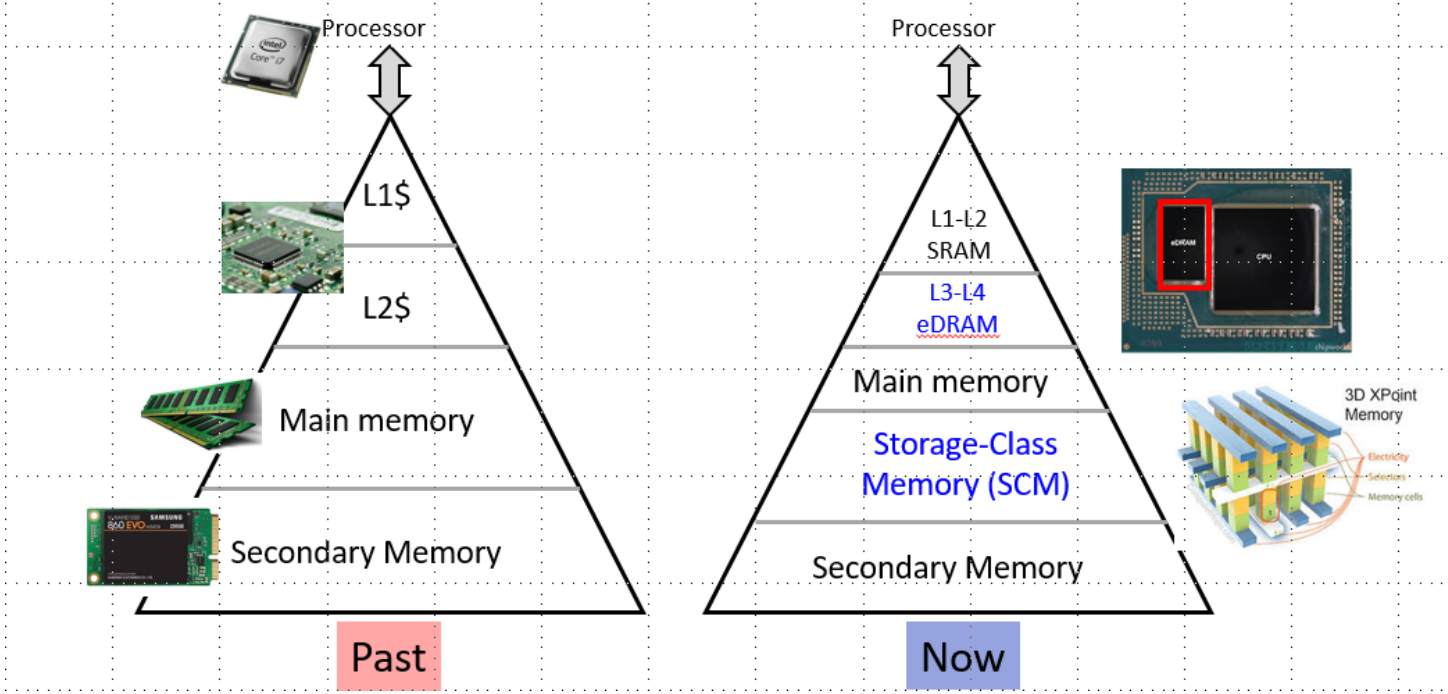
内存管理
Memory Management

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.

```
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…

  - 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:

```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.

- 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:

```
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.

```
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`.

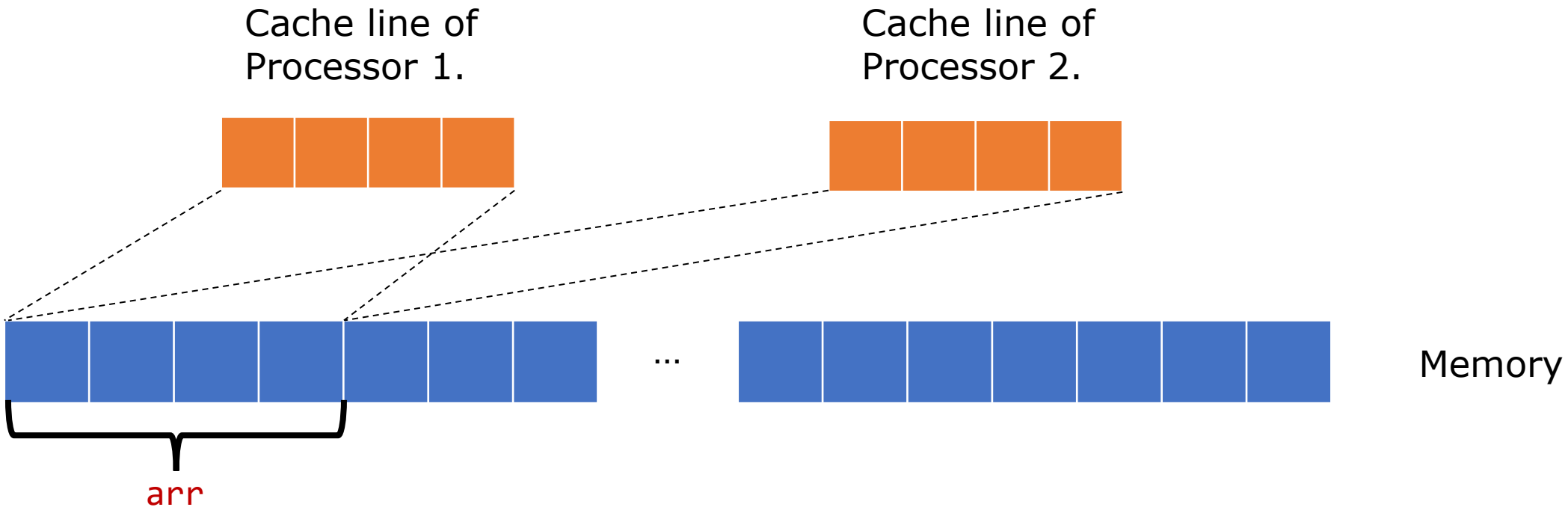
False Sharing

- 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]++; }
```

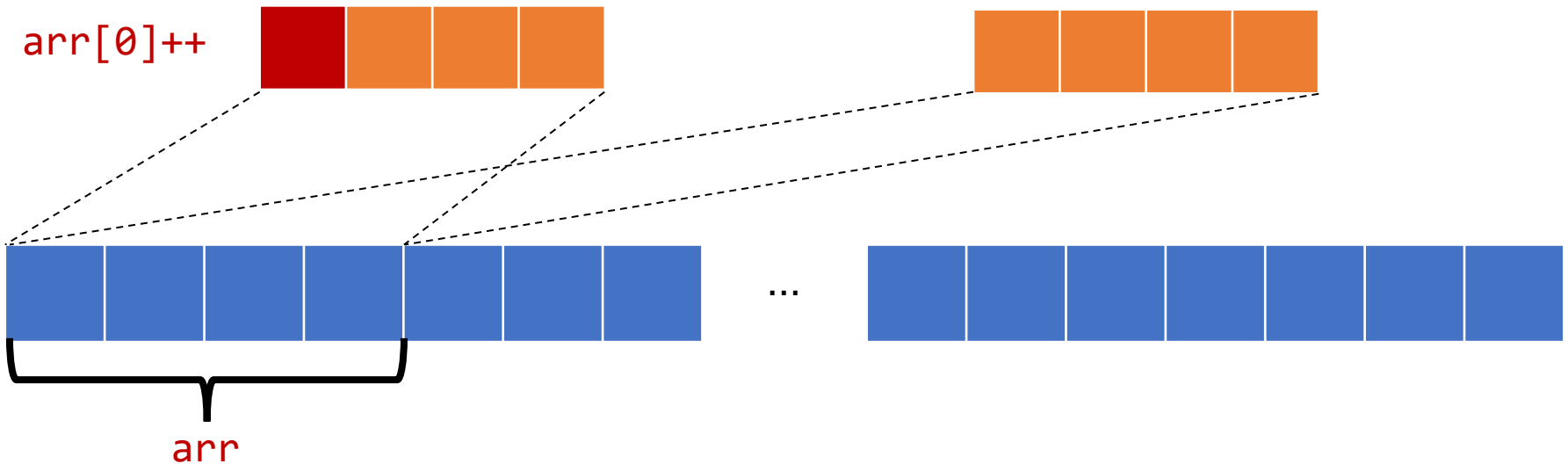
- However, due to limitation of computer architecture, such abstraction is wrong…
  - Cache on different processors has to obey coherence protocol like MESI.
  - To put it simply, when write happens on a cache line, it’ll inform other processors whose cache also own this line to make it invalid.
  -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

False Sharing

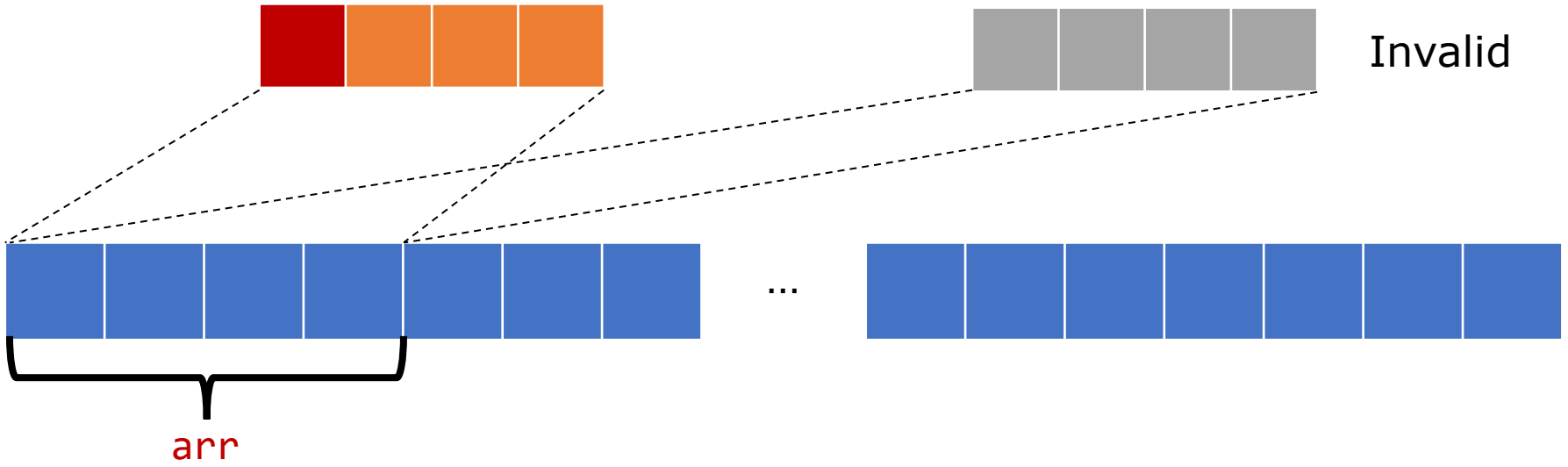


Illustrative animation for false sharing.
(Details may vary for different architectures)

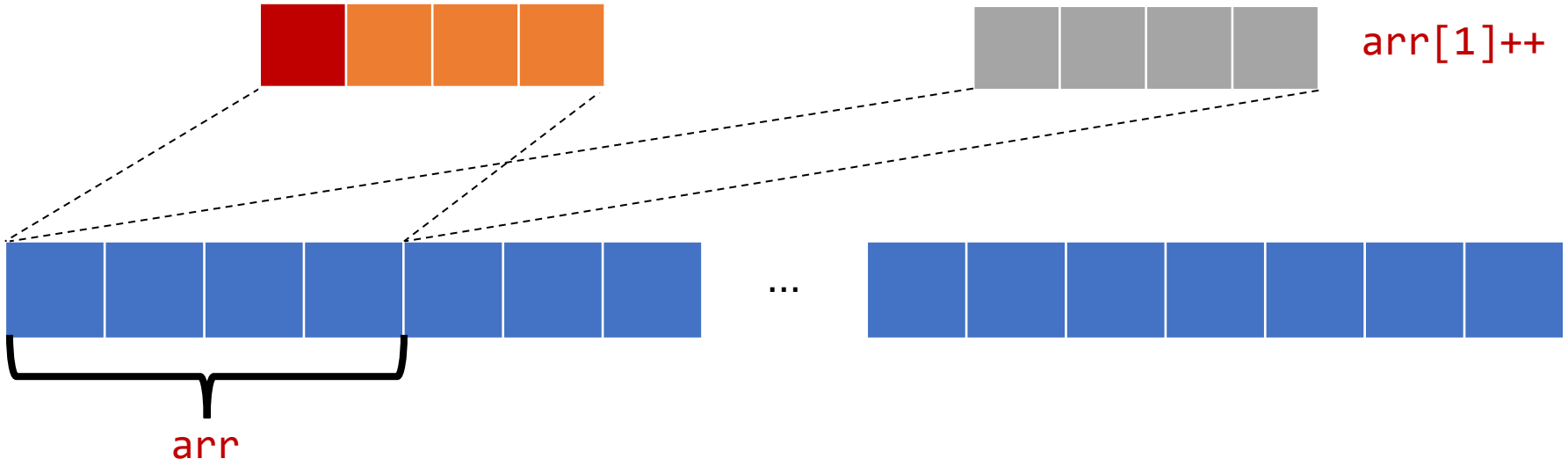
False Sharing



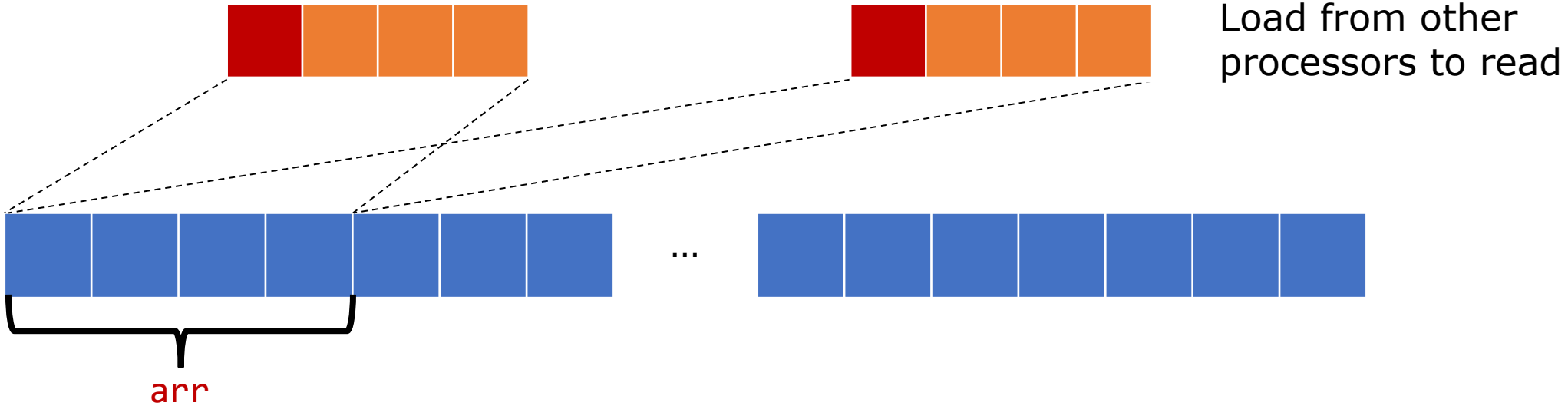
False Sharing



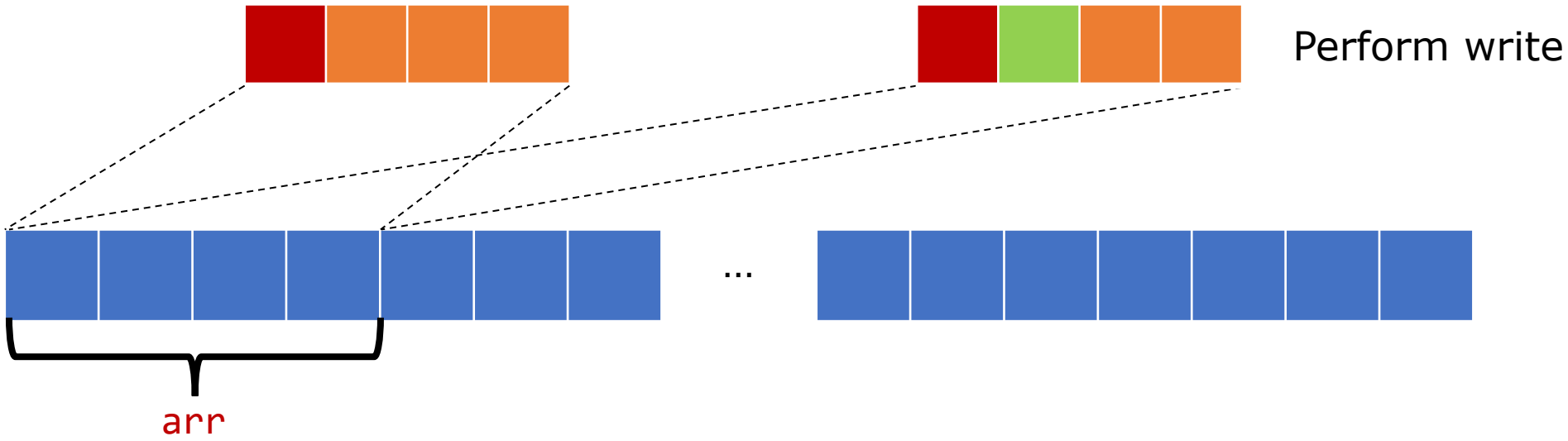
False Sharing



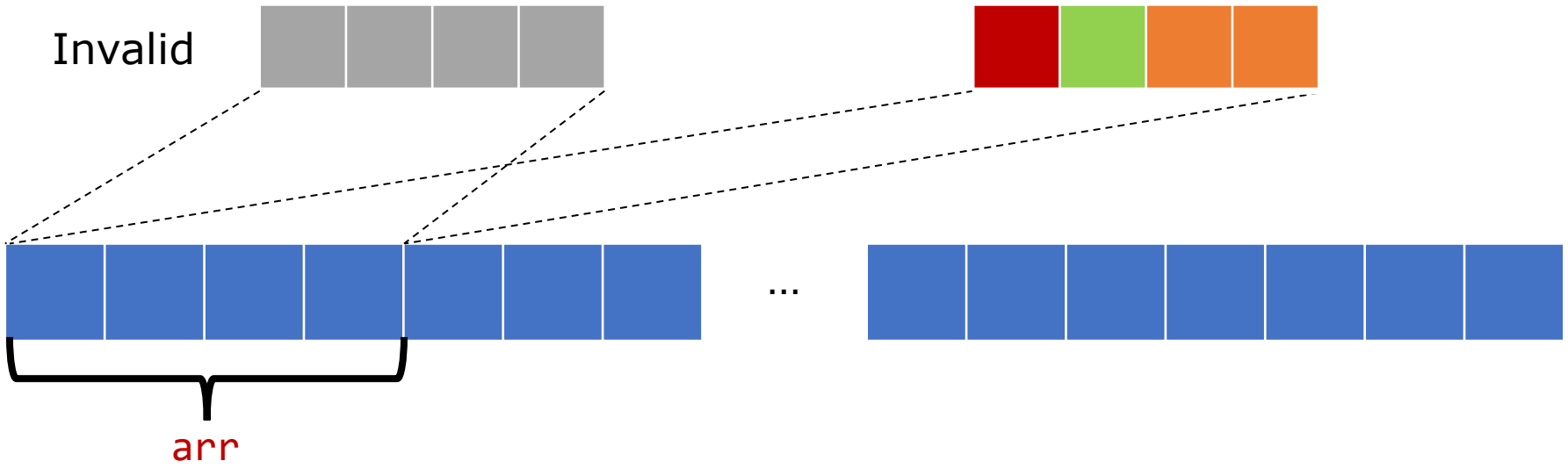
False Sharing



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.

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

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

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

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

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

- The output is like:
  - 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.

```
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

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

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

```
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 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`).

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

```
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.

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

```
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:

```
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 );
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... */ );
```

```
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):

```
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);
    }
};
```

```
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:

```
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:

```
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:

```
class CoroTask {
    inline static std::array<std::byte, 200000> memory;

    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:

```
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.

```
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[]`.
  - 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[]>`.

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

- 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>

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

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:

```
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`.

```
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?

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

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

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

- 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:

```
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());
    }
};
```

```
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:

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

```
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:

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

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:

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

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

```
#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:

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)
```

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

- 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.

```
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;
```

```
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;
```

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)

Memory Management

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

shared_ptr

- `unique_ptr` represents unique ownership to a resource.
  - Which is thus only moveable.
- Sometimes, there exist resources that are shared by many…
  - Everyone has the right to destroy it, and will destroy it if it's the last holder.
- Then we can use `std::shared_ptr`.
  - For example:

```
void Work(std::shared_ptr<Resource> ptr)
{
    // Use ptr to do work...
}

void Dispatch(std::size_t cnt)
{
    auto ptr = std::make_shared<Resource>(...);
    // Every detached thread holds its ownership.
    // i.e. They must keep it valid when using.
    for (std::size_t i = 0; i < cnt; i++)
        std::thread{ Work, ptr }.detach();
    return;
}
```

You should NOT construct two `shared_ptr` with the same raw pointer to share ownership;

Instead, use copy ctor!

shared_ptr

- Essentially, it maintains an atomic reference counter;
  - Just like what we state in stop token handling.
  - So unlike `unique_ptr` that merely stores original pointer, `shared_ptr` stores pointer to a *control block*.
- An example control block:
  - Ctor: assign counter as 1*;
  - Copy ctor: set `blockPtr` as `another.blockPtr`, increase counter;
  - Move ctor: set `blockPtr` as `another.blockPtr`, set `another.blockPtr` as `nullptr`.
  - Assignment: copy-and-swap idiom;
  - Dtor: decrease counter; when counter reaches 0, destroy the resource.

```
template<typename T>
struct PlainBlock
{
    std::atomic<int> cnt;
    T* ptr;
};

template<typename T>
class PlainSharedPtr
{
    PlainBlock<T>* block;
};
```

*: initial counter value is slightly more complex and will be covered later.

shared_ptr

- For ctor:

std::shared_ptr<T>::shared_ptr

```
constexpr shared_ptr() noexcept;                                          (1)
constexpr shared_ptr( std::nullptr_t ) noexcept;                          (2)
template< class Y >
explicit shared_ptr( Y* ptr );                                            (3)
template< class Y, class Deleter >
shared_ptr( Y* ptr, Deleter d );                                          (4)
template< class Deleter >
shared_ptr( std::nullptr_t ptr, Deleter d );                              (5)
template< class Y, class Deleter, class Alloc >
shared_ptr( Y* ptr, Deleter d, Alloc alloc );                             (6)
template< class Deleter, class Alloc >
shared_ptr( std::nullptr_t ptr, Deleter d, Alloc alloc );                 (7)
```

`Y*` must be convertible to `T*`.

Question: unlike `unique_ptr<T, Deleter>`, `shared_ptr` only has a single template parameter `<T>`; How can it initialize with deleter and allocator?

shared_ptr

- By type erasure! (covered in Lecture 12 *Advanced Template*)
  - For example:

```
struct BlockBase
{
    std::atomic<int> cnt;
    virtual void Destroy() = 0;
};

template<typename T>
struct PlainBlock : BlockBase
{
    T* ptr;
    void Destroy() override { delete ptr; }
}; // Y* uses this block;
```

```
template<typename T, typename Deleter>
struct PlainBlockWithDeleter : BlockBase
{
    T* ptr;
    Deleter d;
    void Destroy() override { d(ptr); }
}; // Y*, Deleter uses this block

class SharedPtr
{
    BlockBase* blockPtr;
};
```

Of course, here you can utilize EBO to compress Deleter.

  - So for some ctor, even if you pass `nullptr`, it still needs to allocate.

shared_ptr

- Besides ctor, you can also use `make_shared` to do construction.
  - Or `allocate_shared` if you want an allocator.

```
template< class T, class... Args >
shared_ptr<T> make_shared( Args&&... args );

template< class T, class Alloc, class... Args >
shared_ptr<T> allocate_shared( const Alloc& alloc, Args&&... args );
```

- Unlike assigning pointer like `PlainBlock`, it stores object inside control block directly.
  - It only needs to allocate once, so:
  1. Reduce runtime allocation overhead;
  2. Prevent memory fragmentation.

```
template<typename T>
struct MergeBlock : BlockBase
{
    T obj;
    void Destroy() override { }
};
```

- Thus: **in usual cases, prefer `make_shared` and `allocate_shared` over ctor to get new `shared_ptr`.**

shared_ptr

- Note 1: since C++17, `shared_ptr` also supports array semantics correctly*.

```
std::shared_ptr<int[]> arr{ new int[10] };
std::cout << arr[0] << "\n";
```

`operator[]` (C++17)

  - And since C++20, `make_shared` and `allocate_shared` also add related overloads:

When `T` is unbounded, e.g. `std::shared_ptr<int[]>`.

```
template< class T >
shared_ptr<T> make_shared( std::size_t N );
```

Value-initialized for each element;

```
template< class T >
shared_ptr<T> make_shared( std::size_t N, const std::remove_extent_t<T>& u );
```

Each element is copy-initialized with `u`.

When `T` is bounded, e.g. `std::shared_ptr<int[5]>`.

```
template< class T >
shared_ptr<T> make_shared();
```

```
template< class T >
shared_ptr<T> make_shared( const std::remove_extent_t<T>& u );
```

*: Before it may succeed to compile, but it's incorrect. See StackOverflow for details.

shared_ptr

- Note 2: since C++20, `for_overwrite` variants are also added.

```
template< class T >
shared_ptr<T> make_shared_for_overwrite();

template< class T >
shared_ptr<T> make_shared_for_overwrite( std::size_t N );

template< class T, class Alloc >
shared_ptr<T> allocate_shared_for_overwrite( const Alloc& alloc );

template< class T, class Alloc >
shared_ptr<T> allocate_shared_for_overwrite( const Alloc& alloc,
                                             std::size_t N );
```

- Note 3: some newly-added type aliases since C++17:

Member type	Definition
element_type	T (until C++17)
	std::remove_extent_t<T> (since C++17)
weak_type (since C++17)	std::weak_ptr<T>

Count sharing

- Sometimes, two `shared_ptr` share the same ownership, though they're not convertible.
  - For example:

```
void PartialWork(std::shared_ptr<int> ptr) { }
struct Resource
{
    float b;
    int a;
};
```

```
void Dispatch(std::size_t cnt)
{
    auto ptr = std::make_shared<Resource>(1.0f, 2);
    for (std::size_t i = 0; i < cnt; i++)
        std::thread{ Work, ptr }.detach();
    for (std::size_t i = 0; i < cnt; i++)
        std::thread{ PartialWork, /* ??? */ }.detach();
    return;
}
```

  - `PartialWork` only needs to use `a`, but it still has ownership (i.e. needs to keep the resource valid).
- To solve that, `shared_ptr` introduces so-called *aliasing ctor*.

Count sharing

- For example:

```
for (std::size_t i = 0; i < cnt; i++)
    std::thread{ Work, ptr }.detach();
for (std::size_t i = 0; i < cnt; i++)
    std::thread{ PartialWork, std::shared_ptr<int>{ ptr, &(ptr->a) } }.detach();
```

We have a practical example for `LockFreeStack` in homework.

  - Then it shares the same control block with `ptr`;
  - Note that when reference count reaches 0, the original object is destroyed.
- Particularly, `operator<=>` for `shared_ptr` are compared against stored pointer;
  - So aliased pointer may be unequal to the original pointer (i.e. here `ptr.get() != &(ptr->a)`, assuming `int*` and `Resource*` are comparable).
- If you want owner-based comparison, you should use `.owner_xx`.
  - In owner-based meaning, aliased pointer are always equal to the original pointer (as they share the same ownership).
  - Practically it just compares address of control block.

Count sharing

std::shared_ptr<T>::owner_before

```
template< class Y >
bool owner_before( const shared_ptr<Y>& other ) const noexcept;
```

```
template< class Y >
bool owner_before( const std::weak_ptr<Y>& other ) const noexcept;
```

- For example:

```
struct Foo
{
    int n1;
    int n2;
    Foo(int a, int b) : n1(a), n2(b) {}
};

int main()
{
    auto p1 = std::make_shared<Foo>(1, 2);
    std::shared_ptr<int> p2(p1, &p1->n1);
    std::shared_ptr<int> p3(p1, &p1->n2);

    std::cout << std::boolalpha
              << "p2 < p3 " << (p2 < p3) << '\n'
              << "p3 < p2 " << (p3 < p2) << '\n'
              << "p2.owner_before(p3) " << p2.owner_before(p3) << '\n'
              << "p3.owner_before(p2) " << p3.owner_before(p2) << '\n';
```

The first two will be `true` + `false`;
the last two will be two `false`.

Count sharing

```
template< class T >
struct owner_less<std::shared_ptr<T>>;                    (2)

template< class T >
struct owner_less<std::weak_ptr<T>>;                      (3)
```

- Note 1: it’s wrapped as functor too:
  - However, it can only compare two pointer with same types;
  - But ownership may be shared among different types (like previous `shared_ptr<int>` and `shared_ptr<Resource>`).
  - So since C++17, template `operator()` is added by `std::owner_less<void>`.
    - Which is quite like transparent operator.

std::owner_less<void>::operator()

```
template< class T, class U >
bool operator()( const std::shared_ptr<T>& lhs,                    (since C++17)
                 const std::shared_ptr<U>& rhs ) const noexcept;

template< class T, class U >
bool operator()( const std::shared_ptr<T>& lhs,                    (since C++17)
                 const std::weak_ptr<U>& rhs ) const noexcept;

template< class T, class U >
bool operator()( const std::weak_ptr<T>& lhs,                      (since C++17)
                 const std::shared_ptr<U>& rhs ) const noexcept;

template< class T, class U >
bool operator()( const std::weak_ptr<T>& lhs,                      (since C++17)
                 const std::weak_ptr<U>& rhs ) const noexcept;
```

Compares `lhs` and `rhs` using owner-based semantics. Effectively calls `lhs.owner_before(rhs)`.

The ordering is strict weak ordering relation.

`lhs` and `rhs` are equivalent only if they are both empty or share ownership.

Count sharing

- Note 2: since C++26, owner-based equality and hash are also added.

Member methods:

`owner_before`
`owner_hash` (C++26)
`owner_equal` (C++26)

Functors:

`owner_less` (C++11)
`owner_hash` (C++26)
`owner_equal` (C++26)

  - But `owner_hash` and `owner_equal` are not template at all; they use template method `operator()` directly.

- Note 3: aliasing ctor adds a rvalue overload in C++20.
  - which will transfer pointer directly like move ctor.

```
template< class Y >
shared_ptr( const shared_ptr<Y>& r, element_type* ptr ) noexcept;                (8)

template< class Y >
shared_ptr( shared_ptr<Y>&& r, element_type* ptr ) noexcept;                     (8)    (since C++20)
```

Count sharing

- Note 4: if you want to share ownership with cast pointer, you can use `std::xx_pointer_cast`:
  - Equiv. to cast raw pointer, and then call aliasing ctor.

std::static_pointer_cast, std::dynamic_pointer_cast, std::const_pointer_cast, std::reinterpret_pointer_cast

Defined in header <memory>

`template< class T, class U > std::shared_ptr<T> static_pointer_cast( const std::shared_ptr<U>& r ) noexcept;`	(1)	(since C++11)
`template< class T, class U > std::shared_ptr<T> static_pointer_cast( std::shared_ptr<U>&& r ) noexcept;`	(2)	(since C++20)
`template< class T, class U > std::shared_ptr<T> dynamic_pointer_cast( const std::shared_ptr<U>& r ) noexcept;`	(3)	(since C++11)
`template< class T, class U > std::shared_ptr<T> dynamic_pointer_cast( std::shared_ptr<U>&& r ) noexcept;`	(4)	(since C++20)
`template< class T, class U > std::shared_ptr<T> const_pointer_cast( const std::shared_ptr<U>& r ) noexcept;`	(5)	(since C++11)
`template< class T, class U > std::shared_ptr<T> const_pointer_cast( std::shared_ptr<U>&& r ) noexcept;`	(6)	(since C++20)
`template< class T, class U > std::shared_ptr<T> reinterpret_pointer_cast( const std::shared_ptr<U>& r ) noexcept;`	(7)	(since C++17)
`template< class T, class U > std::shared_ptr<T> reinterpret_pointer_cast( std::shared_ptr<U>&& r ) noexcept;`	(8)	(since C++20)

Self-sharing

- In some cases, a class may need to keep “ownership of itself” inside a member method.
  - For example:

```
class TcpConnection
{
public:
    void AsyncRead()
    {
        std::thread {
            [](std::shared_ptr<TcpConnection> ptr){
                /* Read from TCP stream */
            }, /* this? */
        }.detach();
    }
};
```

Simplified example of `boost::asio::tcp_connection::async_xx()`.

  - We need code like:

```
void Work()
{
    TcpConnection conn;
    conn.AsyncRead();
    return;
}
```

But now when `Work` returns, `conn` will be destructed so code in new thread is UB.

Self-sharing

- Two factors to make it correct:
  1. `TcpConnection` cannot be normally constructed; users should only be able to get its `shared_ptr`.
     - Which is quite easy; just hide all ctors as `private` and expose them as `static shared_ptr<TcpConnection> Create(Args…)`.
  2. In `AsyncRead`, you must get copy of such `shared_ptr`.
     - However, you cannot write `shared_ptr<TcpConnection>{this}`;
     - We've said that it's wrong to construct two `shared_ptr` with the same raw pointer to share ownership.
- To solve this problem, you can let `T` inherits from `std::enable_shared_from_this<T>`.

- For example:

```cpp
class TcpConnection : public std::enable_shared_from_this<TcpConnection>
{
    TcpConnection(int val) {}
public:
    // Why not make_shared: we'll improve this in homework...
    template<typename... Args>
    static std::shared_ptr<TcpConnection> Create(Args&&... args)
    {
        return std::shared_ptr<TcpConnection>{
            new TcpConnection{ std::forward<Args>(args)... }
        };
    }

    void AsyncRead()
    {
        std::thread{
            [](std::shared_ptr<TcpConnection> ptr) {
                /* Read from TCP stream */
            }, shared_from_this()
        }.detach();
    }
};
```

1. Now users can only get `shared_ptr` to `TcpConnection`;
2. To share ownership inside member method, use the inherited `.share_from_this()`.

Note that `enable_shared_from_this` is implemented by `std::weak_ptr` (covered later), and also has `.weak_from_this()` since C++17.

```cpp
void Work()
{
    auto conn = TcpConnection::Create(1);
    conn->AsyncRead();
    // Same as auto conn2 = conn; here.
    auto conn2 = conn->shared_from_this();
    return;
}
```

shared_ptr

*: Time-of-check To Time-of-Use, meaning that when you use the value, it's already different from the value you checked. (so that precondition of code later may not hold anymore).

- Finally just list APIs:

1. Ctor & assignment also accept `std::unique_ptr&&`, which transfer ownership to itself.

2. `.reset()` has variants to accept deleter and allocator, which is equivalent to construct-and-swap.

3. `.use_count()` normally uses relaxed load; it's usually not reliable in multi-threading env for TOC/TOU* problem.

4. `std::get_deleter()` will return `nullptr` if the `shared_ptr` doesn't has one; otherwise return pointer to deleter.

Member functions

(constructor)	constructs new `shared_ptr` (public member function)
(destructor)	destructs the owned object if no more `shared_ptr`s link to it (public member function)
operator=	assigns the `shared_ptr` (public member function)
Modifiers	
reset	replaces the managed object (public member function)
swap	swaps the managed objects (public member function)
Observers	
get	returns the stored pointer (public member function)
operator* **operator->**	dereferences the stored pointer (public member function)
operator[] (C++17)	provides indexed access to the stored array (public member function)
use_count	returns the number of `shared_ptr` objects referring to the same managed object (public member function)
operator bool	checks if the stored pointer is not null (public member function)

(Non-member)

get_deleter	returns the deleter of specified type, if owned (function template)
operator<<(std::shared_ptr)	outputs the value of the stored pointer to an output stream (function template)
std::swap(std::shared_ptr) (C++11)	specializes the `std::swap` algorithm (function template)

shared_ptr*

```
template<typename T>
class SharedPtr
{
    BlockBase* blockPtr;
    T* objPtr;
};
```

Address of object it refers to
v.s.
Control block it refers to
(control block may store `nullptr`)

- A not-that-important note…
  - (A) `shared_ptr.get() == nullptr` (or `!operator bool`) isn't equivalent to (B) `shared_ptr.use_count() == 0` (or called "empty `shared_ptr`").
  - A && B: only these two ctors.

```
constexpr shared_ptr() noexcept;
constexpr shared_ptr(nullptr_t) noexcept : shared_ptr() { }
```

  - A & !B: other ctors that assign `nullptr`;

```
std::shared_ptr<int> ptr1(nullptr);
std::cout << "ptr1 use_count: " << ptr1.use_count() << std::endl;

std::shared_ptr<int> ptr2(nullptr, std::default_delete<int>());
std::cout << "ptr2 use_count: " << ptr2.use_count() << std::endl;

int* essentiallyNullptr = nullptr;
std::shared_ptr<int> ptr3(essentiallyNullptr);
std::cout << "ptr3 use_count: " << ptr3.use_count() << std::endl;
```

```
ptr1 use_count: 0
ptr2 use_count: 1
ptr3 use_count: 1
```

  - B & !A: aliasing ctor that accepts empty source `shared_ptr`, while aliased pointer isn't `nullptr`.

```
std::shared_ptr<int> ptr1(nullptr);
std::cout << (bool)ptr1 << " ptr1: " << ptr1.get() << std::endl;

float temp = 0.0f;
std::shared_ptr<float> ptr2(ptr1, &temp);
std::cout << (bool)ptr2 << " ptr2: " << ptr2.get() << std::endl;
```

```
false ptr1: 0
true ptr2: 0x7fffe81d125c
```

Memory Management

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

weak_ptr

- Instead of ownership, sometimes we only need to observe…
  - That is, we don’t force a resource to exist;
  - Instead, when we use it, we first try to share the ownership;
    - If the resource doesn’t exist, that’s fine; we can process it.
    - Otherwise we get its ownership and just use it.
- `weak_ptr` is just for this!
  - It has a weak reference to `shared_ptr`;
    - That is, even if there is some weak reference, resource of `shared_ptr` can be freed as long as there is no shared reference.
  - When you want to use the resource, use `.lock()` to get the `shared_ptr`;
    - It may return a null pointer, which needs special process;
    - And it may return a valid `shared_ptr`, which then holds the ownership so that you can safely use the resource.

weak_ptr

- For example:

In concurrent environment, some resources may be used by many threads; instead of loading again and again, we can prepare a cache.

When nobody uses it, cache will release the resource immediately.

```
ConcurrentMap<std::string, std::weak_ptr<Resource>> resourceCache;

std::shared_ptr<Resource> LoadResource(std::string path)
{
    auto it = resourceCache.find(path);
    if (it == resourceCache.end())
    {
        std::shared_ptr<Resource> res = LoadFromFile(path);
        resourceCache.emplace(path, res);
        return res;
    }
    // Now iterator is valid.
    auto weakPtr = it->second;
    if (auto sharedPtr = weakPtr.lock())
    {
        return sharedPtr;
    }

    std::shared_ptr<Resource> res = LoadFromFile(path);
    it->second = res;
    return res;
}
```

Converted to `shared_ptr` when used.

Constructed or assigned with `shared_ptr`.

*Note that this function may still be thread-unsafe unless iterator access is thread-safe.

weak_ptr

```
struct BlockBase
{
    std::atomic<int> sharedCnt;
    std::atomic<int> weakCnt;
    virtual void Destroy() = 0;
};
```

- Essentially, control block of `shared_ptr` also maintains a weak counter for `weak_ptr`.
  - When shared count reaches 0 (or called "expired"), object will be destroyed;
    - After that, all `.lock()` of `weak_ptr` will return null `shared_ptr`.
  - When weak count also reaches 0, the control block will be destroyed.
  - `weak_ptr` also stores pointer to the control block.

- Note 1: Weak reference arises problem for `make_shared`...
  - `make_shared` allocates all memory in a single control block;
    - And only when **weak count** reaches 0, the control block will be freed.
  - So the memory occupied by object will **persist** even if shared count reaches 0.

weak_ptr

- So in the case that:
  1. Object memory is very large;
  2. It may be referred by `weak_ptr`;
  3. You want to release memory in time;
- It's improper to use `make_shared`.

- Note 2: `enable_shared_from_this` just has a `weak_ptr` as data member.
  - Ctor of `shared_ptr` will detect and assign.

Question: why not have a `shared_ptr`?

[1] In the constructor definitions below, enables `shared_from_this` with p, for a pointer p of type `Y*`, means that if `Y` has an unambiguous and accessible base class that is a specialization of `enable_shared_from_this` ([util.smartptr.enab]), then `remove_cv_t<Y>*` shall be implicitly convertible to `T*` and the constructor evaluates the statement:

```
if (p != nullptr && p->weak-this.expired())
  p->weak-this = shared_ptr<remove_cv_t<Y>>(*this, const_cast<remove_cv_t<Y>*>(p));
```

The assignment to the *weak-this* member is not atomic and conflicts with any potentially concurrent access to the same object ([intro.multithread]).

weak_ptr

- Note 3: `weak_ptr` can also be used to break cyclic reference of `shared_ptr`.
  - For example, when you implement a list:
    - Normally we will allocate an empty node and concatenate two ends.
    - The list just holds a `shared_ptr` to the empty node.

```
struct BadListNode
{
    std::shared_ptr<BadListNode> prev;
    std::shared_ptr<BadListNode> next;
};
```

  - However, when the list is destructed, nodes are not released…
    - Each node is referred by its previous and next nodes, so all `.use_count()` are 2.
    - This generates unreachable memory, causing memory leak…
  - In real scenarios, cyclic reference will be more subtle and hard to detect.
  - And if you **really** need such cyclic reference, you should insert `weak_ptr`.

```
class Node
{
    char id;
    std::variant<std::weak_ptr<Node>, std::shared_ptr<Node>> ptr;
public:
    Node(char id) : id{id} {}
    ~Node() { std::cout << "  '" << id << "' reclaimed\n"; }
    /*...*/
    void assign(std::weak_ptr<Node> p) { ptr = p; }
    void assign(std::shared_ptr<Node> p) { ptr = p; }
};
```

weak_ptr

- Note 4: finally list APIs.

1. Constructed by either `shared_ptr<Y>` or `weak_ptr<Y>`, where `Y*` is implicitly convertible to `T*`.

2. No `.get()` because it's only safe to access by converting to `shared_ptr` and `.get()`.

3. `.use_count()` is still shared count.

4. Though ctor of `shared_ptr` can accept `weak_ptr`, it will `throw bad_weak_ptr` if `weak_ptr` is expired. So you should always use `.lock()`.

Effectively returns `expired() ? shared_ptr<T>() : shared_ptr<T>(*this)`, executed atomically.

Member types

Member type	Definition
element_type	T (until C++17)
	std::remove_extent_t<T> (since C++17)

Member functions

(constructor)	creates a new weak_ptr (public member function)
(destructor)	destroys a weak_ptr (public member function)
operator=	assigns the weak_ptr (public member function)
Modifiers	
reset	releases the ownership of the managed object (public member function)
swap	swaps the managed objects (public member function)
Observers	
use_count	returns the number of shared_ptr objects that manage the object (public member function)
expired	checks whether the referenced object was already deleted (public member function)
lock	creates a shared_ptr that manages the referenced object (public member function)
owner_before	provides owner-based ordering of weak pointers (public member function)
owner_hash (C++26)	provides owner-based hashing of weak pointers (public member function)
owner_equal (C++26)	provides owner-based equal comparison of weak pointers (public member function)

Non-member functions

std::swap(std::weak_ptr) (C++11)	specializes the std::swap algorithm (function template)

atomic smart pointer

- Finally, concurrent access of `shared_ptr` itself causes data races.
  - For example:

```
std::shared_ptr<int> ptr = std::make_shared<int>(4);
std::thread threadA([&ptr]{
   ptr = std::make_shared<int>(10);
});
std::thread threadB([&ptr]{
   ptr = std::make_shared<int>(20);
});
```

  - Reason: `shared_ptr` holds pointer to atomic counters (i.e. control block with two atomic counters); but the pointer itself is non-atomic.
- For raw pointer, we have specialization `std::atomic<T*>`;
  - Since C++20, `std::atomic<shared_ptr/weak_ptr>` is added.

atomic smart pointer

- Its usage are also quite similar to `std::atomic<T*>`.
  - But no specialized methods like `.fetch_add()`/`operator+=`.

```
std::atomic<std::shared_ptr<int>> ptr = std::make_shared<int>(4);
std::thread threadA([&ptr]{
    ptr.store(std::make_shared<int>(10), std::memory_order_relaxed);
});
std::thread threadB([&ptr]{
    ptr.store(std::make_shared<int>(20), std::memory_order_relaxed);
});
```

- Note 1: this doesn't mean that the underlying object is atomically accessed.
  - Just like `std::atomic<T*>` doesn't mean pointer can access `T` in parallel.
  - Instead, you should use either `shared_ptr<atomic<T>>`, or by `atomic_ref`.
    - Just like `std::atomic<T>*`.

atomic smart pointer

- For example:

```
std::atomic<std::shared_ptr<int>> ptr = std::make_shared<int>(4);
std::thread threadA([&ptr]{
    std::shared_ptr<int> p0 = ptr; // equiv. to ptr.load(seq_cst)
    (*p0)++; // Data races for two threads
});

std::shared_ptr<std::atomic<int>> ptr = std::make_shared<std::atomic<int>>(4);
std::thread threadA([&ptr]{
    auto p0 = ptr; // This copy doesn't incur data races
    (*p0)++; // Safe
});

std::shared_ptr<int> ptr = std::make_shared<int>(4);
std::thread threadA([&ptr]{
    std::atomic_ref<int> ref{ *ptr };
    ref++; // Safe
});
```

atomic smart pointer

- Note 2: though `atomic<shared_ptr>` doesn't exist before C++20, you can use global methods for `shared_ptr` (no `weak_ptr`).
  - For example:

```
template< class T >
std::shared_ptr<T> atomic_load( const std::shared_ptr<T>* p );                    (2)   (since C++11)
                                                                                        (deprecated in C++20)
                                                                                        (removed in C++26)

template< class T >
std::shared_ptr<T> atomic_load_explicit                                           (3)   (since C++11)
    ( const std::shared_ptr<T>* p, std::memory_order mo );                              (deprecated in C++20)
                                                                                        (removed in C++26)
```

  - However, as it's hard for functions to store states, these methods are likely to have worse performance than specialized class.
    - For example, `std::atomic<shared_ptr>` can store a `std::atomic_flag` as data member to implement, but functions are hard to do so.
  - Thus removed in C++26.

Memory Management

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

Smart pointer adaptors

- In C APIs, sometimes we need to accept `T**`, meaning to set a pointer.
  - It'll be slightly clumsy to manage by smart pointers:

```
auto close_db = [](sqlite3* db) { sqlite3_close(db); };
sqlite3* dbPtr;
sqlite3_open(":memory:", &dbPtr);
std::unique_ptr<sqlite3, decltype(close_db)> up{ dbPtr };
```

- C++23 adds small utilities to make it easier:

```
auto close_db = [](sqlite3* db) { sqlite3_close(db); };
std::unique_ptr<sqlite3, decltype(close_db)> up;
sqlite3_open(":memory:", std::out_ptr(up));
```

  - Essentially, this function creates a temporary `std::out_ptr_t<SmartPtr, RawPointer, Args…>`.

Smart pointer adap

- `std::out_ptr_t` is basically like:

1. it also has `operator void**`, which just `static_cast` from `RawPtr*`.

2. Actually it's `.reset(static_cast<SP>(rptr), …)` as below:

- SP be
  - `Smart::pointer`, if it is valid and denotes a type, otherwise,
  - `Smart::element_type*`, if `Smart::element_type` is valid and denotes a type, otherwise,
  - `std::pointer_traits<Smart>::element_type*`, if `std::pointer_traits<Smart>::element_type` is valid and denotes a type, otherwise,
  - `Pointer`.

```
template<typename SmartPtr, typename RawPtr, typename... Args>
class OutPtrT
{
    SmartPtr& sptr;
    RawPtr rptr;
    std::tuple<Args...> args;

public:
    // For all code below, if sptr.reset doesn't exist,
    // it will use 'sptr = SmartPtr(...)'.
    OutPtrT(SmartPtr& init_sptr, Args... args) : sptr{ init_sptr },
        rptr{}, args{ std::forward<Args>(args) }
    {
        sptr.reset();
    }

    operator RawPtr*() const noexcept
    {
        return const_cast<RawPtr*>(&rptr);
    }

    ~OutPtrT()
    {
        if (rptr)
        {
            std::apply([&](auto&&... args){
                sptr.reset(rptr, std::forward<decltype(args)>(args)...);
            }, std::move(args));
        }
    }
};
```

Smart pointer adaptors

- The additional arguments `args` are useful for `std::shared_ptr`.
  - For example:

```
auto close_db = [](sqlite3* db) { sqlite3_close(db); };
std::shared_ptr<sqlite3> sp;
sqlite3_open(":memory:", std::out_ptr(sp, close_db));
```

  - As it's normally used with foreign framework…
  - When calling `.reset()`, you almost always need to pass deleter.
    - So `std::out_ptr_t` will make compile fail for `shared_ptr` when `sizeof…(Args) == 0`.

- Besides, there is another adaptor `std::inout_ptr_t`.
  - This is used for functions that first release the resource, and then do re-initialization (quite like `freopen`).

Smart pointer ada

- Essentially like:

1. As the foreign function will release automatically, we shouldn't call `.reset()`. (Thus impossible to use `std::shared_ptr`)

2. They also have support for raw pointer; `std::out_ptr(rawPtr)` and `std::inout_ptr(rawPtr)` is basically equiv. to `&rawPtr` or `(void**)&rawPtr`. (Do pay attention to possible leak)

Note: these adaptors should always be used by passing `std::inout_ptr(…)` or `std::out_ptr(…)` into function directly. Otherwise it's easy to lead to e.g. dangling reference.

```
template<typename SmartPtr, typename RawPtr, typename... Args>
class InOutPtrT
{
    SmartPtr& sptr;
    RawPtr rptr;
    std::tuple<Args...> args;

public:
    // For all code below, if sptr.reset doesn't exist,
    // it will use 'sptr = SmartPtr(...)'.
    InOutPtrT(SmartPtr& init_sptr, Args... args) : sptr{ init_sptr },
        rptr{ init_sptr.get() }, args{ std::forward<Args>(args) }
    { // This release is allowed to move to dtor.
      sptr.release();
    }

    operator RawPtr*() const noexcept
    {
        return const_cast<RawPtr*>(&rptr);
    }

    ~InOutPtrT()
    {
        if (rptr)
        {
            std::apply([&](auto&&... args){
                sptr.reset(rptr, std::forward<decltype(args)>(args)...);
            }, std::move(args));
        }
    }
};
```

Memory Management

Allocator

Memory Management

- Allocator
  - Basics
  - PMR

Allocator

- In standard library, most of types that need dynamic memory allocation will use allocator.
  - Users can specify different allocators to control the behavior.
- Formally, allocator is a general concept that encapsulates strategies for allocation/deallocation.
  - The standard library will extract properties of allocator by `std::allocator_traits<Alloc>`.
  - Theoretically, allocator needs to regulate lots of alias for generalization;
    - For example, what is pointer type for `T`?
  - But `std::allocator_traits` provides many default values that are enough to use in most cases.
    - Thus we will omit trivial parts.

Allocator

- For example:

The default, non-specialized, `std::allocator_traits` contains the following members:

Member types

Type	Definition
`allocator_type`	`Alloc`
`value_type`	`Alloc::value_type`
`pointer`	`Alloc::pointer` if present, otherwise `value_type*`
`const_pointer`	`Alloc::const_pointer` if present, otherwise `std::pointer_traits<pointer>::rebind<const value_type>`
`void_pointer`	`Alloc::void_pointer` if present, otherwise `std::pointer_traits<pointer>::rebind<void>`
`const_void_pointer`	`Alloc::const_void_pointer` if present, otherwise `std::pointer_traits<pointer>::rebind<const void>`
`difference_type`	`Alloc::difference_type` if present, otherwise `std::pointer_traits<pointer>::difference_type`
`size_type`	`Alloc::size_type` if present, otherwise `std::make_unsigned<difference_type>::type`

The only one that needs to be defined.

Default value is usually enough (e.g. `T*`, `const T*`, …)

Allocator

- To support a minimal allocator `Alloc<T>`, you need:
  1. Nested type `value_type`, e.g. by alias `using value_type = T;`
  2. Member method `allocate(elemNum)` and `deallocate(ptr, elemNum)`;
  3. “Equality-comparable” and “Copyable”.
- For example:

```
template<class T>
struct Mallocator
{
1.  using value_type = T;

    Mallocator() = default;

    template<class U>
3.  constexpr Mallocator(const Mallocator <U>&) noexcept {}
    // != is automatically generated since C++20.
    template<typename U>
3.  bool operator==(const Mallocator<U>&) const{ return true; }
```

```
[[nodiscard]] T* allocate(std::size_t n)      2.
{
    if (n > std::numeric_limits<std::size_t>::max() / sizeof(T))
        throw std::bad_array_new_length();

    if (auto p = static_cast<T*>(std::malloc(n * sizeof(T))))
    {
        report(p, n);
        return p;
    }

    throw std::bad_alloc();
}

void deallocate(T* p, std::size_t n) noexcept 2.
{
    report(p, n, 0);
    std::free(p);
}
```

Allocator

- Then we can use it with `std::vector`!

```
std::vector<int, Mallocator<int>> v(8);
v.push_back(42);
```

  - We can also easily write an “allocator-aware” container:

```
template<typename T, typename Alloc>
class NaiveVector
{
    Alloc alloc_;
    T* begin_ = nullptr, *end_ = nullptr;

    // Rigorously, we should use rebind; but omit here.
    using AllocTraits = std::allocator_traits<Alloc>;

public:
    NaiveVector(const Alloc& alloc = Alloc{}) : alloc_{alloc} {}
```

```
void Resize(std::size_t n, const T& initVal)
{
    begin_ = AllocTraits::allocate(alloc_, n);
    end_ = begin_ + n;
    for (std::size_t i = 0; i < n; i++)
        new(begin_ + i) T{ initVal };
}
~NaiveVector()
{
    for (auto ptr = begin_; ptr != end_; ptr++)
        ptr->~T();

    AllocTraits::deallocate(alloc_, begin_, end_ - begin_);
}
```

Note that a correct version should deallocate previous memory.

Allocator

- We can apply allocator for `std::list` similarly:

```
std::list<int, Mallocator<int>> v;
v.push_back(42);
```

- Well, not that similar…
  - Does `std::list` really allocate a single `int`?
  - No, it allocates a `Node<int>`, which cannot be handled by `Alloc<int>`.
    - Instead, it should be handled by `Alloc<Node<int>>`.
- So, allocator supports “rebind”, meaning to transform `Alloc<T>` to `Alloc<U>`.

Member alias templates

Type	Definition
`rebind_alloc<T>`	`Alloc::rebind<T>::other` if present, otherwise `SomeAllocator<T, Args>` if this `Alloc` is of the form `SomeAllocator<U, Args>`, where `Args` is zero or more type arguments
`rebind_traits<T>`	`std::allocator_traits<rebind_alloc<T>>`

Allocator

NOTICE that it's library UB to use `Container<T, Alloc<U>>` before C++20, and compilation error since C++20. Here we enhance it.

`Allocator` - An allocator that is used to acquire/release memory and to construct/destroy the elements in that memory. The type must meet the requirements of *Allocator*. The behavior is undefined(until C++20) The program is ill-formed(since C++20) if `Allocator::value_type` is not the same as T.

- If we apply it in our vector implementation:

```
template<typename T, typename Alloc>
class NaiveVector
{
    // To prevent users from passing Allocator<U> instead of <T>.
    using RealAlloc = std::allocator_traits<Alloc>::template rebind_alloc<T>;
    using AllocTraits = std::allocator_traits<RealAlloc>;

    RealAlloc alloc_;
    T* begin_ = nullptr, *end_ = nullptr;

public:
    NaiveVector(const Alloc& alloc = Alloc{}) : alloc_{alloc} {}
```

- Thus "copyable" means ability to convert `Alloc<U>` to `Alloc<T>`.

```
template<class U>
constexpr Mallocator(const Mallocator <U>&) noexcept {}
```

Allocator

- Similarly, for our previous homework `List` implementation:

```
template<typename T>
class ListBase
{
protected:
    ListNode<T> sentinel_{ &sentinel_, &sentinel_ };
    ~ListBase()
    {
        for (auto it = sentinel_.next; it != &sentinel_;)
        {
            it = it->next;
            delete it->prev;
        }
```

Rewrite it with allocator…

```
template<typename T>
class List : public ListBase<T>
{
    template<typename It>
    List(It begin, It end)
    {
        auto pos = &this->sentinel_;
        while (begin != end)
        {
            std::unique_ptr<ListNode<T>> ptr{ new ListNode<T>{ pos, pos->next,
                                                               *begin } };
            ++begin;

            pos->next->prev = ptr.get();
            pos->next = ptr.release();
            pos = pos->next;
        }
        return;
    }
```

Allocator

```cpp
int arr[5]{ 1,2,3,4,5 };
List<int, Mallocator<int>> l{ arr, arr + 5 };
for (auto it = l.begin(); it != l.end(); it++)
    std::cout << *it << " ";
std::cout << "\n";
```

```
Alloc: 24 bytes at 00000217055A83F0
Alloc: 24 bytes at 00000217055A8150
Alloc: 24 bytes at 00000217055A7B50
Alloc: 24 bytes at 00000217055A7670
Alloc: 24 bytes at 00000217055A7850
1 2 3 4 5
Dealloc: 24 bytes at 00000217055A83F0
Dealloc: 24 bytes at 00000217055A8150
Dealloc: 24 bytes at 00000217055A7B50
Dealloc: 24 bytes at 00000217055A7670
Dealloc: 24 bytes at 00000217055A7850
```

```cpp
template<typename T, typename Alloc>
class ListBase
{
protected:
    using NodeType = ListNode<T>;
    using RealAlloc = std::allocator_traits<Alloc>::template
        rebind_alloc<NodeType>;
    using AllocTraits = std::allocator_traits<RealAlloc>;

    RealAlloc alloc_;
    ListNode<T> sentinel_{ &sentinel_, &sentinel_ };
    ListBase(const Alloc& alloc) : alloc_{ alloc } {}
    ~ListBase()
    {
        for (auto it = sentinel_.next; it != &sentinel_;)
        {
            it = it->next;
            it->~NodeType();
            AllocTraits::deallocate(alloc_, it->prev, 1);
        }
    }
};
```

```cpp
template<typename T, typename Alloc>
class List : public ListBase<T, Alloc>
{
    using Super_ = ListBase<T, Alloc>;
    using AllocTraits = Super_::AllocTraits;
    template<typename It>
    List(It begin, It end, const Alloc& alloc_ = {}) : Super_{ alloc_ }
    {
        auto pos = &this->sentinel_;
        while (begin != end)
        {
            ListNode<T>* newNode = AllocTraits::allocate(this->alloc_, 1);
            new(newNode) ListNode<T>{ pos, pos->next, *begin };
            ++begin;

            pos->next->prev = newNode;
            pos->next = newNode;
            pos = pos->next;
        }
        return;
    }
```

Allocator-defined con/destruction

- Besides, allocator can also control construction & destruction.
  - For example, allocators may pass additional parameters, so it's not enough to just placement new directly…

`a.construct(xp, args...)` (optional)	*(not used)*	Constructs an object of type X in previously-allocated storage at the address pointed to by `xp`, using `args...` as the constructor arguments.
`a.destroy(xp)` (optional)	*(not used)*	Destructs an object of type X pointed to by `xp`, but does not deallocate any storage.

  - And similarly, `allocator_traits` will provide default version in its static methods:

construct [static]	constructs an object in the allocated storage (function template)
destroy [static]	destructs an object stored in the allocated storage (function template)

    - For `construct`, placement new by default;
    - For `destroy`, dtor by default.

Allocator

- So essentially we should manage them by allocator in code:

```
ListNode<T>* newNode = AllocTraits::allocate(this->alloc_, 1);
AllocTraits::construct(this->alloc_, newNode, pos, pos->next, *begin);
```

```
AllocTraits::destroy(this->alloc_, it);
AllocTraits::deallocate(alloc_, it->prev, 1);
```

- However, there seems no guarantee for exception safety…
  - When construction throws exception, then allocation needs to be reverted.
  - So we may write a simple RAII wrapper ourselves:

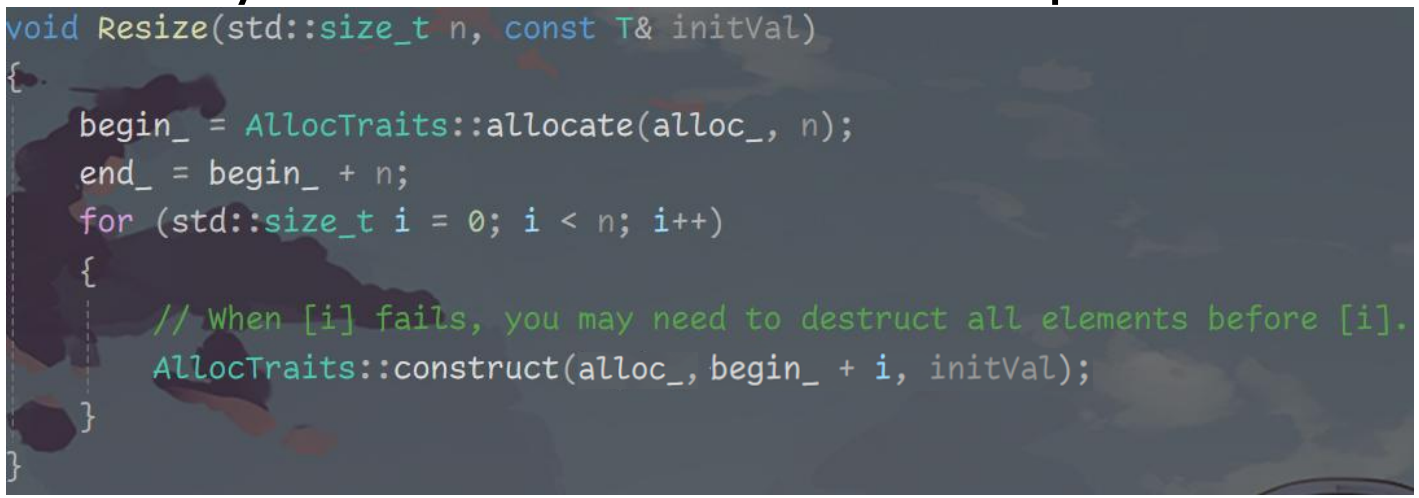
```
template<typename T>
class AllocGuard
{
    using AllocTraits = std::allocator_traits<T>;
    using PointerType = AllocTraits::pointer;

    T& alloc_;
    PointerType ptr_;
    std::size_t size_;
```

```
public:
    AllocGuard(T& alloc, std::size_t n = 1) : alloc_{ alloc },
        ptr_{ AllocTraits::allocate(alloc, n) }, size_{ n }
    {
    }

    auto Get() const noexcept { return ptr_; }
    auto Release() noexcept { return std::exchange(ptr_, PointerType{}); }

    ~AllocGuard()
    {
        if (ptr_)
            AllocTraits::deallocate(alloc_, ptr_, size_);
    }
};
```

```
auto newNode = AllocGuard{ this->alloc_ };
AllocTraits::construct(this->alloc_, newNode.Get(), pos,
                       pos->next, *begin);
```

Allocator

- More generally, we may even need to revert all previous constructions:



```cpp
void Resize(std::size_t n, const T& initVal)
{
    begin_ = AllocTraits::allocate(alloc_, n);
    end_ = begin_ + n;
    for (std::size_t i = 0; i < n; i++)
    {
        // When [i] fails, you may need to destruct all elements before [i].
        AllocTraits::construct(alloc_, begin_ + i, initVal);
    }
}
```

- C++ provides *uninitialized memory algorithms* (defined in <memory>), but unfortunately they are **allocator-unaware**.
  - i.e. construction just uses placement new, and destroy uses dtor.

Allocator

Defined in header <memory>

uninitialized_copy	copies a range of objects to an uninitialized area of memory (function template)
uninitialized_copy_n (C++11)	copies a number of objects to an uninitialized area of memory (function template)
uninitialized_fill	copies an object to an uninitialized area of memory, defined by a range (function template)
uninitialized_fill_n	copies an object to an uninitialized area of memory, defined by a start and a count (function template)
uninitialized_move (C++17)	moves a range of objects to an uninitialized area of memory (function template)
uninitialized_move_n (C++17)	moves a number of objects to an uninitialized area of memory (function template)
uninitialized_default_construct (C++17)	constructs objects by default-initialization in an uninitialized area of memory, defined by a range (function template)
uninitialized_default_construct_n (C++17)	constructs objects by default-initialization in an uninitialized area of memory, defined by a start and a count (function template)
uninitialized_value_construct (C++17)	constructs objects by value-initialization in an uninitialized area of memory, defined by a range (function template)
uninitialized_value_construct_n (C++17)	constructs objects by value-initialization in an uninitialized area of memory, defined by a start and a count (function template)
destroy (C++17)	destroys a range of objects (function template)
destroy_n (C++17)	destroys a number of objects in a range (function template)

Ranges-version is also provided since C++20.

Allocator

std::uninitialized_copy

Defined in header <memory>

```
template< class InputIt, class NoThrowForwardIt >
NoThrowForwardIt uninitialized_copy( InputIt first, InputIt last,          (1)   (constexpr since C++26)
                                     NoThrowForwardIt d_first );

template< class ExecutionPolicy, class ForwardIt,
          class NoThrowForwardIt >
NoThrowForwardIt uninitialized_copy( ExecutionPolicy&& policy,             (2)   (since C++17)
                                     ForwardIt first, ForwardIt last,
                                     NoThrowForwardIt d_first );
```

1) Copies elements from the range [`first` , `last`) to an uninitialized memory area beginning at `d_first` as if by

```
for (; first != last; ++d_first, (void) ++first)
    ::new (voidify(*d_first))
        typename std::iterator_traits<NoThrowForwardIt>::value_type(*first);
```

If an exception is thrown during the initialization, the objects already constructed are destroyed in an unspecified order.

2) Same as (1), but executed according to `policy` .

This overload participates in overload resolution only if all following conditions are satisfied:

`std::is_execution_policy_v<std::decay_t<ExecutionPolicy>>` is `true` .	(until C++20)
`std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>` is `true` .	(since C++20)

If `d_first + [0, std::distance(first, last))` overlaps with [`first` , `last`), the behavior is undefined. (since C++20)

- As if:

```cpp
template<class InputIt, class NoThrowForwardIt>
constexpr NoThrowForwardIt uninitialized_copy(InputIt first, InputIt last,
                                               NoThrowForwardIt d_first)
{
    using T = typename std::iterator_traits<NoThrowForwardIt>::value_type;
    NoThrowForwardIt current = d_first;
    try
    {
        for (; first != last; ++first, (void) ++current)
            ::new (static_cast<void*>(std::addressof(*current))) T(*first);
        return current;
    }
    catch (...)
    {
        for (; d_first != current; ++d_first)
            d_first->~T();
        throw;
    }
}
```

Otherwise it may throw in catch block, and elements afterwards won't be destructed.

- No increment, assignment, comparison, or indirection through valid instances of `NoThrowForwardIt` may throw exceptions. Applying `&*` to a `NoThrowForwardIt` value must yield a pointer to its value type. (until C++11)

- For example:

```cpp
void Resize(std::size_t n, const T& initVal)
{
    AllocGuard guard{ alloc_, n };
    auto begin = guard.Get(), end = begin + n;
    std::uninitialized_fill(begin, end, initVal);

    begin_ = guard.Release(), end_ = end;
}
```

Allocator-aware version will be left as our homework.

Allocator propagation

- Finally, what happens during container assignment?
  - For example, previously we've stated what `std::vector` does normally during move assignment:
    - Release the original memory and exchange three pointers.
    - Where is allocator?
  - So in essence, allocators define `propagate_on_container_move_assignment` (POCMA) to determine what should be done.

So normally POCMA is alias to `true_type` or `false_type`.

1. When `POCMA::value` is `true`, allocator will be copied to the assigned container;
   - The assigned container just releases memory by the original allocator, gets the new allocator, and then exchange pointers.
2. When `POCMA::value` is `false`, the assigned container will keep its own allocator.
   - Then allocators will be compared by equality;
   - **Equal allocators mean that their memory are inter-operatable**, i.e. it's correct to use allocator A to release memory allocated by allocator B.

Allocator propagation

- So again two sub-cases for case 2:
  ① When two containers have equal allocators, then just same as normal ones.
    - i.e. Release the original memory and exchange three pointers.
    - No allocator copy.
  ② When two containers have unequal allocators, we cannot exchange memory pointer directly.
    - So every element is move-assigned individually.
    - If memory isn't enough, the original allocator should allocate more.
- And finally, allocator can define `is_always_equal` for optimization.
  - Then allocator comparison can be done during compilation time.
  - Normally only *stateless allocator* defines it, i.e. all allocators manage the same memory (e.g. `std::allocator` only manages global heap).

Allocator propagation

- Similarly, we can define `propagate_on_container_copy_assignment` (POCCA) and `propagate_on_container_swap` (POCS) to determine behavior of copy assignment and swap.
  - In most cases, POCMA, POCCA and POCS are of same value.
  - Particularly, when POCS is `false` and allocators are unequal, direct swap is not well-defined (and standard containers mark it as UB).
- And we can extract these properties by `allocator_traits` too:

`propagate_on_container_copy_assignment`	`Alloc::propagate_on_container_copy_assignment` if present, otherwise `std::false_type`
`propagate_on_container_move_assignment`	`Alloc::propagate_on_container_move_assignment` if present, otherwise `std::false_type`
`propagate_on_container_swap`	`Alloc::propagate_on_container_swap` if present, otherwise `std::false_type`
`is_always_equal`	`Alloc::is_always_equal` if present, otherwise `std::is_empty<Alloc>::type` When allocator class is empty class, it doesn't store any state and is regarded as always equal.

Allocator propagation

- For example:

```
vector& operator=(const vector& another)
{
    if (this == &another)
        return false;

    if constexpr (AllocTraits::propagate_on_container_copy_assignment::value)
    {
        if constexpr (!AllocTraits::is_always_equal::value)
        {
            if (alloc_ != another.alloc_)
                Destroy_(); // Reallocate for the next assign.
        }
        alloc_ = another.alloc_;
    }

    this->assign(another.begin(), another.end());
    return *this;
}
```

```
vector& operator=(vector&& another)
{
    if (this == &another)
        return false;

    if constexpr (AllocTraits::propagate_on_container_move_assignment::value)
    {
        Destroy_(); // Release its own memory
        ExchangePointers_(another);
        alloc_ = another.alloc_;
        return *this;
    }
    // Else do NOT propagate allocator.
    if constexpr (!AllocTraits::is_always_equal::value)
    {
        if (alloc_ != another.alloc_)
        {
            // Move-assign elements one by one.
            this->assign(std::move_iterator{ another.begin() },
                         std::move_iterator{ another.end() });
            return *this;
        }
    }
    // Else, two allocators are equal
    Destroy_(); // Release its own memory
    ExchangePointers_(another);
    return *this;
}
```

Allocator propagation

- And finally, allocator will be informed when container is copy-constructed.
  - Just define `select_on_container_copy_construction` (SOCCC), which should return new allocator from current allocator.
  - And similarly, you can call it by allocator traits:
    - By default just return copy of parameter.

select_on_container_copy_construction [static] obtains the allocator to use after copying a standard container
(public static member function)

std::allocator_traits<Alloc>::select_on_container_copy_construction

Defined in header <memory>

```
static Alloc select_on_container_copy_construction( const Alloc& a );
```

(since C++11)
(constexpr since C++20)

- For example:

```
vector(const vector& another) : alloc_{
    AllocTraits::select_on_container_copy_construction(another.alloc_) }
{ /* ... */ }
```

Allocator propagation

- Particularly, this doesn’t apply for move ctor of container.
  - Move ctor just moves allocator and exchanges pointers.

```
_CONSTEXPR20 vector(const vector& _Right)
    : _Mypair(_One_then_variadic_args_t{}, _Alty_traits::select_on_container_copy_construction(_Right._Getal())) {
```

```
_CONSTEXPR20 vector(vector&& _Right) noexcept
    : _Mypair(_One_then_variadic_args_t{}, _STD move(_Right._Getal()),
          _STD exchange(_Right._Mypair._Myval2._Myfirst, nullptr),
          _STD exchange(_Right._Mypair._Myval2._Mylast, nullptr),
          _STD exchange(_Right._Mypair._Myval2._Myend, nullptr)) {
```

  - And all ctors (including copy ctor and move ctor) have allocator-aware version, which will then copy the allocator and allocate by that.

```
_CONSTEXPR20 vector(const vector& _Right, const _Identity_t<_Alloc>& _Al)
    : _Mypair(_One_then_variadic_args_t{}, _Al) {
```

Guess how move ctor with allocator is implemented.

All gati

```
template<typename T, typename Allocator>
class indirect
{
    T* ptr_ = nullptr;
    Allocator alloc_;
};
```

```
void AllocAndCopyConstruct_(const indirect& other)
{
    ptr_ = AllocTraits::allocate(alloc_, 1);
    AllocTraits::construct(alloc_, ptr_, *other.ptr_);
}
```

- Besides container, let's use `std::indirect` as another example to see how we may treat propagation behaviors.
  - Copy ctor: `self` allocator is either initialized with SOCCC of `other`'s allocator, or initialized with explicitly provided parameter `a`.
    - The owned object is allocated and copy constructed from object of `other`.

```
constexpr indirect( const indirect& other );
```

```
constexpr indirect( std::allocator_arg_t, const Allocator& a,
                    const indirect& other );
```

```
indirect(const indirect& other) : alloc_{ AllocTraits::SOCCC(other.alloc_) }
{
    if (other.ptr_ != nullptr)
        AllocAndCopyConstruct_(other);
}

indirect(std::allocator_arg_t, const Allocator& a, const indirect& other)
    : alloc_{ a }
{
    if (other.ptr_ != nullptr)
        AllocAndCopyConstruct_(other);
}
```

Pseudo code, no e.g. exception safety.

Allocator prop

```
constexpr indirect( indirect&& other ) noexcept;
constexpr indirect( std::allocator_arg_t, const Allocator& a,
                    indirect&& other ) noexcept(/* see below */);
```

- Move ctor: `self` allocator is either moved from `other`'s allocator, or initialized with explicitly provided parameter `a`.

`other.valueless_after_move()` is `true` after move.

1. When moved from `other`'s allocator, just take its pointer;
2. Otherwise if its allocator is equal to `other`'s allocator, just take its pointer;
3. Otherwise, the owned object is allocated and move constructed from object of `other`.

```
indirect(indirect&& other) : alloc_{ std::move(other.alloc_) },
    ptr_{ std::exchange(other.ptr_, nullptr) } { }

indirect(std::allocator_arg_t, const Allocator& a, indirect&& other)
    : alloc_{ a }, ptr_{ nullptr }
{
    if (other.ptr_ == nullptr)
        return;
    if (alloc_ == other.alloc_) // Optimizable with is_alway_equal.
    {
        ptr_ = std::exchange(other.ptr_, nullptr);
        return;
    }
    ptr_ = AllocTraits::allocate(alloc_, 1);
    AllocTraits::construct(alloc_, ptr_, std::move(*other.ptr_));
}
```

Allocator pro

```
void Destroy_()
{
    if (ptr_ != nullptr)
    {
        AllocTraits::destroy(alloc_, ptr_);
        AllocTraits::deallocate(alloc_, ptr_, 1);
        ptr_ = nullptr;
    }
}
```

```
void TryUpdateAlloc_(const indirect& other)
{
    if constexpr(AllocTraits::POCCA)
        alloc_ = other.alloc_;
}
```

- Copy assignment: will finally copy allocator when POCCA.
  1. If `self` allocator is equal to `other`'s allocator and `self` is not valueless, copy-assign the underlying object.

```
indirect& operator=(const indirect& other)
{
    if (this == &other)
        return *this;
```

```
if (alloc_ == other.alloc_ && ptr_ != nullptr)
{
    *ptr_ = *other.ptr_;
    TryUpdateAlloc_(other);
    return *this;
}
```

  2. Otherwise, two allocators are not equal, `std::indirect` chooses to "copy-and-swap" to process all cases uniformly.

```
const Allocator& newAlloc = AllocTraits::POCCA ? other.alloc_ : alloc_;
std::indirect temp{ std::allocator_arg_t, newAlloc, other };
Destroy_();
TryUpdateAlloc_();
ptr_ = std::exchange(temp.ptr_, nullptr);
return *this;
```

Allocator propagation

- Strictly speaking, to prevent unnecessary operations (e.g. `temp` needs to copy allocator), copy-and-swap is done manually.

1) If `std::addressof(other) == this` is `true`, does nothing. Otherwise, let `need_update` be `traits::propagate_on_container_copy_assignment::value`:
   - If `other` is valueless, `*this` becomes valueless and the object owned by `*this` (if any) is destroyed using `traits::destroy` and then the storage is deallocated.
   - Otherwise, if *`alloc`* `== other.`*`alloc`* is `true` and `*this` is not valueless, equivalent to `**this = *other`.
   - Otherwise:
     1. Constructs a new owned object in `*this` using `traits::construct` with `*other` as the argument, using the allocator `update_alloc ? other.`*`alloc`* `: `*`alloc`*.
     2. The previously owned object in `*this` (if any) is destroyed using `traits::destroy` and then the storage is deallocated.
     3. *p* points to the new owned object.

   After updating the object owned by `*this`, if `need_update` is `true`, *`alloc`* is replaced with a copy of `other.`*`alloc`*.

Allocator propagation

```
if (alloc_ == other.alloc_)
{
    std::swap(ptr_, other_.ptr_);
    other_.Destroy_();
    // Uses POCMA instead of POCCA.
    TryUpdateAlloc_(other);
    return *this;
}
```

- Move assignment: basically equivalent to copy assignment, except that case 1 doesn't move-assign, but just steal-and-destroy.

2) If `std::addressof(other) == this` is `true`, does nothing. Otherwise, let `need_update` be `traits::propagate_on_container_move_assignment::value`:

- If `other` is valueless, `*this` becomes valueless and the object owned by `*this` (if any) is destroyed using `traits::destroy` and then the storage is deallocated.
- Otherwise, if *`alloc`* `== other.`*`alloc`* is `true`, swaps the owned objects in `*this` and `other`; the owned object in `other` (if any) is then destroyed using `traits::destroy` and then the storage is deallocated.
- Otherwise:
    1. Constructs a new owned object in `*this` using `traits::construct` with `std::move(*other)` as the argument, using the allocator `update_alloc ? other.`*`alloc`*` : `*`alloc`*.
    2. The previously owned object in `*this` (if any) is destroyed using `traits::destroy` and then the storage is deallocated.
    3. *p* points to the new owned object.

After updating the objects owned by `*this` and `other`, if `need_update` is `true`, *`alloc`* is replaced with a copy of `other.`*`alloc`*.

Allocator propagation

- Swap: similarly, UB when POCS is `false` while allocators are unequal.

```
constexpr void swap( indirect& other ) noexcept(/* see below */);     (since C++26)
```

Swaps the contents with those of `other`.

In the description below, `swap_allocators` refers to `std::allocator_traits<Allocator>::propagate_on_container_swap::value`.

Swaps the states of `*this` and `other`, exchanging owned objects or valueless states.

- If `swap_allocators` is `true`, then executes
```
using std::swap;
swap(alloc, other.alloc);
```
.
- Otherwise, the allocators are not swapped.

If one of the following conditions is satisfied, the behavior is undefined:

- `swap_allocators` is `true`, and `Allocator` does not satisfy the requirements of *Swappable*.
- `swap_allocators` is `false`, and `get_allocator() == other.get_allocator()` is `false`.

- To conclude: though allocators have some regulations (e.g. by POCMA), the specific propagation behaviors are determined by the class itself.
  - E.g. move-assignment or steal-and-destroy, it's up to your design!

Allocator

- Note 1: C++23 adds an optional interface `allocate_at_least` for allocator.

std::allocator_traits<Alloc>::allocate_at_least

```
static constexpr std::allocation_result<pointer, size_type>
    allocate_at_least( Alloc& a, size_type n );
```
(since C++23)

  - It should allocate space not less than `n` elements.
  - This is useful for some allocation that tolerates more space; a typical example is vector reallocation.
    - For example, our allocator can only allocate 16-byte chunks; but reallocation strategy wants three `int` (assuming 4-byte).
    - Then we can return 16-byte chunk and such reallocation leads to four `int`.

```
template< class Pointer, class SizeType = std::size_t >
struct allocation_result;
```

Member name
ptr
count

```
void Reallocation()
{
    auto expectedSize = GetNextCapacity(end_ - begin_);
    auto [newPtr, newSize] = AllocTraits::allocate_at_least(
        alloc_, expectedSize);
    // move_if_noexcept original elements, and destroy original
    // elements and resource.
    begin_ = newPtr, end_ = newPtr + newSize;
    return;
}
```

Allocator

The history is a little bit complex… `void*` is seemingly hard to handle in compilers' constexpr evaluator and thus the template method `construct_at` is introduced. But actually the type information is always tracked and thus some conversion to `void*` in context is allowed since C++26, and thus placement new is also possible.

- Default behavior of `allocator_traits` is just return `{allocate(alloc, n), n}`, i.e. allocate requested size exactly.

- Note 2: placement new is `constexpr` **since C++26**, with some restrictions (mentioned in homework of Lecture 11).
  - Before that, you can use template function `std::construct_at` since C++20.
    - And that's why the default behavior of `allocator_traits::allocate` is changed to `std::construct_at` in C++20.

std::allocator_traits<Alloc>::construct

Defined in header <memory>

```
template< class T, class... Args >                          (since C++11)
static void construct( Alloc& a, T* p, Args&&... args );    (constexpr since C++20)
```

If possible, constructs an object of type T in allocated uninitialized storage pointed to by `p`, by calling `a.construct(p, std::forward<Args>(args)...)`.

If the above is not possible (e.g. `Alloc` does not have the member function `construct()`), then calls

```
::new (static_cast<void*>(p)) T(std::forward<Args>(args)...)   (until C++20)
std::construct_at(p, std::forward<Args>(args)...)              (since C++20)
```

Allocator

Special note that `std::construct_at` cannot handle bounded array correctly before Issue 3436: std::construct_at should support arrays.

std::construct_at

Defined in header <memory>

```
template< class T, class... Args >
constexpr T* construct_at( T* location, Args&&... args );
```
(since C++20)

Creates a T object initialized with the arguments in `args` at given address `location`.

```
if constexpr (std::is_array_v<T>)
    return ::new (voidify(*location)) T[1]();
else
    return ::new (voidify(*location)) T(std::forward<Args>(args)...);
```

This `[1]` is to correctly return pointer to array.

Equivalent to , except that `construct_at` may be used in evaluation of constant expressions(until C++26).

- And symmetrically, `std::destroy_at`:

std::destroy_at

Defined in header <memory>

```
template< class T >
void destroy_at( T* p );
```
(since C++17)
(constexpr since C++20)

If T is not an array type, calls the destructor of the object pointed to by `p`, as if by `p->~T()`.

If T is an array type, the program is ill-formed(until C++20)recursively destroys elements of `*p` in order, as if by calling `std::destroy(std::begin(*p), std::end(*p))` (since C++20).

- Actually dtor can be `constexpr`, so this function is introduced in C++17 to act as if `ptr->~auto()`, since the type is deduced by template.

Memory Management

- Allocator
  - Basics
  - PMR

PMR

- We say that allocators are equal when their memory are inter-operatable.
  - So a simple way to implement allocator is to use a static variable to denote its memory arena:

```cpp
class MyStaticAllocator
{
    static constexpr inline std::size_t s_size = 200000;
    static std::array<std::byte, s_size> s_buffer;
public:
    // Just allocate from s_buffer; we omit other components.
};
```

- However, then every arena leads to a unique allocator type…
  - Oops, you have to write template code to support every allocator.
  - And containers don't support e.g. copy for different allocators.
    - You have to write e.g. `.assign(another.begin(), another.end())`.
- Instead, we can use same allocator type, and control memory arena by *polymorphic memory resource* (PMR).

PMR

Defined in <memory_resource>.
All components are defined in namespace std::pmr and thus not repeated afterwards.

- The allocator is just std::pmr::polymorphic_allocator, which controls memory represented by std::pmr::memory_resource.
  - Just like:

```
template<typename T>
class PolymorphicAllocator
{
    MemoryResource* mr_;
public:
    T* allocate(std::size_t n)
    { // Use the memory resource to control allocation.
        return static_cast<T*>(mr_->allocate(n * sizeof(T), alignof(T)));
    }
};
```

You can change allocation strategy without changing the type of the allocator.

  - memory_resource needs to expose interface below:

Public member functions

allocate	allocates memory (public member function)
deallocate	deallocates memory (public member function)
is_equal	compare for equality with another memory_resource (public member function)

Non-member-functions

operator== operator!= (removed in C++20)	compare two memory_resources (function)

PMR

- To define a customized resource, just inherit `memory_resource` and override private virtual methods:

Private member functions

do_allocate [virtual]	allocates memory (virtual private member function)
do_deallocate [virtual]	deallocates memory (virtual private member function)
do_is_equal [virtual]	compare for equality with another `memory_resource` (virtual private member function)

  - So essentially `memory_resource` just uses template method pattern.
- A very naïve example:

```
class NewDeleteResource : public std::pmr::memory_resource
{
public:
    void* do_allocate(std::size_t bytes, std::size_t alignment) override
    {
        return ::operator new(bytes, std::align_val_t{ alignment });
    }
```

```
    void do_deallocate(void* p, std::size_t bytes, std::size_t alignment) override
    {
        return ::operator delete(p, bytes, std::align_val_t{ alignment });
    }

    bool do_is_equal(const std::pmr::memory_resource& other) const noexcept override
    { // All NewDeleteResource are equal, since they manage the same arena.
        return dynamic_cast<const NewDeleteResource*>(&other) != nullptr;
    }
};
```

PMR

- There are some predefined memory resources in standard library.
  - Two naïve singleton resources that are returned by function:
    1. `null_memory_resource()`: allocate nothing.

std::pmr::null_memory_resource

Defined in header <memory_resource>

```
std::pmr::memory_resource* null_memory_resource() noexcept;    (since C++17)
```

Returns a pointer to a memory_resource that doesn't perform any allocation.

Return value

Returns a pointer p to a static storage duration object of a type derived from `std::pmr::memory_resource`, with the following properties:

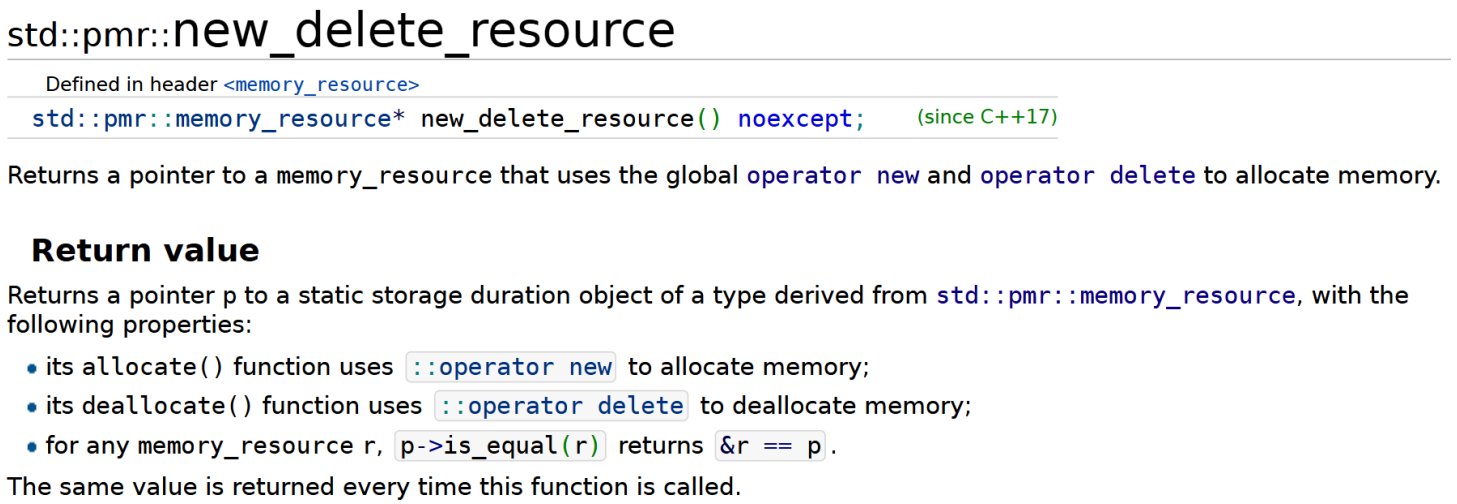
- its `allocate()` function always throws `std::bad_alloc`;
- its `deallocate()` function has no effect;
- for any `memory_resource r`, `p->is_equal(r)` returns `&r == p`.

The same value is returned every time this function is called.

Singleton can be compared just by singleton address, no need to use `dynamic_cast`.

PMR

2. `new_delete_resource()`: use operator new and delete.



std::pmr::new_delete_resource

Defined in header <memory_resource>

```
std::pmr::memory_resource* new_delete_resource() noexcept;
```
(since C++17)

Returns a pointer to a memory_resource that uses the global operator new and operator delete to allocate memory.

Return value

Returns a pointer p to a static storage duration object of a type derived from std::pmr::memory_resource, with the following properties:

- its allocate() function uses ::operator new to allocate memory;
- its deallocate() function uses ::operator delete to deallocate memory;
- for any memory_resource r, p->is_equal(r) returns &r == p.

The same value is returned every time this function is called.

- There exists “global default resource”, which can be got/set (thread-safely) by `get_default_resource()` and `set_default_resource(ptr_to_mr) -> old_ptr`.
  - Quite like `std::get/set_new_handler()`.
- The initial default resource is just `new_delete_resource()`.

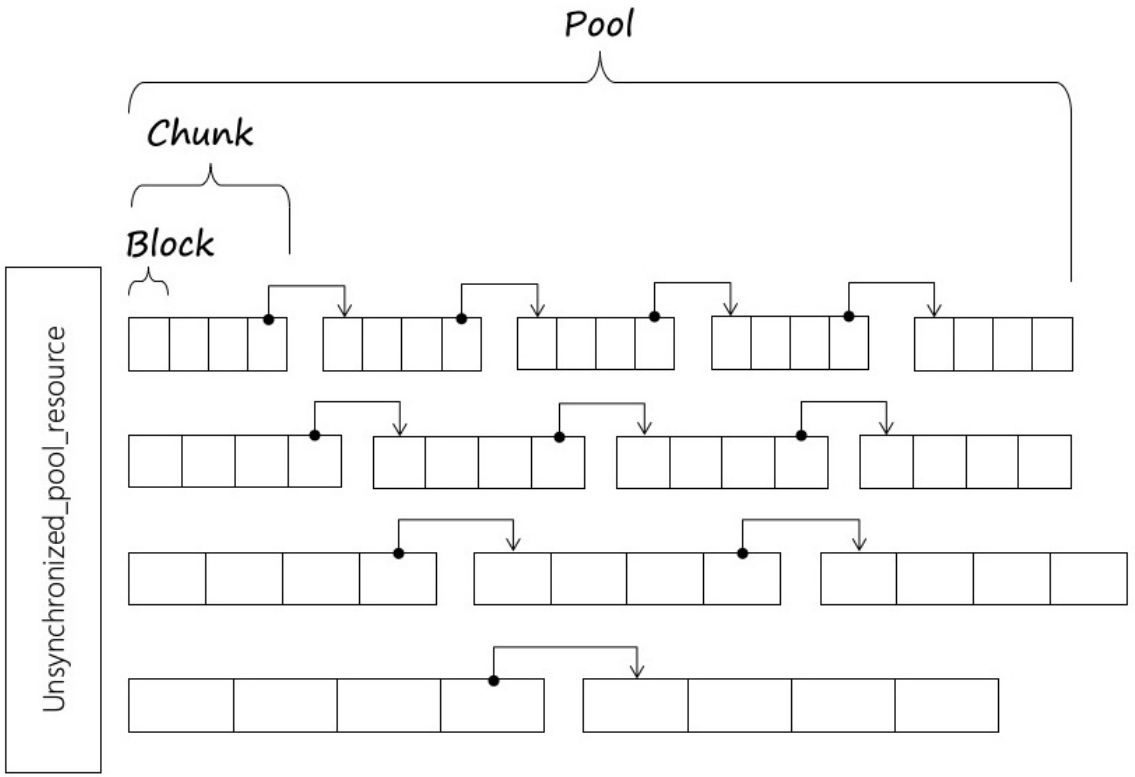
PMR

- And three complex classes.
  - They all have an “upstream” memory resource; when their own memory are exhausted, they will request more from the upstream.
    - The default upstream is just global default resource.

1. Memory pool: `synchronized_pool_resource/unsynchronized_pool_resource`;
   - It consists of a collection of chunk pools (bins), with each one owning many blocks of the same size.
     - Just like what we mentioned in allocation strategy in sized-delete.
     - Allocations are dispatched to the smallest bin that can accommodate required size.
     - Deallocation returns memory to pool, and pool may or may not deallocate blocks further by releasing to the upstream memory.
   - And `unsynchronized_pool_resource` assumes all allocations happen in the same thread (i.e. no possible races), which is likely to be faster than `synchronized`.

PMR

- Just like:



Credit: Thanks for the memory (allocator) - Sticky Bits - Powered by Feabhas

PMR

- You can configure the pool by `struct pool_options`, with two data members:
  - `std::size_t max_blocks_per_chunk`: when a pool allocates new chunk from upstream memory, how many blocks are allowed to allocate at once.
    - Implementation may use a smaller value than specified.
  - `std::size_t largest_required_pool_block`: the maximum block size in pool; if the required size is larger, it will be allocated from upstream memory directly.
    - Implementation may use a pass-through threshold.
  - There exists an implementation-defined limit for each member, which will be used when some greater-than-limit value or 0 is provided.
- Non-inherited APIs are easy:

Public member functions

release	release all allocated memory (public member function)	Equiv. to call `upstream->deallocate()`.
upstream_resource	returns a pointer to the upstream memory resource (public member function)	
options	returns the options that control the pooling behavior of this resource (public member function)	

PMR

`Pool(const pool_options& opts, memory_resource* upstream)`, and every parameter can be omitted (filled with default). For example, here default ctor equiv. to `{ pool_options{}, get_default_resource() }`.

- For example:

```cpp
std::pmr::unsynchronized_pool_resource pool;
auto PrintDiff = [&pool]<typename T>() {
    std::pmr::list<T> list{ &pool };
    auto* lastPtr = &list.emplace_back();
    for (int i = 1; i < 5; i++)
    {
        auto& val = list.emplace_back();
        std::println("{}", reinterpret_cast<std::intptr_t>(&val) -
                           reinterpret_cast<std::intptr_t>(lastPtr));
        lastPtr = &val;
    }
};

PrintDiff.operator()<int>();
std::println("Array case: ");
PrintDiff.operator()<std::array<int, 40>>();
```

```
32
36256
32
32
Array case:
256
256
768
256
```

`std::pmr::list<T>` is alias of `std::list<T,std::pmr::polymorphic_allocator>`.

Actually constructed by `polymorphic_allocator`, which can be further constructed by `memory_resource*` (implicitly).

```cpp
template <class _Value_
struct _List_node { //
    using value_type =
    using _Nodeptr   =
    _Nodeptr _Next; //
    _Nodeptr _Prev; //
    _Value_type _Myval
```

Though `sizeof(Node)` is 24 or 176, the block size is 32 & 256, which is allocated as a whole.

Blocks of the chunk have been exhausted, which then allocate from upstream memory, leading to jumping interval.

By memory resources, we can arrange nodes to locate close to each other, making it much more cache-friendly.

PMR

- 2. monotonic memory: `monotonic_buffer_resource`.
  - Allocated memory will NEVER be deallocated; new allocations will fill buffer *monotonically*.
    - Thus, allocation and deallocation are very fast.
  - For example: (Ignore align)

  - You can provide an initial buffer (not managed by the class); when it's exhausted, `monotonic_buffer_resource` will request from upstream, and
    1. Like vector reallocation, requested size will increase exponentially.
    2. All requested memory will be released (i.e. call `upstream->deallocate()`) when `monotonic_buffer_resource` is destructed or its `.release()` is called.
       - `.release()` will reset buffer and size to initial buffer and size.

PMR

*Note that reallocation strategy differs for different implementation. And here we use MS-STL with Visual Studio Release mode. In Debug mode, MS-STL may allocate more memory for safety check.

- For example:

Use stack as the initial buffer to make allocation very fast.

```
std::array<std::byte, 64> buffer;
std::println("Stack buffer address: {}", (void*)buffer.data());
std::pmr::monotonic_buffer_resource pool{ buffer.data(), buffer.size() };
std::pmr::vector<int> vec(4, 1, &pool);
std::size_t lastCapacity = vec.capacity();
for (int i = 0; i < 6; i++)
{
    vec.push_back(i);
    if (auto newCapacity = vec.capacity(); newCapacity != lastCapacity)
    {
        std::println("Reallocation {}: new address at: {}",
                     newCapacity, (void*)vec.data());
        lastCapacity = newCapacity;
    }
}
```

```
Stack buffer address: 0x4fce7cfa78
Reallocation 6: new address at: 0x4fce7cfa88
Reallocation 9: new address at: 0x1cfb8512c50
Reallocation 13: new address at: 0x1cfb8510570
```

Allocated on the stack.

Since the buffer can at most hold 16 int, new buffer must be allocated from upstream.
And though 13 ints can be accommodated inside the initial buffer, `monotonic_buffer_resource` will never go back so it's still allocated on heap.

PMR

- And note ctors of `monotonic_buffer_resource`:

std::pmr::monotonic_buffer_resource::monotonic_buffer_resource

```
monotonic_buffer_resource();                                                    (1)  (since C++17)
explicit monotonic_buffer_resource( std::pmr::memory_resource* upstream );      (2)  (since C++17)
explicit monotonic_buffer_resource( std::size_t initial_size );                 (3)  (since C++17)
monotonic_buffer_resource( std::size_t initial_size,
                           std::pmr::memory_resource* upstream );               (4)  (since C++17)
monotonic_buffer_resource( void* buffer, std::size_t buffer_size );             (5)  (since C++17)
monotonic_buffer_resource( void* buffer, std::size_t buffer_size,
                           std::pmr::memory_resource* upstream );               (6)  (since C++17)
```

(1~4): set current buffer to `nullptr`; when an allocation happens, allocate from upstream with `initial_size` as the initial buffer. Absent `initial_size` will be some implementation-defined value, and absent upstream will use default resource.

(5, 6): set with explicit buffer; absent upstream will use default resource.

PMR

- Of course, you can combine these resources by setting upstream.
  - Like our previous example in coroutine:

```
class CoroTask {
    inline static std::array<std::::byte, 200000> memory;

    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);
        }
    };
}
```

When allocation fails, don’t request more memory; instead throw an exception.

PMR

- Note 1: lots of raw pointers are used in PMR and thus lifetime should be carefully managed.
  1. You need to ensure upstream resource and initial buffer to be valid when they’re used.
  2. When you call `set_default_resource`, the original resource shouldn’t be destroyed immediately since previously allocated container may use it.
  3. When you destruct objects (or equivalently, call `.release()`) of `monotonic_buffer_resource/(un)synchronized_pool_resource`, you need to ensure nothing occupies its allocated memory.

*There exists DR to fix previous unnecessary `dynamic_cast`.

PMR

- Note 2: these classes have some special properties:
  - They are neither copyable nor moveable.
  - Different resource objects are always unequal*.

Return value

`this == &other`

    - Though theoretically these memory resources may be inter-operatable, the standard library chooses to conservatively mark them as unequal.
- Note 3: PMR are "sticky"; it doesn't POCMA, POCCA and POCS.
  - And SOCCC just returns default constructed allocator (which uses default memory resource, instead of same resource as copied allocator).
  - And remember that…
    - Particularly, when POCS is `false` and allocators are unequal, direct swap is not well-defined (and standard containers mark it as UB).
  - This leads to the fact that it's usually UB to swap two `std::pmr::container` that have unequal `polymorphic_allocator`.
    - Particularly, different standard MRs are always unequal, so such code is UB:

```
std::pmr::unsynchronized_pool_resource pool1, pool2;
std::pmr::vector<int> v1{ &pool1 }, v2{ &pool2 };
v1.push_back(1), v2.push_back(2);
v1.swap(v2); // UB for !POCS && !equal
```

PMR

- Note 4: since C++20, `polymorphic_allocator` adds some utilities to avoid rebinding.

allocate_bytes (C++20) **deallocate_bytes** (C++20)	Equiv. to use resource to allocate bytes directly (instead of `* sizeof(T)`). Default alignment parameter is `alignof(std::max_align_t)` instead of `alignof(T)`.
allocate_object (C++20) **deallocate_object** (C++20)	With template parameter `<U>`, equiv. to use `polymorphic_allocator<U>` with the same memory resource to do allocation.
new_object (C++20) **delete_object** (C++20)	With template parameter `<U>`, equiv. to use `polymorphic_allocator<U>` with the same memory resource to do allocation and construction. Exception-safe, i.e. when construction fails, space will be deallocated.

And C++20 also adds `std::byte` as default template type parameter for `polymorphic_allocator`.
Personally I think it's a design mistake, in line with Arthur O'Dwyer.

Uses-allocator Construction

- Finally, PMR will perform *uses-allocator construction* to "down-propagate" allocators.
  - That is, `polymorphic_allocator::construct` will try to pass itself to the constructed object.
  - A naïve example:

```
std::pmr::monotonic_buffer_resource pool;

std::pmr::vector<std::pmr::string> coll{&pool};
for (int i=0; i < 100; ++i) {
  coll.emplace_back("just a non-SSO string");
}
```

    - Here since `std::pmr::string` can accept `polymorphic_allocator`, allocator of the vector will be passed to newly constructed `std::pmr::string`.
      - So string will allocate its memory from `pool` too.

Uses-allocator Construction

- Essentially, whether allocator is propagated to new object of type `T` is judged by procedures below:
  1. Judge whether `T` uses allocator by trait `std::uses_allocator_v<T>`.
     - It's `true` if `T` has nested type `allocator_type`.
     - Or you can specialize it manually, like many other types:
  2. If 1. is `true`, find the way to pass the allocator;
     ① First try *leading-allocator convention*, i.e. ctor is invocable by `T(std::allocator_arg, alloc, args...)`.
     ② If it fails, then try *trailing-allocator convention*, i.e. ctor is invocable by `T(args..., alloc)`.
     ③ Otherwise ill-formed.
- If we finally find the way, allocator is propagated by ctor.

std::uses_allocator<std::tuple> (C++11)
std::uses_allocator<std::queue> (C++11)
std::uses_allocator<std::priority_queue> (C++11)
std::uses_allocator<std::stack> (C++11)
std::uses_allocator<std::flat_map> (C++23)
std::uses_allocator<std::flat_set> (C++23)
std::uses_allocator<std::flat_multimap> (C++23)
std::uses_allocator<std::flat_multiset> (C++23)
std::uses_allocator<std::function> (C++11)(until C++17)
std::uses_allocator<std::promise> (C++11)
std::uses_allocator<std::packaged_task> (C++11)(until C++17)

Uses-allocator Construction

- Since C++20, this process is wrapped as `std::make_obj_using_allocator<T>(alloc, args…);`
  - Essentially: Equivalent to

```
return std::make_from_tuple<T>(
    std::uses_allocator_construction_args<T>(alloc, std::forward<Args>(args)...)
);
```

    - where `std::make_from_tuple<T>` in `<tuple>` is same as unpacking the tuple and construct `T` (quite like `std::apply`);
    - And `std::uses_allocator_construction_args<T>` judges the calling convention, and return the corresponding tuple that satisfies the convention;
      - Particularly, if `std::uses_allocator_v<T>` is `false`, then just ignore `alloc` and return tuple of reference to `args`.

Uses-allocator Construction

- And if you want to placement new, you can use `std::uninitialized_construct_using_allocator<T>(p, alloc, args…)`.

Equivalent to

```
return std::apply(
    [&]<class... Xs>(Xs&&...xs)
    {
        return std::construct_at(p, std::forward<Xs>(xs)...);
    },
    std::uses_allocator_construction_args<T>(alloc, std::forward<Args>(args)...));
```

  - Thus `construct` of `polymorphic_allocator` can be easily implemented:

```
template<class U, class... Args>
void construct(U *where, Args&&... args) {
    std::uninitialized_construct_using_allocator<U>(
        where, *this, std::forward<Args>(args)...
    );
}
```

Uses-allocator Construction

- Note 1: Pay attention to object that's constructed outside the container, leading to possible inefficiency.
  - For example, explain what happens in code below:

```
char buf[10'000];
auto mr = std::pmr::monotonic_buffer_resource(buf, sizeof(buf));
auto v = std::pmr::vector<std::pmr::string>(&mr);
v.push_back("hello world, for example");
assert(v[0].get_allocator().resource() == &mr);
```

  - `.push_back` accepts `std::pmr::string`, so it's constructed outside.
    - From `const char*` to `string`, default constructed allocator will be used if it's not provided, meaning that the default memory resource will be used!
  - As PMR is sticky and non-equal, element-wise move will happen (and in `string` it's just **equivalent to copy**).
  - Solution: either by `.emplace_back`, or construct with allocator explicitly.

Uses-allocator Construction

- Note 2: `std::uses_allocator_v<std::pair<…>>` is `false`, but `std::pair` actually **uses allocator for its two elements**.
  - Its ctors don't fulfill calling conventions, so uses-allocator construction provides lots of overloads for it (each one corresponds to a ctor overload).

T is a specialization of `std::pair`

```
template< class T, class Alloc, class Tuple1, class Tuple2 >
constexpr auto uses_allocator_construction_args( const Alloc& alloc,                (2)  (since C++20)
    std::piecewise_construct_t, Tuple1&& x, Tuple2&& y ) noexcept;

template< class T, class Alloc >
constexpr auto uses_allocator_construction_args( const Alloc& alloc ) noexcept;     (3)  (since C++20)

template< class T, class Alloc, class U, class V >
constexpr auto uses_allocator_construction_args( const Alloc& alloc,                (4)  (since C++20)
    U&& u, V&& v ) noexcept;

template< class T, class Alloc, class U, class V >
constexpr auto uses_allocator_construction_args( const Alloc& alloc,                (5)  (since C++23)
    std::pair<U, V>& pr ) noexcept;

template< class T, class Alloc, class U, class V >
constexpr auto uses_allocator_construction_args( const Alloc& alloc,                (6)  (since C++20)
    const std::pair<U, V>& pr ) noexcept;

template< class T, class Alloc, class U, class V >
constexpr auto uses_allocator_construction_args( const Alloc& alloc,                (7)  (since C++20)
    std::pair<U, V>&& pr ) noexcept;

template< class T, class Alloc, class U, class V >
constexpr auto uses_allocator_construction_args( const Alloc& alloc,                (8)  (since C++23)
    const std::pair<U, V>&& pr ) noexcept;

template< class T, class Alloc, class NonPair >
constexpr auto uses_allocator_construction_args( const Alloc& alloc,                (9)  (since C++20)
    NonPair&& non_pair ) noexcept;
```

Uses-allocator Construction

- By contrast, `std::tuple` uses leading-allocator convention, so it just specializes `std::uses_allocator` without other special treatments:

Allocator-extended constructors

`template< class Alloc >` `tuple( std::allocator_arg_t, const Alloc& a );`	(15)	(since C++11) (constexpr since C++20) (conditionally explicit)

`template< class Alloc >`
`tuple( std::allocator_arg_t, const Alloc& a,`
`  const Types&... args );`
	(16)	(since C++11)
		(constexpr since C++20)
		(conditionally explicit)

`template< class Alloc, class... UTypes >`
`tuple( std::allocator_arg_t, const Alloc& a,`
`  UTypes&&... args );`
	(17)	(since C++11)
		(constexpr since C++20)
		(conditionally explicit)

`template< class Alloc, class... UTypes >`
`constexpr tuple( std::allocator_arg_t, const Alloc& a,`
`  tuple<UTypes...>& other );`
	(18)	(since C++23)
		(conditionally explicit)

`template< class Alloc, class... UTypes >`		(since C++11)

```cpp
template<class T>
struct Mallocator
{
    using value_type = T;

    std::string name = "EVIL";
    Mallocator() = default;
    Mallocator(std::string init_name) : name{ std::move(init_name) } {}

    template<class U>
    constexpr Mallocator(const Mallocator <U>& another) noexcept: name{ another.name } {}
```

std::scoped_allocator_adaptor

Defined in header <scoped_allocator>

```cpp
template< class OuterAlloc, class... InnerAllocs >
class scoped_allocator_adaptor
    : public OuterAlloc;
```

Defined in <scoped_allocator>

- Note 3: standard library also provides an allocator that "propagates" allocators to construct subobjects.
  - To be exact, it collects allocators at once and dispatch them level by level.
  - For example, for vector of vector of int…
    - We need to use two allocators, for int and vector of int respectively.

```cpp
using Alloc = Mallocator<int>;
using VAlloc = Mallocator<std::vector<int, Alloc>>;

VAlloc a{ "ALICE" };
Alloc b{ "BOB" };
```

  - So normally, we need to write code like:

```cpp
std::vector<std::vector<int, Alloc>, VAlloc> vec{ a };
auto& newSubvec = vec.emplace_back(b);
newSubvec.emplace_back();
```

```
From [ALICE]: Alloc: 56 bytes at 0000023A66AD86C0
From [BOB]: Alloc: 4 bytes at 0000023A66AE5D50
From [BOB]: Dealloc: 4 bytes at 0000023A66AE5D50
From [ALICE]: Dealloc: 56 bytes at 0000023A66AD86C0
```

    - What if we forget to pass b?

Uses-allocator Construction

- Then the new vector uses default-constructed allocator, instead of `b`.

```
std::vector<std::vector<int, Alloc>, VAlloc> vec{ a };
auto& newSubvec = vec.emplace_back(b);
newSubvec.emplace_back();
// Oops!
auto& whatSubVec = vec.emplace_back();
std::cout << "---------------FOCUS ON---------------\n";
whatSubVec.emplace_back();
std::cout << "---------------FOCUS ON---------------\n";
```

```
From [ALICE]: Alloc: 56 bytes at 000002B47AC28580
From [BOB]: Alloc: 4 bytes at 000002B47AC34A70
From [ALICE]: Alloc: 112 bytes at 000002B47AC2DA40
From [ALICE]: Dealloc: 56 bytes at 000002B47AC28580
---------------FOCUS ON---------------
From [EVIL]: Alloc: 4 bytes at 000002B47AC34B50
---------------FOCUS ON---------------
From [BOB]: Dealloc: 4 bytes at 000002B47AC34A70
From [EVIL]: Dealloc: 4 bytes at 000002B47AC34B50
From [ALICE]: Dealloc: 112 bytes at 000002B47AC2DA40
```

- If we use scoped allocator:

```
using ScopedAlloc = std::scoped_allocator_adaptor<VAlloc, Alloc>;
ScopedAlloc alloc{ a, b };
std::vector<std::vector<int, Alloc>, ScopedAlloc> vec{ alloc };
auto& newSubvec = vec.emplace_back(); // This uses a, and passes b.
newSubvec.emplace_back();             // This passes b.
```

```
Microsoft Visual Studio 调试控
From [ALICE]: Alloc: 56 bytes at 0000017E81818E40
From [BOB]: Alloc: 4 bytes at 0000017E818265C0
From [BOB]: Dealloc: 4 bytes at 0000017E818265C0
From [ALICE]: Dealloc: 56 bytes at 0000017E81818E40
```

Uses-allocator Construction

- For nested container, you need to use nested scoped allocator:

`Scoped<Outer, Inners…>{ o, is… }` will use `o` to allocate and construct subobject `S`; the construction passes `Scoped<Inners…>{ is… }` to do uses-allocator construction.

```
using Alloc = Mallocator<int>;
using Vec = std::vector<int, Alloc>;

using VAlloc = Mallocator<Vec>;
using ScopedAlloc1 = std::scoped_allocator_adaptor<VAlloc, Alloc>;
using VVec = std::vector<Vec, ScopedAlloc1>;

using LVAlloc = Mallocator<VVec>;
using ScopedAlloc2 = std::scoped_allocator_adaptor<LVAlloc, VAlloc, Alloc>;
using LVVec = std::list<VVec, ScopedAlloc2>;

LVAlloc a{ "ALICE" };
VAlloc b{ "BOB" };
Alloc c{ "CARL" };

ScopedAlloc2 alloc{ a, b, c };
LVVec list{ alloc };
list.emplace_back().emplace_back().emplace_back();
```

```
Microsoft Visual Studio 调试控
From [ALICE]: Alloc: 104
From [ALICE]: Alloc: 104
From [BOB]: Alloc: 56 byt
From [CARL]: Alloc: 4 byt
From [CARL]: Dealloc: 4 b
From [BOB]: Dealloc: 56 b
From [ALICE]: Dealloc: 10
From [ALICE]: Dealloc: 10
```

Uses-allocator Construc

std::scoped_allocator_adaptor

Defined in header <scoped_allocator>

```
template< class OuterAlloc, class... InnerAllocs >
class scoped_allocator_adaptor
    : public OuterAlloc;
```

- If you stop to use scoped allocator, then uses-allocator construction cannot find the calling convention and thus forward without allocator.

Process:

1. `LVVec` uses `LVAlloc` (`a`) to construct, and passes `ScopedAlloc1` (`b+c`) to `VVec`;

2. `VVec` accepts `VAlloc`, while `ScopedAlloc1` inherits from `VAlloc`, thus `VVec` is constructed with `b`.
   - `c` is discarded here!

3. `VVec` uses `VAlloc` (`b`) to construct; as it's not scoped allocator, `Vec` is normally constructed with default `Alloc`.

```
using Alloc = Mallocator<int>;
using Vec = std::vector<int, Alloc>;

using VAlloc = Mallocator<Vec>;
using ScopedAlloc1 = std::scoped_allocator_adaptor<VAlloc, Alloc>;
using VVec = std::vector<Vec, VAlloc>;

using LVAlloc = Mallocator<VVec>;
using ScopedAlloc2 = std::scoped_allocator_adaptor<LVAlloc, VAlloc, Alloc>;
using LVVec = std::list<VVec, ScopedAlloc2>;

LVAlloc a{ "ALICE" };
VAlloc b{ "BOB" };
Alloc c{ "CARL" };

ScopedAlloc2 alloc{ a, b, c };
LVVec list{ alloc };
list.emplace_back().emplace_back().emplace_back();
```

```
Microsoft Visual Studio 调试控
From [ALICE]: Alloc: 72
From [ALICE]: Alloc: 72
From [BOB]: Alloc: 56 b
From [EVIL]: Alloc: 4 b
From [EVIL]: Dealloc: 4
From [BOB]: Dealloc: 56
From [ALICE]: Dealloc:
From [ALICE]: Dealloc:
```

Summary

- Low-level memory management
  - Object layout, alignment
  - operator new & delete
- Smart Pointers
  - unique_ptr
    - pimpl
    - indirect and polymorphic
  - shared_ptr, weak_ptr
    - atomic specialization
  - out_ptr, inout_ptr
- Allocators
  - Interface and traits (POCMA, POCCA, POCS, SOCCC).
  - PMR
    - Uses-allocator construction

Next lecture…

- Eventually, we've successfully finished all major topics!
- In the final lecture, we'll quickly cover topics that are left out.
  - File system;
  - Chrono (time-related facilities);
  - Math (including random-number generation);
  - And finally, we'll give a rough introduction to C++26 for future vision.