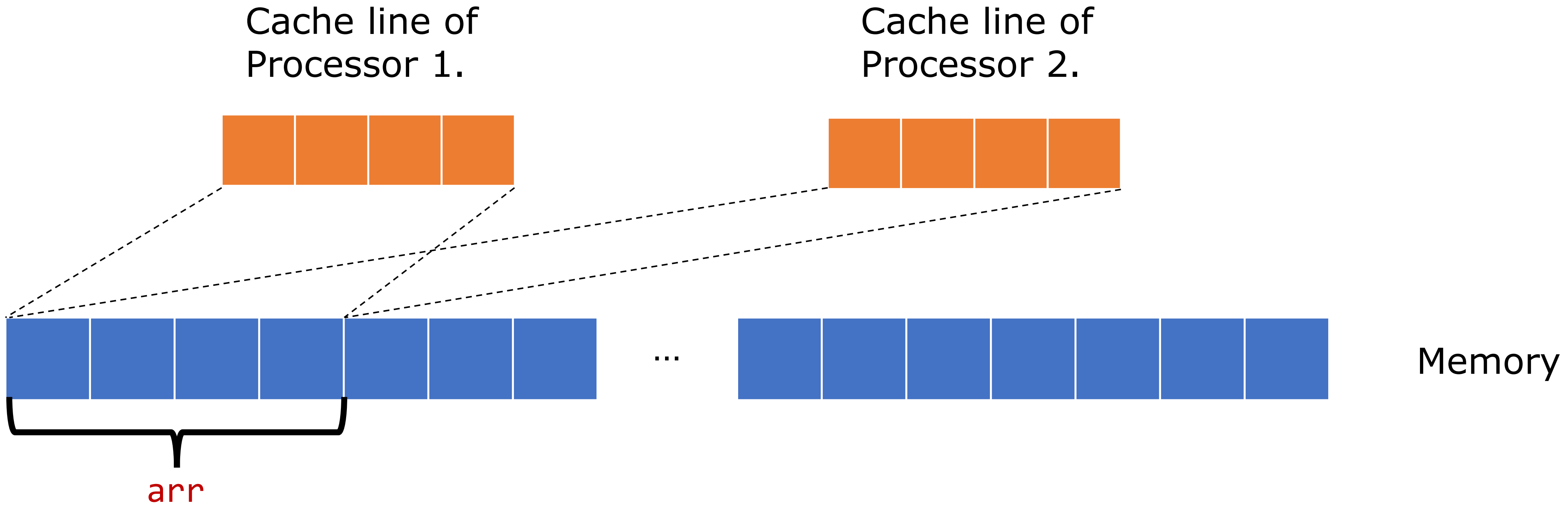
False Sharing

- Practical example: false sharing
  - From abstraction, when different threads operate on different data, parallelism will be maximized since no lock is needed.

```
// Here to prevent compiler optimization to collapse
// We use atomic<int> instead of int.
std::atomic<int> arr[4];
void Work(int idx) { arr[idx]++; }
```

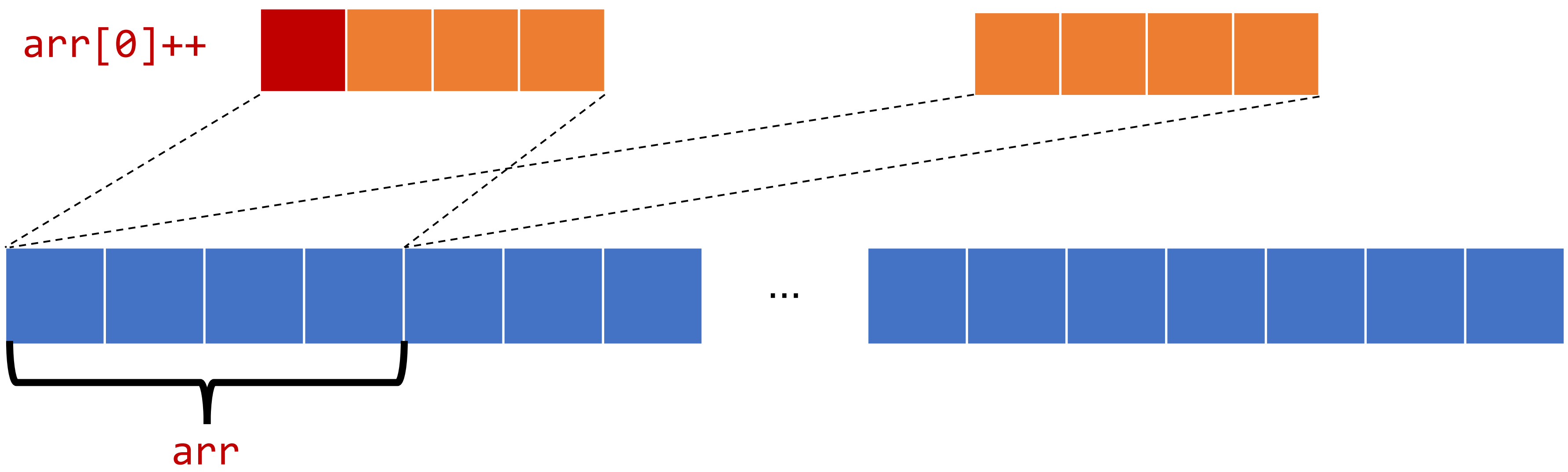
- However, due to limitation of computer architecture, such abstraction is wrong…
  - Cache on different processors has to obey coherence protocol like MESI.
  - To put it simply, when write happens on a cache line, it'll inform other processors whose cache also own this line to make it invalid.
  - And invalid line needs to be reloaded, leading to inefficiency.

False Sharing

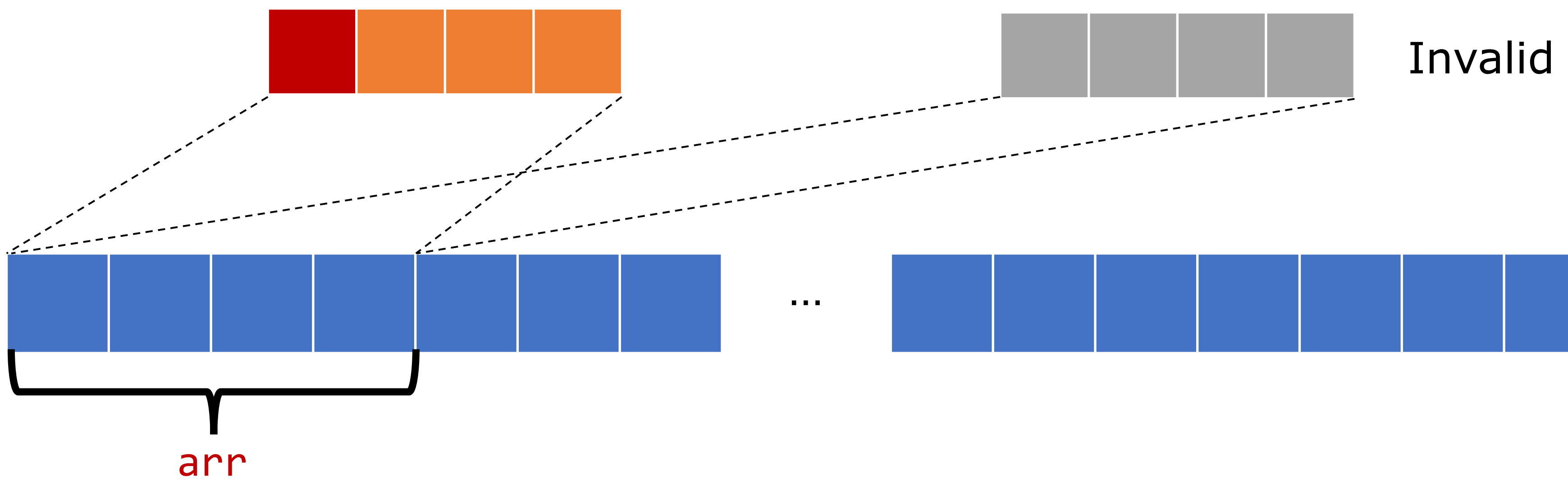


Illustrative animation for false sharing.
(Details may vary for different architectures)

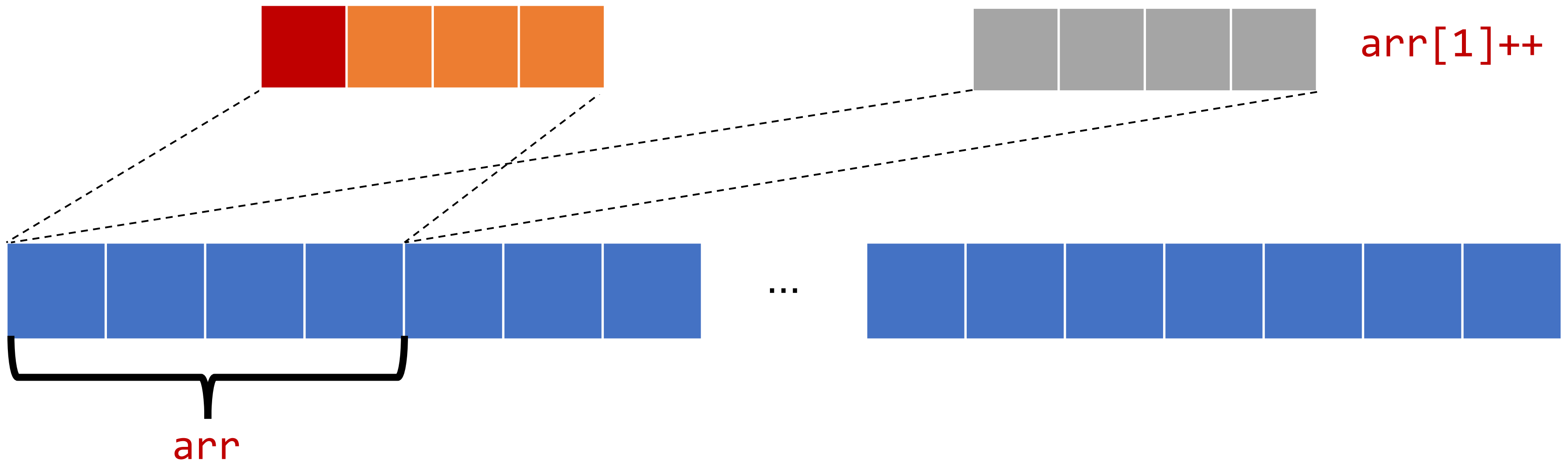
False Sharing



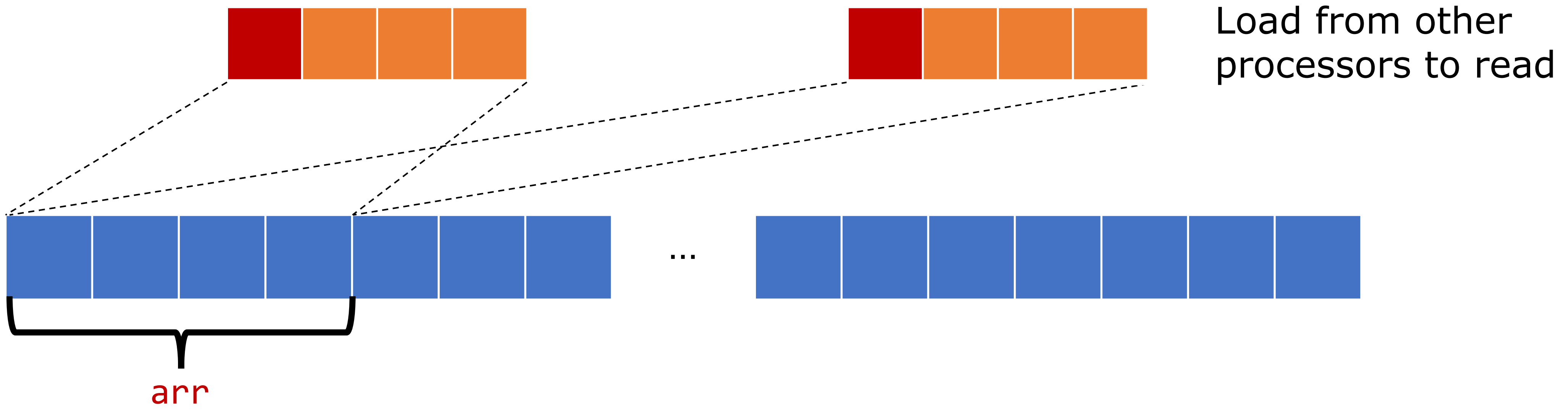
False Sharing



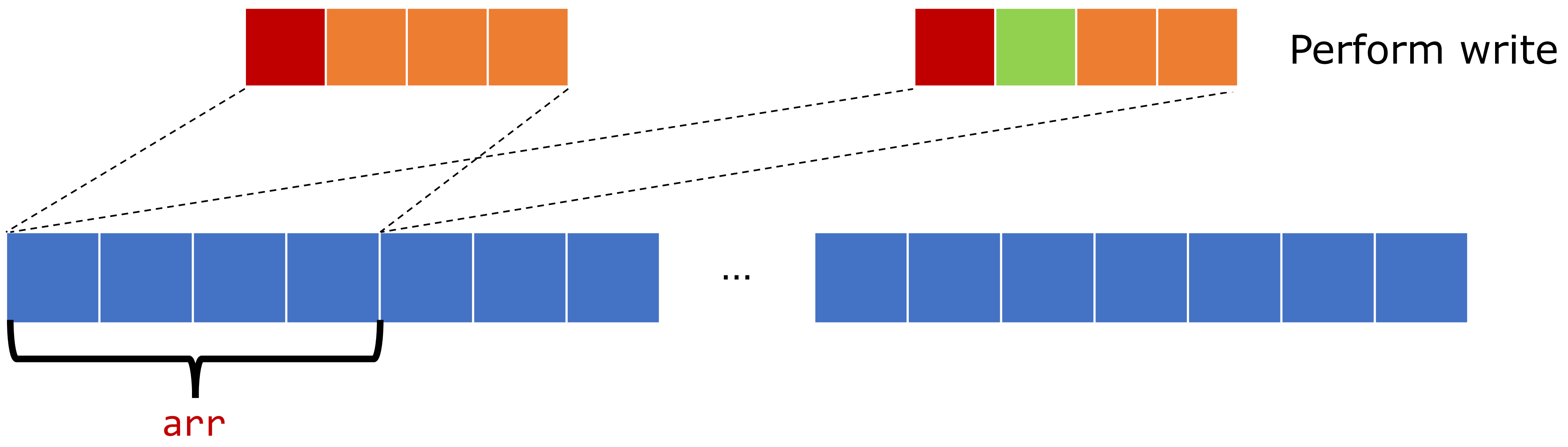
False Sharing



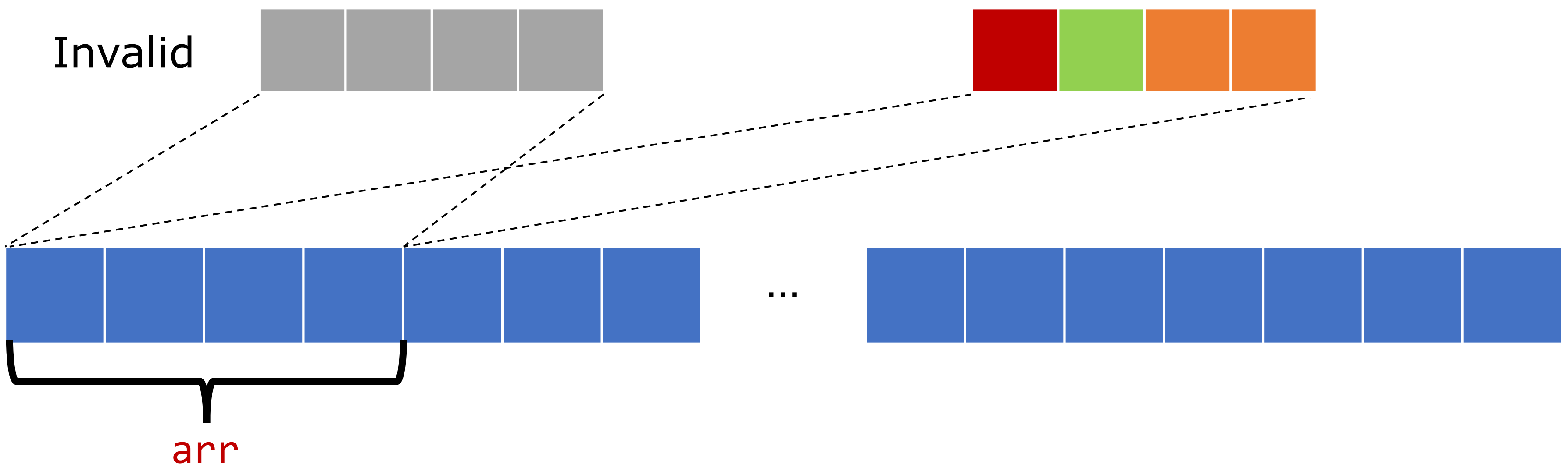
False Sharing



False Sharing



False Sharing



False Sharing

- So when writes in different threads are on the same cache line, every write will happen exclusively as if having a lock.
  - This leads to false parallelism, degrading the performance.
- Solution: make threads access data on different cache lines!
  - C++17 provides constant `std::hardware_destructive_interference_size` in `<new>`.
  - For example:

```
struct OveralignedInt
{
    alignas(std::hardware_destructive_interference_size) std::atomic<int> elem;
};
// alignas(N) T arr[4] won't align every element, but just arr[0].
// To align every element, we need to wrap inside a struct.
OveralignedInt arr[4];
void Work(int idx) { arr[idx].elem++; }
```

False Sharing

- On the other hand, for a single thread, we hope accessed data to lie on the same cache line to minimize pollution.
  - For example:

❌ Improperly aligned,
use two cache lines.

✅ Properly aligned,
use single cache line.

  - To force data to lie on the same cache line, we can align the head as cache line head.
  - C++17 thus introduces `std::hardware_constructive_interference_size` for that.

False Sharing

- For example:

```
struct alignas(hardware_constructive_interference_size)
OneCacheLiner // occupies one cache line
{
    std::atomic_uint64_t x{};
    std::atomic_uint64_t y{};
}
oneCacheLiner;

struct TwoCacheLiner // occupies two cache lines
{
    alignas(hardware_destructive_interference_size) std::atomic_uint64_t x{};
    alignas(hardware_destructive_interference_size) std::atomic_uint64_t y{};
}
twoCacheLiner;
```

- Question: aren't `std::hardware_destructive_interference_size` and `std::hardware_constructive_interference_size` just same as cache line size?
  - Why do we need two constants to represent them?

False Sharing

- Reason: in some architecture, destructive interference will be larger than a cache line…
  - For example, on Intel Sandy Bridge processor, it will do adjacent-line prefetching.
  - So when loading a cache line, the next cache line may or may not be substituted, leading `hardware_destructive_interference_size == 128` while `hardware_constructive_interference_size == 64`.

Supplementary

- Note 1: there exist several utilities for alignment in `<memory>`.

  1. `std::align`:

```
void* align( std::size_t alignment,
             std::size_t size,
             void*& ptr,
             std::size_t& space );
```

  - Assuming that we have a space that starts from `ptr` and has size `space`;
  - Now we want to allocate an object with `size` and `alignment` on the space;
    - Assuming that it can be allocated on `new_ptr` on `new_space` (i.e. suitably aligned).
  - So `std::align` just modifies `ptr` to `new_ptr`, `space` to `new_space`, and returns `new_ptr`.
    - If space is too small, then nothing happens and `nullptr` is returned.

Supplementary

- For example:

```
class Buffer
{
    std::vector<std::byte> buffer_;
    std::size_t size_;
    void* ptr_;

public:
    Buffer(std::size_t size) : size_{ size }
    {
        buffer.resize(size);
        ptr_ = buffer.data();
    }

    template<typename T>
    void* Alloc()
    {
        auto addr = std::align(sizeof(T), alignof(T), ptr_, size_);
        ptr_ += sizeof(T);
        size_ -= sizeof(T);
        return addr;
    }
}
```

Supplementary

2. To maximize optimization, you can inform compiler that a pointer is aligned by `std::assume_aligned<N>(ptr)` since C++20.
   - It's UB if it's not aligned to `N`, quite like `[[assume]]`.
   - Since C++26 you can also add `std::is_sufficiently_aligned<N>(ptr)` to check precondition in debug mode.
   - For example:

```cpp
void Func(int* ptr)
{
    static constexpr std::size_t alignment = 64;
    assert(std::is_sufficiently_aligned<alignment>(ptr));
    std::assume_aligned<alignment>(ptr);
    // Then compilers may do optimization based on
    // assumption of 64 alignment.
}
```

Supplementary

- Note 2: since C++17, you can use trait `std::has_unique_object_representations` to check if same value representations of two objects lead to the same object representation.
  - For example, for `float`, two `NaN` are not distinguishable but may have different bits, so the trait returns `false`.
  - Particularly, for a `struct`, when there are padding bytes, then it definitely returns `false` since they are not part of value of `struct`.

  - This trait can be used to check whether it’s correct for a type to be hashed as a byte array.