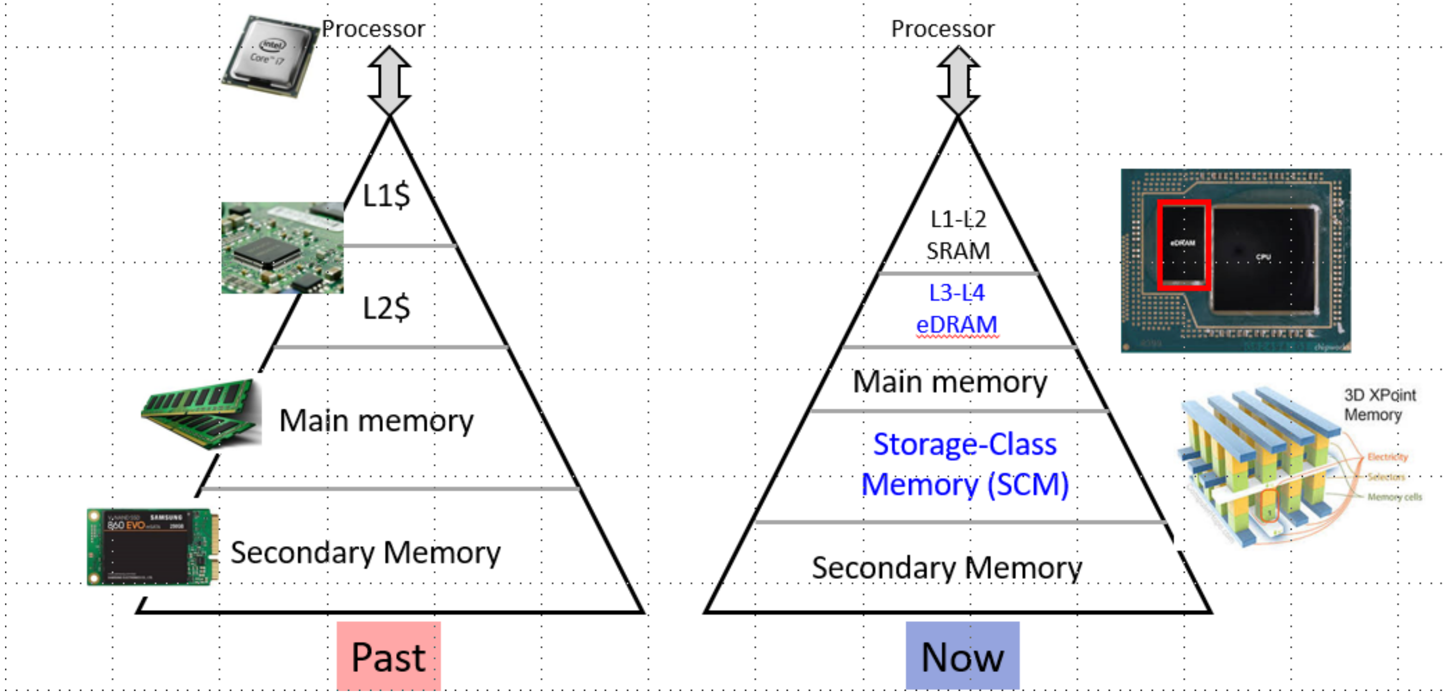
内存管理
Memory Management

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:

```cpp
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:

```cpp
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…

> - 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

- Now EBCO doesn't guarantee to happen:

```
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:

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

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

These three members are all aligned to 16.

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`.

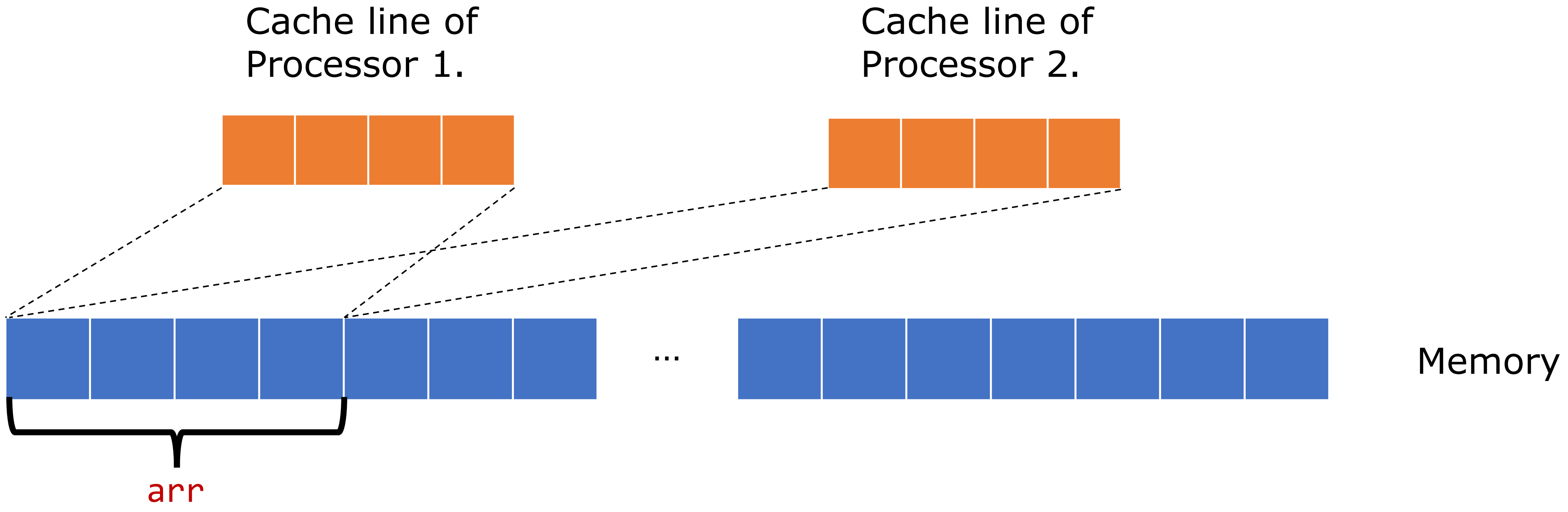
False Sharing

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

```cpp
// 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]++; }
```

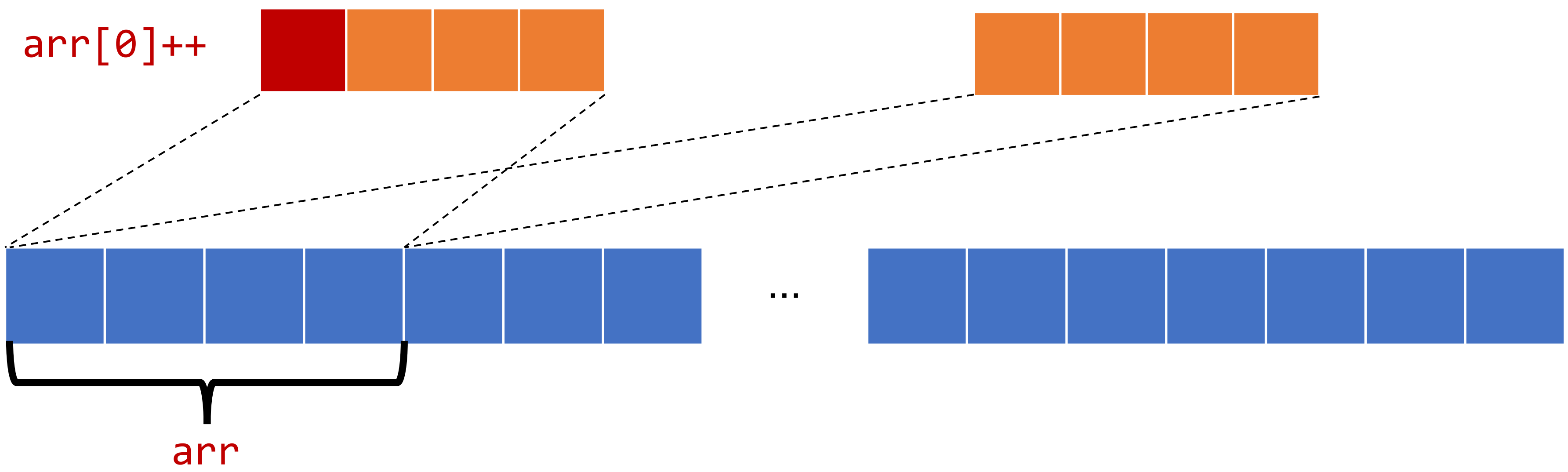
- However, due to limitation of computer architecture, such abstraction is wrong…
  - Cache on different processors has to obey coherence protocol like MESI.
  - To put it simply, when write happens on a cache line, it'll inform other processors whose cache also own this line to make it invalid.
  -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

False Sharing

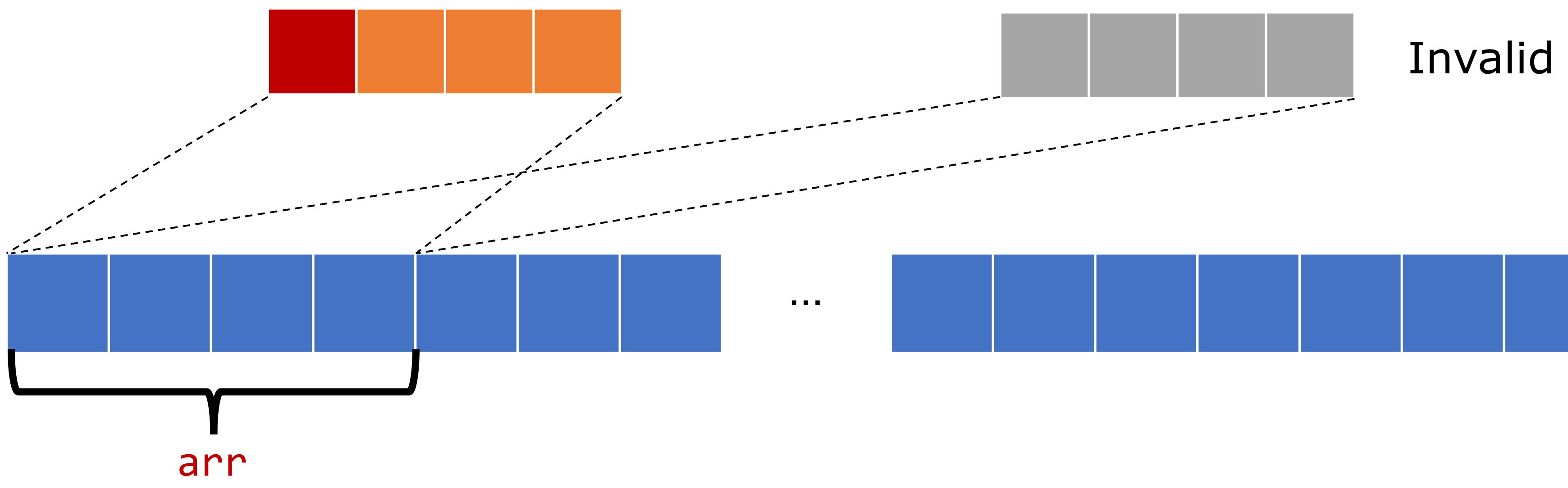


Illustrative animation for false sharing.
(Details may vary for different architectures)

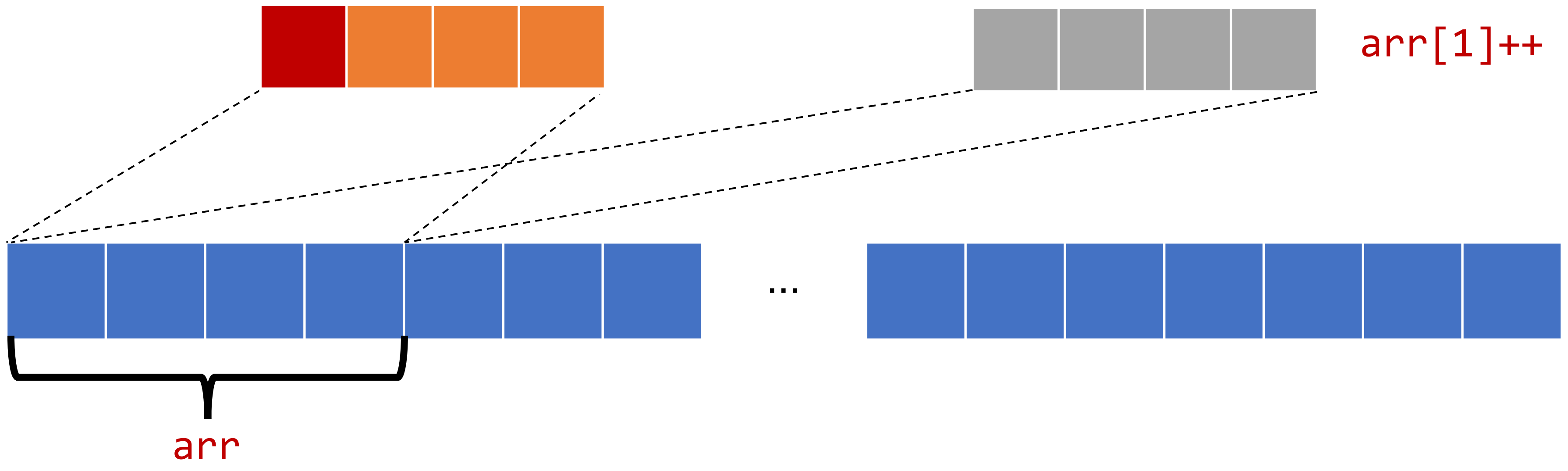
False Sharing



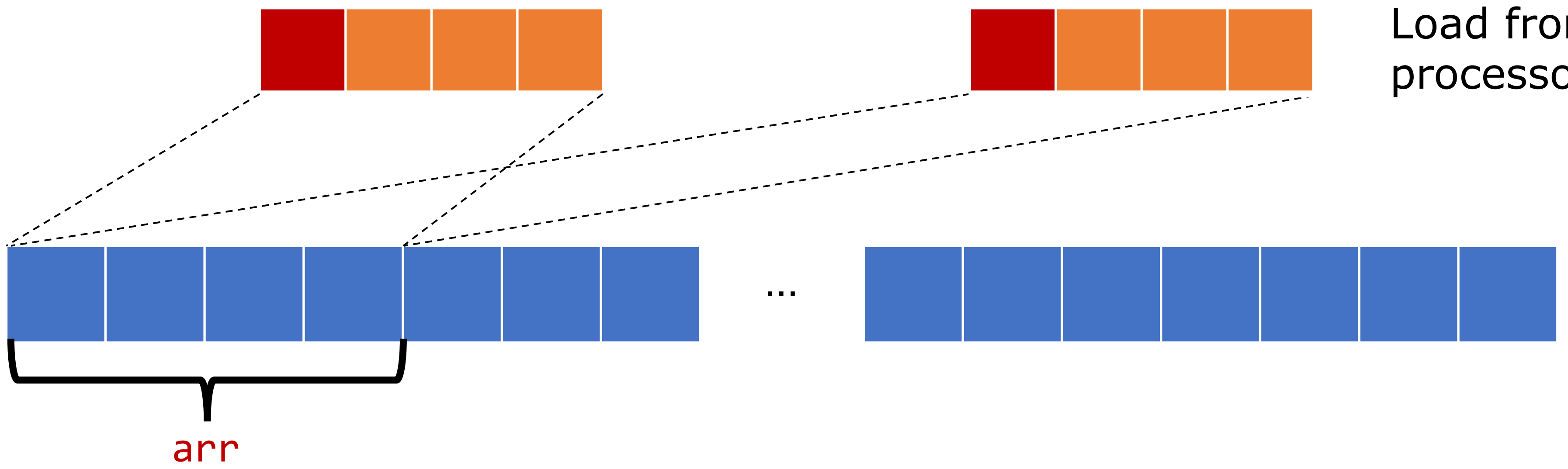
False Sharing



False Sharing

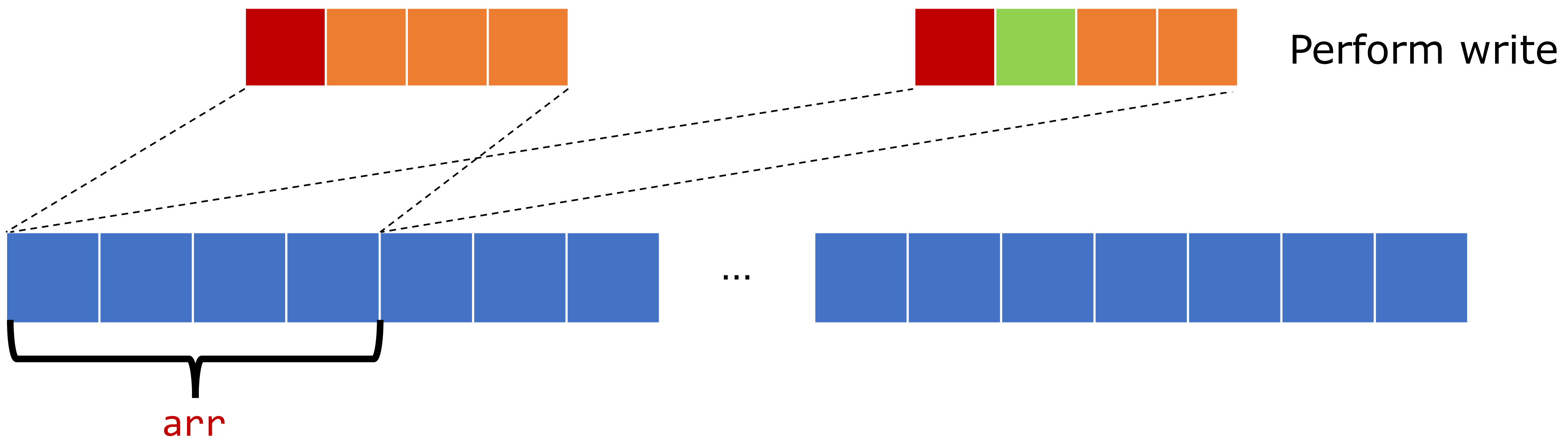


False Sharing

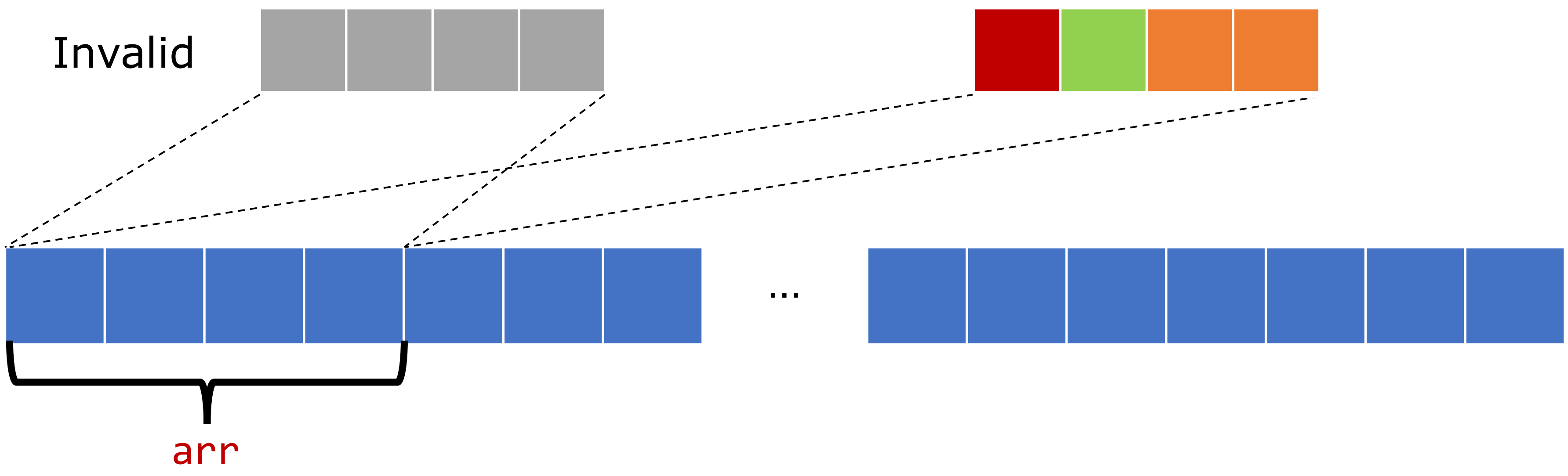


Load from other processors to read

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:

```cpp
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.

Memory Management

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

new/delete

- To combine allocation and construction, C++ uses **new-expression** to substitute `malloc` in C.
  - Roughly speaking, it calls two different functions:
  - Allocation new, which only allocates memory (quite like `malloc`).
  - Placement new, i.e. construct the object on memory.
- And similarly, **delete-expression** has two parts:
  - Destructor, i.e. destruct the object on memory.
  - Deallocation delete, which only deallocates memory (quite like `free`).
- C++ allows you to **override** (replace) the allocation new by `operator new` and the deallocation delete by `operator delete`.

new/delete

- Thus the most basic versions like `malloc` and `free` are as below:
  - You can override them in global scope (i.e. namespace `::`).

```
void* operator new  ( std::size_t count );
void* operator new[]( std::size_t count );
void operator delete  ( void* ptr ) noexcept;
void operator delete[]( void* ptr ) noexcept;
```

- Besides, you can provide class-specific allocation & deallocation:

```
void* T::operator new  ( std::size_t count );
void* T::operator new[]( std::size_t count );
void T::operator delete  ( void* ptr );
void T::operator delete[]( void* ptr );
```

  - Which is preferred than global override, and isn't required to be `noexcept`.
  - They are always static function, even if you don't add keyword `static`.

```cpp
void* operator new(std::size_t byteCnt)
{
    std::println("Called overrided operator new, size={}", byteCnt);
    if (auto ptr = malloc(byteCnt); ptr)
        return ptr;
    throw std::bad_alloc{};
}

void operator delete(void* ptr) noexcept
{
    std::println("Called overrided operator delete, ptr={}", ptr);
    free(ptr);
}

class Base
{
public:
    static void* operator new[](std::size_t byteCnt)
    {
        std::println("INSIDE CLASS: Called overrided operator new[], size={}", byteCnt);
        if (auto ptr = malloc(byteCnt); ptr)
            return ptr;
        throw std::bad_alloc{};
    }
```

Here we don’t add `static`, but it’s still static function. You can’t use `this` here.

```cpp
    void operator delete[](void* ptr) noexcept
    {
        std::println("INSIDE CLASS: Called overrided operator delete[], ptr={}", ptr);
        free(ptr);
    }
};
```

NOTE: before P3107 (DR23), `std::println` will use `std::string` and thus may need to use `operator new/delete`, causing infinite recursion. MS-STL has implemented this DR so it’s fine to do so.

```cpp
int main()
{
    auto a = new int{ 1 };
    auto b = new Base[2];
    std::println("PtrA = {}, val = {}; PtrB = {}",
                 (void*)a, *a, (void*)b);
    delete[] b;
    delete a;
    return 0;
}
```

```
Called overrided operator new, size=4
INSIDE CLASS: Called overrided operator new[], size=2
PtrA = 0x218395c8d10, val = 1; PtrB = 0x218395c8fd0
INSIDE CLASS: Called overrided operator delete[], ptr=0x218395c8fd0
Called overrided operator delete, ptr=0x218395c8d10
```

new/delete

- However, C++ also allows you to `delete basePtr`, which will call virtual dtor.
  - Ideally, it should call `Derived::operator delete` instead of `Base::operator delete`…
  - Let's try it!

```cpp
class Base
{
public:
    static void* operator new(std::size_t byteCnt)
    {
        std::println("Base: Called overrided operator new, size={}", byteCnt);
        if (auto ptr = malloc(byteCnt); ptr)
            return ptr;
        throw std::bad_alloc{};
    }

    void operator delete(void* ptr) noexcept
    {
        std::println("Base: Called overrided operator delete, ptr={}", ptr);
        free(ptr);
    }

    virtual ~Base() { std::println("Base dtor"); }
};
```

```cpp
class Derived : public Base
{
public:
    static void* operator new(std::size_t byteCnt)
    {
        std::println("Derived: Called overrided operator new, size={}", byteCnt);
        if (auto ptr = malloc(byteCnt); ptr)
            return ptr;
        throw std::bad_alloc{};
    }

    void operator delete(void* ptr) noexcept
    {
        std::println("Derived: Called overrided operator delete, ptr={}", ptr);
        free(ptr);
    }

    virtual ~Derived() { std::println("Derived dtor"); }
};
```

new/delete

```cpp
Base* ptr = new Derived;
delete ptr;
```

- The output is like:

```
Derived: Called overrided operator new, size=8
Derived dtor
Base dtor
Derived: Called overrided operator delete, ptr=0x2b96ff5c320
```

  - So `Derived::operator delete` is called, quite like virtual dtor!
  - But `operator delete` is `static`! How?
- Reason: compiler will generate a “deleting destructor”*.
  - That is, it will generate a new virtual function:
    - For a normal object, just use normal dtor;
    - For `delete ptr`, it will call this new function.

```cpp
virtual void DestroyWhenDelete(void* addr)
{
    this->~Derived();
    Derived::operator delete(addr);
}
```

- With virtual dispatch, we can extract more information from the type to improve `malloc`-like version!

*: This is implementation-defined; here we use method of Itanium ABI. See this blog for details.

new/delete

- Before going on, let’s do some recap…
- In ICS, we’ve written a very basic allocation strategy:
  - Allocate memory block that’s slightly larger than requested, then store block size and pointer to next block alongside it.
  - However, many metadata will never change after allocation, which will pollute cache line.
- So in modern memory allocators, it’s much more complicated…
  - Roughly speaking, a common way is to split memory into bins **indexed by approximate size**.
  - And metadata may record more info:

```
// Thread local data
struct mi_tld_s {
  unsigned long long  heartbeat;
  bool                recurse;
  mi_heap_t*          heap_backing;
  mi_heap_t*          heaps;
  mi_segments_tld_t   segments;
  mi_os_tld_t         os;
  mi_stats_t          stats;
};
```

Adopted from mimalloc.

Sized-delete

- And in C++, we can almost always know object type exactly…
  - So we can know size of object!
- To facilitate optimization, C++ introduces *size-aware delete* (also called *sized-delete*).
  - Global sized delete is provided in C++14, while class-specific one is from C++11.

```
void operator delete  ( void* ptr, std::size_t sz ) noexcept;
void operator delete[]( void* ptr, std::size_t sz ) noexcept;
```

  - For a naïve example:

```
void operator delete(void* ptr, std::size_t size) noexcept
{
    std::println("Derived: Called overrided operator delete, ptr={}, size={}",
        ptr, size);
    free(ptr);
}
```

```
int* ptr = new int;
delete ptr;
```

```
Called overrided operator new, size=4
Derived: Called overrided operator delete, ptr=0x145540d4ce0, size=4
```

Sized-delete

```
JEMALLOC_ALWAYS_INLINE
void
sizedDeleteImpl(void* ptr, std::size_t size) noexcept {
    if (unlikely(ptr == nullptr)) {
        return;
    }
    LOG("core.operator_delete.entry", "ptr: %p, size: %zu", ptr, size);

    je_sdallocx_noflags(ptr, size);

    LOG("core.operator_delete.exit", "");
}
```

- Note 1: some practical example:
  - Like in jemalloc:

```
void
operator delete(void *ptr, std::size_t size) noexcept {
    sizedDeleteImpl(ptr, size);
}
```

- Note 2: compilers are free to choose sized-delete or normal delete.
  - So, programmer should **always provide both of them**.
  - For gcc/msvc/clang (clang needs `-fsized-deallocation` flag):
    - For global override, it will prefer sized version when it exists.
    - For class-specific override, it will prefer normal delete when it exists (since you can easily know its size by `sizeof`).

Aligned new/delete

- But these overloads don’t specify alignment…
  - Before C++17, over-aligned types may be not correctly handled (normally compiler warning in e.g. `-Wall`).

```cpp
// ptr is possibly not aligned to 1024!
A* ptr = new A;
delete ptr;
```

```cpp
struct alignas(1024) A
{
    int a;
};
```

- Since C++17, alignment-aware new/delete are introduced.
  - Here `std::align_val_t` is scoped enumeration as tag.
  - For type whose alignment requirement exceeds macro `__STDCPP_DEFAULT_NEW_ALIGNMENT__`, alignment-aware overloads are preferred.
  - Of course, you can override them too.

```cpp
void* operator new  ( std::size_t count, std::align_val_t al );
void* operator new[]( std::size_t count, std::align_val_t al );
```

```cpp
void operator delete  ( void* ptr, std::align_val_t al ) noexcept;
void operator delete[]( void* ptr, std::align_val_t al ) noexcept;
void operator delete  ( void* ptr, std::size_t sz ) noexcept;
void operator delete[]( void* ptr, std::size_t sz ) noexcept;
void operator delete  ( void* ptr, std::size_t sz,
                        std::align_val_t al ) noexcept;
void operator delete[]( void* ptr, std::size_t sz,
                        std::align_val_t al ) noexcept;
```

- For class-specific ones:

```cpp
void* T::operator new  ( std::size_t count, std::align_val_t al );
void* T::operator new[]( std::size_t count, std::align_val_t al );
```

```cpp
void T::operator delete  ( void* ptr, std::align_val_t al );
void T::operator delete[]( void* ptr, std::align_val_t al );
void T::operator delete  ( void* ptr, std::size_t sz );
void T::operator delete[]( void* ptr, std::size_t sz );
void T::operator delete  ( void* ptr, std::size_t sz, std::align_val_t al );
void T::operator delete[]( void* ptr, std::size_t sz, std::align_val_t al );
```

C11/C++17 provides `aligned_alloc` similarly; but MS-STL doesn’t provide `aligned_alloc` since Windows doesn’t provide ability to allocate aligned memory and thus must over-allocate and align manually. Therefore, it cannot be freed correctly by `free`; instead, `_aligned_alloc` and `_aligned_free` should be used.

new/delete

- Note 1: all new-overloads has nothrow variants:

```
void* operator new  ( std::size_t count, const std::nothrow_t& tag );
void* operator new[]( std::size_t count, const std::nothrow_t& tag );
void* operator new  ( std::size_t count, std::align_val_t al,
                      const std::nothrow_t& tag ) noexcept;
void* operator new[]( std::size_t count, std::align_val_t al,
                      const std::nothrow_t& tag ) noexcept;
```

- Note 2: essentially, new-expression `new(args…) Type{…}` will call `operator new(size(, align), args…)`.
  - The arguments before `args…` are usually determined by compilers, while the latter are specified by users.

new/delete

*: Placement is abused in the context of new/delete; here it just means to provide additional parameters in `new(…)`, which includes nothrow variant and placement-new variant.

- That's why you can use:
  - `new(std::nothrow) Type`, for there exists nothrow variant.
  - `new(ptr) Type`, for there exists placement-new variant.
    - But they can't be overridden by users.

Non-allocating placement allocation functions

```
void* operator new  ( std::size_t count, void* ptr );
void* operator new[]( std::size_t count, void* ptr );
```

- More generally, you can provide customized arguments for user-defined placement* allocation new:
  - Class-specific ones also exist, omitted here.

```
void* operator new  ( std::size_t count, /* args... */ );
void* operator new[]( std::size_t count, /* args... */ );
void* operator new  ( std::size_t count,
                      std::align_val_t al, /* args... */ );
void* operator new[]( std::size_t count,
                      std::align_val_t al, /* args... */ );
```

```cpp
void operator delete  ( void* ptr, args... );
void operator delete[]( void* ptr, args... );
```

- Plus placement deallocation delete:
  - Each user-defined `new` must has a matching user-defined `delete`; when **constructor throws**, `new` memory will be freed by corresponding `delete`.
  - Otherwise memory leak! For example (omit sized-delete):

```cpp
struct S
{
    S() = default;
    S(int) { throw std::runtime_error{ "Hi" }; }

    void* operator new(std::size_t byteCnt, const std::string& msg)
    {
        std::println("New {}: {}", msg, byteCnt);
        return ::operator new(byteCnt);
    }

    // Non-placement deallocation function:
    void operator delete(void* ptr)
    {
        std::println("Delete {}", ptr);
        ::operator delete(ptr);
    }

    void operator delete(void* ptr, const std::string& msg)
    {
        std::println("Delete {}: {}", msg, ptr);
        ::operator delete(ptr);
    }
};
```

```cpp
int main()
{
    S* p = new ("123") S;
    delete p;

    try {
        p = new("442") S{1};
    } catch(const std::exception& ex) {
        std::println("Exception: {}", ex.what());
    }
}
```

```
New 123: 1
Delete 0x5a75def022c0
New 442: 1
Delete 442: 0x5a75def022c0
Exception: Hi
```

Prevent memory leak.

Only failed new-expression will call corresponding placement delete!

new/delete

- And similarly, for nothrow new, you need to customize placement delete…

```
void operator delete  ( void* ptr, const std::nothrow_t& tag ) noexcept;
void operator delete[]( void* ptr, const std::nothrow_t& tag ) noexcept;
void operator delete  ( void* ptr, std::align_val_t al,
                        const std::nothrow_t& tag ) noexcept;
void operator delete[]( void* ptr, std::align_val_t al,
                        const std::nothrow_t& tag ) noexcept;
```

- Finally, if a placement allocation corresponds to a non-placement deallocation, then compile error.

```
struct S
{
    // Placement allocation function:
    static void* operator new(std::size_t, std::size_t);

    // Non-placement deallocation function:
    static void operator delete(void*, std::size_t);
};

S* p = new (0) S; // error: non-placement deallocation function matches
                  //        placement allocation function
```

This is sized delete.

new/delete

- Note 3: for the default thrown `operator new`, it will call new handler when allocation fails.
  - As if:

```cpp
void* operator new(std::size_t byteCnt)
{
    auto ptr = operator new(byteCnt, std::nothrow);
    while (ptr == nullptr)
    {
        auto handler = std::get_new_handler();
        if (handler == nullptr)
            throw std::bad_alloc{};
        handler();
        ptr = operator new(byteCnt, std::nothrow);
    }
    return ptr;
}
```

The default new handler is just `nullptr`, so it will `throw` `std::bad_alloc` directly.

new/delete

- You can customize it by `std::set_new_handler(…)` in `<new>` (thread-safe), and the handler is expected to:
  1. Make more memory available (so after calling handler, allocation retry may succeed);
  2. Terminate the program (e.g. by `std::terminate`);
  3. Throw exception derived from `std::bad_alloc`, or `std::set_new_handler(nullptr)`.
  - Return value: previous handler.
  - For example:

```cpp
void handler()
{
    std::cout << "Memory allocation failed, terminating\n";
    std::set_new_handler(nullptr);
}

int main()
{
    std::set_new_handler(handler);
    try
    {
        while (true)
        {
            new int[1000'000'000ul]();
        }
    }
    catch (const std::bad_alloc& e)
    {
        std::cout << e.what() << '\n';
    }
}
```

new/delete

- Note 4: C++20 introduces class-specific destroying-delete.

```
void T::operator delete( T* ptr, std::destroying_delete_t );
```

```
void T::operator delete( T* ptr, std::destroying_delete_t,
                         std::align_val_t al );
```

```
void T::operator delete( T* ptr, std::destroying_delete_t, std::size_t sz );
```

```
void T::operator delete( T* ptr, std::destroying_delete_t,
                         std::size_t sz, std::align_val_t al );
```

  - Which will be preferred over all other overloads.
  - delete-expression will call destroying-delete directly, without calling dtor.
    - That is, it’s duty of the destroying-delete to call dtor.
  - Array doesn’t have this overload.
- Note 5: it should be thread-safe to call `operator new/delete`.

new/delete in coroutine

- Special example: control allocation of coroutine.
  - Coroutine will allocate its state/frame by `new`;
  - C++ allows you to customize `operator new/delete` of `promise_type` to control such allocation!
- It's specially treated so not exactly same as normal class-specific allocation/deallocation.
  - Class-specific ones need lots of overloads to cover every possible case;
  - But `promise_type` only needs to define a few for compiler to choose!
  - For new, it only needs: `void* operator new ( std::size_t count );`
  - For delete, it only needs: `void operator delete ( void* ptr, std::size_t sz ) noexcept;`
    - When this overload doesn't exist, it needs: `void operator delete ( void* ptr ) noexcept;`

new/delete in coroutine

Memory resource will be covered in later sections.

- For example:

```cpp
class CoroTask {
    inline static std::array<std::byte, 200000> memory;
    // covered in t
    inline static std::pmr::monotonic_buffer_resource buffer{
        memory.data(), memory.size(), std::pmr::null_memory_resource()
    };
    inline static std::pmr::synchronized_pool_resource mempool{ &buffer };

public:
    struct promise_type {
        void* operator new(std::size_t size) {
            return mempool.allocate(size);
        }
        void operator delete(void* ptr, std::size_t size) {
            mempool.deallocate(ptr, size);
        }
    };
}
```

new/delete in coroutine

- Note 1: when defining `get_return_object_on_allocation_failure`, you should make `operator new` act as if nothrow instead of defining nothrow variant.
  - For example:

```cpp
void* operator new(std::size_t size) {
    return new(std::nothrow) std::byte[size];
}
```

- Note 2: compilers are allowed to omit your `operator new/delete` when performing HALO.
  - So theoretically, one way to ensure HALO is to only declare `operator new/delete` without definition, so allocating on heap will lead to link error.
- Note 3: `operator new` is allowed to accept parameters of coroutine.
  - A naïve example:
  - And it's preferred if exist.

```cpp
struct promise_type {
    void* operator new(std::size_t sz, int, const std::string&) {
        return mempool.allocate(sz);
    }
};

CoroTask coro1(int a, std::string s); // 这个协程会使用重载的operator new.
CoroTask coro2(int a); // 这个协程不会使用。
```

new/delete in coroutine

- Take `std::generator` as an example:

```
void* operator new( std::size_t size )
    requires std::same_as<Allocator, void> ||
             std::default_initializable<Allocator>;
```

```
template< class Alloc, class... Args >
void* operator new( std::size_t size, std::allocator_arg_t,
                    const Alloc& alloc, const Args&... );
```

For member coroutine.

```
template< class This, class Alloc, class... Args >
void* operator new( std::size_t size, const This&, std::allocator_arg_t,
                    const Alloc& alloc, const Args&... );
```

  - Implementation may then allocate more bytes than `size`, then put allocator on additional space.
  - The `delete` can extract allocator from the frame to do deallocation.

```
void operator delete( void* ptr, std::size_t n ) noexcept;
```

new/delete in coroutine

- Use it by passing additional parameters.

```
template< class Alloc, class... Args >
void* operator new( std::size_t size, std::allocator_arg_t,
                    const Alloc& alloc, const Args&... );
```

```
using Alloc = std::allocator<void>;
template<typename T>
using Generator = std::generator<T, void, Alloc>;

Generator<char> TestImpl(std::allocator_arg_t, Alloc alloc,
                         std::string str)
{
    for (auto ch : str)
        co_yield ch;
}

Generator<char> Test(std::string str)
{
    return TestImpl(std::allocator_arg, Alloc{}, std::move(str));
}
```

new/delete

- Final note: in shared library, global override of `operator new/delete` should be paid special attention.
  - Reason: if each shared library has its own override, it may be unclear which one is used.
    - For example, when A is loaded, its memory is allocated by its `operator new`;
    - And B is loaded, then `operator delete` is replaced;
    - And when A frees its memory, it uses `operator delete` of B, causing unknown results.
    - The behaviors are totally implementation-defined.
  - In static library, this will cause link error for symbol conflict.

Memory Management

Smart Pointers

Overview

- Similar to every RAII type, smart pointer can be used to prevent memory leak by releasing resource in dtor.

```
Vector(const Vector& another){
    auto size = another.size();
    std::unique_ptr<T[]> arr{ new T[size] };
    std::ranges::copy(another.first_, another.last_, arr.get());
    first_ = arr.release();
    last_ = end_ = first_ + size;
}
```

In Lecture 7 *Error Handling*, Section “Exception safety”.

- Generally, smart pointers represent kind of “ownership”.
  - `std::unique_ptr` represents unique ownership; only one can destroy it.
  - `std::shared_ptr` represents shared ownership; the last holder will destroy it.
  - …
- So when someone doesn’t need ownership, it’s enough to use raw pointer. Do NOT abuse smart pointer.
  - We’ll talk more about this later.

Memory Management

- Smart Pointers
  - unique_ptr
    - indirect and polymorphic (C++26)
  - shared_ptr
  - weak_ptr
  - Adaptors

All of them are defined in `<memory>`.

unique_ptr

- As it's easy and we've taught it briefly, we first list APIs and add some important notes.

Move-only, i.e. have move ctor & assignment, no copy ctor & assignment.

Give up ownership and set nullptr; Return original pointer.

Destroy original resource; replace it with parameter ptr (by default nullptr).

Member functions

(constructor)	constructs a new unique_ptr (public member function)
(destructor)	destructs the managed object if such is present (public member function)
operator=	assigns the unique_ptr (public member function)
Modifiers	
release	returns a pointer to the managed object and releases the ownership (public member function)
reset	replaces the managed object (public member function)
swap	swaps the managed objects (public member function)
Observers	
get	returns a pointer to the managed object (public member function)
get_deleter	returns the deleter that is used for destruction of the managed object (public member function)
operator bool	checks if there is an associated managed object (public member function)
Single-object version, unique_ptr<T>	
operator* operator->	dereferences pointer to the managed object (public member function)

unique_ptr

- Note 1: we know that unique_ptr can also handle array by specifying T[].

```
Vector(const Vector& another){
    auto size = another.size();
    std::unique_ptr<T[]> arr{ new T[size] };
```

  - Such partial specialization is slightly different:
    1. Instead of having operator->/*, it has operator[] as if accessing an array.
    2. Of course, if will call delete[] by default.
  - This also makes it impossible to do CTAD for ambiguity; given a pointer, it cannot determine whether it's unique_ptr<T> or unique_ptr<T[]>.
- Note 2: if you want to denote const T* (i.e. point to immutable object), you should use unique_ptr<const T> instead of const unique_ptr<T>.

unique_ptr

std::unique_ptr

Defined in header <memory>

```cpp
template<
    class T,
    class Deleter = std::default_delete<T>
> class unique_ptr;
```
(1) (since C++11)

```cpp
template <
    class T,
    class Deleter
> class unique_ptr<T[], Deleter>;
```
(2) (since C++11)

- Note 3: more generally, `unique_ptr` can handle any resource by customized deleter.
  - A deleter needs to define:
    1. A type named `pointer` (if it doesn't exist, `T*` will be used);
       - Which is stored and managed inside `unique_ptr`.
    2. `operator()` to do destroy operation (e.g. `delete` in `std::default_delete<T>`, and `delete[]` in specialization `std::default_delete<T[]>`).
  - For example:

```cpp
struct BufferArrayDeleter
{
    unsigned int n = 1;

    using pointer = unsigned int*;
    void operator()(pointer buffer) const noexcept {
        glDeleteBuffers(n, buffer);
    }
};
```

Remove GPU resources related to descriptor `buffer`.

```cpp
unsigned int size = 5;
std::unique_ptr<unsigned int[]> glBufferBuffer{
    new unsigned int[size]
};
glGenBuffers(size, glBufferBuffer.get());
BufferArrayDeleter deleter{ size };
std::unique_ptr<unsigned int[], BufferArrayDeleter> glBuffer2{
    glBufferBuffer.get(), deleter
};
std::println("Buffer[0]: {}", glBuffer2[0]);
```

This `unique_ptr` manages memory.

This `unique_ptr` manages OpenGL descriptor.

unique_ptr

- Another example?

```cpp
struct BufferDeleter
{
    using pointer = unsigned int;
    void operator()(pointer buffer) const noexcept {
        glDeleteBuffers(1, &buffer);
    }
};
```

1. `unique_ptr` now stores `unsigned int` instead of a pointer;
2. `operator()` will be called in dtor.

```cpp
unsigned int buffer = 0;
glGenBuffers(1, &buffer);
std::unique_ptr<unsigned int, BufferDeleter> glBuffer{ buffer };
```

- But it cannot use some methods (like `.release()`), since it will try to assign `nullptr` as empty resource…
- To make it fully compatible, you should make pointer fulfill *NullablePointer*.
    - And support `operator*/->` additionally if needs to use these `operator*/->` of `unique_ptr`.

unique_ptr*

- For example:

```cpp
struct BufferDeleter
{
    class pointer
    {
        unsigned int buffer_;
    public:
        pointer(unsigned int buffer = 0) : buffer_{ buffer } {}
        pointer(std::nullptr_t) : pointer{} {}
        // To pass by value, use friend instead of member function.
        friend auto operator<=>(pointer, pointer) noexcept = default;
        friend bool operator==(pointer, pointer) noexcept = default;
        explicit operator bool() const noexcept { return buffer_ != 0; }

        auto GetBufferPtr() const noexcept { return &buffer_; }

        // Only needed when you need to use unique_ptr.operator*/->.
        auto operator->() const noexcept { return GetBufferPtr(); }
        // To mimic shallow const semantics of pointer...
        auto& operator*() const noexcept {
            return const_cast<unsigned int&>(buffer_);
        }
    };

    void operator()(pointer p) const noexcept {
        glDeleteBuffers(1, p.GetBufferPtr());
    }
};
```

```cpp
unsigned int buffer = 0;
glGenBuffers(1, &buffer);
std::unique_ptr<unsigned int, BufferDeleter> glBuffer{ buffer };
std::println("{}", *glBuffer);
```

Then all methods of `std::unique_ptr` are defined with e.g. pointer class. We don't dig into that and just check cppreference.

Quite complex…

If you only need some general RAII wrapper, you can write it yourself instead of using `std::unique_ptr` with customized deleter (especially if pointer is some customized class).

unique_ptr

- Note 4: dtor will actually check empty state first.
  - So if your deleter cannot process `nullptr` correctly, it's okay.

```
constexpr ~unique_ptr();
```

1 *Effects*: Equivalent to: `if (get()) get_deleter()(get());`

[*Note 1*: The use of `default_delete` requires `T` to be a complete type. — *end note*]

2 *Remarks*: The behavior is undefined if the evaluation of `get_deleter()(get())` throws an exception.

- Note 5: you can also use `std::make_unique<T>(Args…)` to do construction.

```
// Pointer to vector that has 10 elements, all of them are 1.
auto ptr = std::make_unique<std::vector<int>>(10, 1);
// Equiv. to:
std::unique_ptr<std::vector<int>> ptr{ new std::vector<int>(10, 1) };
```

Initialized by () instead of {}

  - For array, only size can be specified and all elements are value-initialized.
    - E.g. here all elements are 0.

```
std::size_t size = 10;
auto arr = std::make_unique<int[]>(size);
```

unique_ptr

```
A a{ std::unique_ptr<int>{ new int{ 1 } },
     std::unique_ptr<int>{ new int{ 2 } } };
```

- Before C++17, `std::make_unique` can prevent subtle memory leak caused by indeterministic evaluation order.
  - For example, order may be `new int{1}` -> `new int{2}` -> construct `unique_ptr`;
  - So when `new int{2}` throws, memory leak may still happen.
- Since C++17, we know that function parameters are evaluated in a non-overlapping way, so this problem won't happen at all.

- And sometimes it may be unnecessary to do value initialization…
  - For example, we'll read binary data from network, so we don't need to assign all elements 0.
  - Then you can use `std::make_unique_for_overwrite` since C++20.
    - The essential difference is just `new int()` v.s. `new int`.

```
// Allocated elements have random values.
auto ptr = std::make_unique_for_overwrite<int>();
std::size_t size = 10;
auto arr = std::make_unique_for_overwrite<int[]>(size);
```

unique_ptr

- Back to our previous claim…
  - “When someone doesn’t need ownership, it’s enough to use raw pointer. Do NOT abuse smart pointer.”

- Use function parameter as example…
  - Raw pointer (`T*`)
  - `std::unique_ptr<T>`
  - `std::unique_ptr<T>&`
  - `std::unique_ptr<T>&&`
  - `const std::unique_ptr<T>&`

  which one to choose?

unique_ptr

1. In most cases, raw pointer is enough…
    - Precondition: except for `nullptr`, pointed object is valid.
    - And function read / write the object by pointer.
        - By contrast, it should rarely manipulate lifetime like by `delete`.
        - Observation instead of ownership.

- For example:

```cpp
void func(A* ptr)
{
    if(ptr == nullptr)
        return;
    ptr->c = 1.0f;
    ptr->d.push_back(1);
    // etc.
}
```

```cpp
A a;
std::unique_ptr<A> b{ new A };

func(&a);
func(b.get());
func(nullptr);
```

- This function does NOT care about where `ptr` comes from (stack, heap, or static segment, etc.); it only observes.

unique_ptr

2. By contrast, `std::unique_ptr<T>` means to hold the ownership;
   - So the caller will give up its ownership.
   - And the function may transfer ownership to others, or just let it destroy automatically when exiting function.

- For example:

```cpp
class B
{
public:
    B(std::unique_ptr<A> init_a) : a { std::move(init_a) } {}
private:
    std::unique_ptr<A> a;
};
```

```cpp
std::unique_ptr<A> ptr{ new A };

B b{ std::move(ptr) };
B b2{ std::make_unique<A>() }; // 填入参数，我们这里默认构造
B b3{ std::unique_ptr<A>{ new A{} } }; // 和上一行等价
```

`std::unique_ptr<T>&&` is quite similar, except that when you don't move inside function, the caller won't release its ownership.

While by taking value as parameter, ownership will be definitely released.

unique_ptr

3. For `std::unique_ptr<T>&`…
   - Generally, for a ref parameter `U&`, what we want to do is to modify the parameter itself.
   - So similarly, `std::unique_ptr<T>&` means to modify caller's `unique_ptr`.
   - For example, set a new object ownership:

```
void func2(std::unique_ptr<A>& ptr2)
{
    ptr2 = std::unique_ptr<A>{ new A };
    // 等价于ptr2.reset(new A);
}
```

   - Of course, it can read & write content, and transfer ownership to others;
     - But if it only needs to undertake these duty, it's unnecessary to use `&`.
     - Which is quite like `T*` v.s. `T**`.
4. Finally, for `const std::unique_ptr<T>&`, since its read-only features are same as `T*`, this form is useless.

Memory Management

- Smart Pointers
  - unique_ptr
    - indirect and polymorphic (C++26)
  - shared_ptr
  - weak_ptr
  - Adaptors

PImpl

- Before going on, we first introduce a technique called **p**ointer to **impl**ementation idiom (pimpl).
- When programming in multiple files, for a class:
  - We need to expose in header files:
    - Data members;
    - Declaration of methods and (non-inline) static variables;
  - And hide in source files:
    - Definition of methods and static variables.
- So when we:
  1. Want to add / remove methods;
  2. Want to modify data members, no matter change type or add new ones.
  - We have to code in header files, and all related files need to re-compile…

PImpl

  - But ideally, when public members remain the same, what is exposed to users is unchanged; other files should not re-compile.
- PImpl tries to solve this problem.
  - Class in header only owns a pointer to its members, and expose public interface.
- For example, a naïve example of normal implementation:

```
class SomeComplexClass
{
    int a_;
    float b_;

    float InnerProd_() const noexcept;

public:
    SomeComplexClass(int a, float b) : a_{ a }, b_{ b } {}
    float Sum() const noexcept { return static_cast<float>(a_) + b_; }
    float Prod() const noexcept;
};
```

When we add `float cacheSum_, cacheProd_;` and remove `InnerProd_`, then other parts need to re-compile…

```
#include "test.hpp"

float SomeComplexClass::InnerProd_() const noexcept
{
    return a_ * b_;
}

float SomeComplexClass::Prod() const noexcept
{
    return InnerProd_();
}
```

PImpl

- If we use PImpl:

```cpp
class SomeComplexClass
{
    struct Impl;
    Impl *impl_;

public:
    SomeComplexClass(int a, float b);
    ~SomeComplexClass();
    float Sum() const noexcept;
    float Prod() const noexcept;
};
```

```cpp
#include "test.hpp"

struct SomeComplexClass::Impl
{
    int a;
    float b;

    float InnerProd() const noexcept;
};
// Equiv. to private method InnerProd_
float SomeComplexClass::Impl::InnerProd() const noexcept
{
    return a * b;
}

SomeComplexClass::SomeComplexClass(int a, float b) : impl_{ new Impl{ a, b } }
{
}

SomeComplexClass::~SomeComplexClass()
{
    delete impl_;
}

float SomeComplexClass::Sum() const noexcept
{
    return static_cast<float>(impl_->a) + impl_->b;
}

float SomeComplexClass::Prod() const noexcept
{
    return impl_->InnerProd();
}
```

When we add `float cacheSum_, cacheProd_;` and remove `InnerProd_`, then only this source file will be modified. Thus we only need to re-compile a single file, and relink.

PImpl

- We notice that pimpl has many variants.
  - For example, previous code doesn't manage inheritance well.
    1. The derived class needs to allocate new space for its own members, causing memory fragmentation;
    2. You cannot change protected APIs freely, as it will change header file.
  - We can then improve like:
    1. Write `BaseImpl` class into another header, which is only included for inheritance (thus re-compilation is restricted in limited files only);
    2. The `DerivedImpl` class then inherits `BaseImpl`;
    3. `Base` exposes pointer to `BaseImpl` as `protected`;
    4. And finally, `Derived` inherits `Base`, and assigns `new`'ed `DerivedImpl` to `Base` in ctor; when it needs to use `DerivedImpl`, just `static_cast` it.
  - Also, if you want to use interface of `Class` in `ClassImpl`, you can also add a `Class*` in `ClassImpl*` to point back.
  - etc…

The above two variants are adopted in QT and renamed as "[d-pointer & q-pointer]".

PImpl

- Pros:
  - Reduce build time **significantly** when project is large.
  - Maintain binary compatibility.
    - Normally, when data members change, object layout will also change;
      - Then new-version header files + old-version shared library will crash; users have to re-link.
    - However, by pimpl, users just pass the pointer, and how to process it is completely determined by library.
      - As long as users don’t use new public APIs, they don’t need to re-link.
  - Completely hide members that is originally be in public header files, so no privacy concerns.
- Of course, everything comes with a cost…

PImpl

- Cons:
  - Initialization overhead: need an additional dynamic allocation;
  - Runtime overhead: all member access need one more indirect addressing;
  - Cannot inline simple methods, since header files don’t know members;
  - Cannot utilize default special member functions (e.g. default copy ctor, default dtor, etc.);
  - Const incorrectness: `const Class` object has `ClassImpl* const` instead of `const ClassImpl*`, so `const` methods in `Class` can access non-`const` methods in `ClassImpl`.
    - Which then needs additional attention to maintain correctness.

PImpl

- We can notice that we are managing pointer manually…
  - It seems very proper to use `std::unique_ptr`!
- But when we compile, it fails…

```
D:\Softwares\Visual Studio\VC\Tools\MSVC\14.44.35207\include\memory(3308):
error C2338: static_assert failed: 'can't delete an incomplete type'
D:\Softwares\Visual Studio\VC\Tools\MSVC\14.44.35207\include\memory(3309):
warning C4150: 删除指向不完整“SomeComplexClass::Impl”类型的指针；没有调用析
构函数
```

```cpp
#include <memory>

class SomeComplexClass
{
    struct Impl;
    std::unique_ptr<Impl> impl_;

public:
    SomeComplexClass(int a, float b);
    float Sum() const noexcept;
    float Prod() const noexcept;
};
```

  - Reason: it's UB to `delete` incomplete type that has a non-trivial dtor.
    - So `std::default_delete` enhances safety, which will emit error directly inside `operator()`.
    - And default dtor is inline inside class, which is thus equivalent to call `operator()` when type is incomplete.
  - Solution: write default definition in source file!

PImpl

In header file:

```
public:
    SomeComplexClass(int a, float b);
    SomeComplexClass(SomeComplexClass &&) noexcept;
    SomeComplexClass &operator=(SomeComplexClass &&) noexcept;
    ~SomeComplexClass();
```

In source file:

```
// We can see complete definition now, generate dtor here.
SomeComplexClass::SomeComplexClass(SomeComplexClass &&) noexcept = default;
SomeComplexClass &SomeComplexClass::operator=(SomeComplexClass &&) noexcept =
    default;
SomeComplexClass::~SomeComplexClass() = default;
```

- For example:
  - Move ctor and move assignment are quite similar*.
- Sometimes default move may be not our expectation, as it breaks abstraction of pimpl.
  - It just points to implementation, so it should perform value semantics (i.e. all operations should happen in the underlying object).
  - For example, for copy ones, we'll write like:

```
SomeComplexClass::SomeComplexClass(const SomeComplexClass &another)
    : impl_{ new Impl{ *another.impl_ } }
{
}

SomeComplexClass &SomeComplexClass::operator=(const SomeComplexClass &another)
{
    *impl_ = *another.impl_;
}
```

*: strictly speaking, move ctor of `unique_ptr` doesn't require complete type. However, C++ regulates that ctor may call dtor of subobjects (see [class.base.init]), so all compilers reject inline `=default`.

PImpl

```cpp
// This noexcept is faked up.
SomeComplexClass::SomeComplexClass(SomeComplexClass &&another) noexcept
    : impl_{ new Impl{ std::move(*another.impl_) } }
{
}
SomeComplexClass &SomeComplexClass::operator=(
    SomeComplexClass &&another) noexcept
{
    *impl_ = std::move(*another.impl_);
}
```

- Similarly, for move:
  - You can call underlying move ctor & assignment if you want.
  - Then moved-from interface points to a moved-from implementation, instead of getting `nullptr`.

- Though easier to implement compared with raw pointer, `std::unique_ptr` is still kind of inconvenient.
  - You need to reimplement many methods, like copy, comparison, etc.
    - As default ones will copy / compare /… pointers, which is pointer-semantics instead of value-semantics.
  - And const-correctness is still under concern.

std::indirect

- Since C++26, we can use `std::indirect` to solve it!
  - It's a value-semantic `std::unique_ptr`, i.e. major operations just call methods of the underlying object.
    - Copy ctor & assignment;
    - Comparison;
    - Hash.
  - And some special methods:
    - swap: swap the pointer;
    - Move ctor: transfer the **pointer**.
      - Thus, the stored pointer of the moved-from object will be `nullptr`, which can be checked by `.valueless_after_move()`.
    - Move assignment: swap pointers, and destroy resource of the other.

std::indirect

- For ctor:

Construct T by forwarded v or args or initializer list + args.

std::indirect<T, Allocator>::indirect

`constexpr explicit indirect();`	(1)	(since C++26)
`template< class U = T >` `constexpr explicit indirect( U&& v );`	(3)	(since C++26)
`template< class... Args >` `constexpr explicit indirect( std::in_place_t, Args&&... args );`	(5)	(since C++26)

`template< class I, class... Args >`
`constexpr explicit indirect( std::in_place_t, std::initializer_list<I> ilist,`
`Args&&... args );`
	(7)	(since C++26)

`constexpr indirect( const indirect& other );`	(9)	(since C++26)
`constexpr indirect( indirect&& other ) noexcept;`	(11)	(since C++26)

- Default ctor value-initializes the underlying object instead of owning `nullptr`.
- Every ctor has an allocator-aware variant; we'll talk about allocator later.

```
template< class U = T >
constexpr explicit indirect( std::allocator_arg_t, const Allocator& a,
                             U&& v );
```

std::indirect

- And finally you can also use `operator->/*` to access.
  - All methods will maintain const correctness, e.g. here `const std::indirect<T>` will access by `const T*`.
- For pimpl, it's then very easy to implement basic operations:
  - Just `=default` all of them in source file.

```cpp
SomeComplexClass::SomeComplexClass(const SomeComplexClass &) = default;
SomeComplexClass &SomeComplexClass::operator=(const SomeComplexClass &) =
    default;
SomeComplexClass::SomeComplexClass(SomeComplexClass &&) noexcept = default;
SomeComplexClass &SomeComplexClass::operator=(SomeComplexClass &&) noexcept =
    default;
SomeComplexClass::~SomeComplexClass() = default;
std::strong_ordering SomeComplexClass::operator<=>(
    const SomeComplexClass &) const noexcept = default;
bool SomeComplexClass::operator==(const SomeComplexClass &) const noexcept =
    default;
```

```cpp
SomeComplexClass(const SomeComplexClass &);
SomeComplexClass &operator=(const SomeComplexClass &);
SomeComplexClass(SomeComplexClass &&) noexcept;
SomeComplexClass &operator=(SomeComplexClass &&) noexcept;
~SomeComplexClass();

// If it needs to support comparison
std::strong_ordering operator<=>(const SomeComplexClass &) const noexcept;
bool operator==(const SomeComplexClass &) const noexcept;
```

std::indirect

- We notice that the real effects are slightly different if allocators of two `std::indirect` are unequal.
  - For example, for move ctor `std::indirect<T> a = std::move(b)`:
    - When they have “equal” allocators, then `a` just takes pointer of `b`;
    - But when they have “unequal” allocators, it will be like:
      - `a` uses its allocator to allocate memory;
      - Construct `T` with `std::move(*b)`.
- We’ll cover them later…

std::polymorphic

- Finally, `std::indirect<T>` can only handle `T`, though it stores `T*`.
- `std::polymorphic<Base>` is to correctly handle inheritance!
  - You can store any `Derived` object inside it.

std::polymorphic<T, Allocator>::polymorphic

`constexpr explicit polymorphic();`	(1)	(since C++26)
`template< class U = T >` `constexpr explicit polymorphic( U&& v );`	(3)	(since C++26)
`template< class U, class... Args >` `constexpr explicit polymorphic( std::in_place_type_t<U>, Args&&... args );`	(5)	(since C++26)

`template< class U, class I, class... Args >`
`constexpr explicit polymorphic( std::in_place_type_t<U>,`
`std::initializer_list<I> ilist,`
`Args&&... args );`
	(7)	(since C++26)

`constexpr polymorphic( const polymorphic& other );`	(9)	(since C++26)
`constexpr polymorphic( polymorphic&& other ) noexcept;`	(11)	(since C++26)

Here `U` must be same as or publicly derived from `T`.
Arguments are used to construct `U` too.

std::polymorphic

- Copy ctor: construct `Derived` object, where `Derived` is same as the underlying copied object.
  - It will NOT slice, i.e. it doesn't construct `Base` for `std::polymorphic<Base>`.
- Copy assignment: copy-and-swap, still to prevent slicing problem.
  - For two `std::polymorphic<Base>`, assuming the underlying objects are `Derived1` and `Derived2`, it will store pointer to a copy of `Derived2`.
- Move ctor / assignment: same as `std::indirect`, by taking pointer and swap-and-destroy.
- Dtor: it will call dtor of `Derived` directly, even if dtor of `Base` is not `virtual`.

- Since types may vary, other methods are limited:

Observers

`operator->` `operator*`	accesses the owned value (public member function)
`valueless_after_move`	checks if the polymorphic is valueless (public member function)
`get_allocator`	returns the associated allocator (public member function)

Modifiers

`swap`	exchanges the contents (public member function)