An allocator is a handle to a heap

Lessons learned from std::pmr

Arthur O’Dwyer
2018-09-24

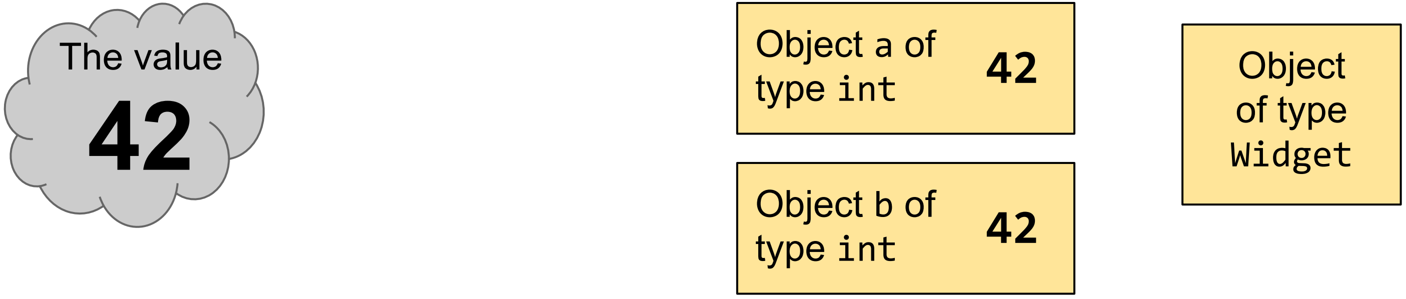
Outline

- What are objects? What are values? [3–11]
- An Allocator is a handle to a MemoryResource [12–19]
- An Allocator is a "copy-only" type [20–23]
- An Allocator belongs to a rebindable family [24–30]
- An Allocator is more than a handle to a MemoryResource [31–45]
- Relating allocators to other parts of C++ [46–54]
- Examples and bonus slides [55–61]

Hey look!
Slide numbers!

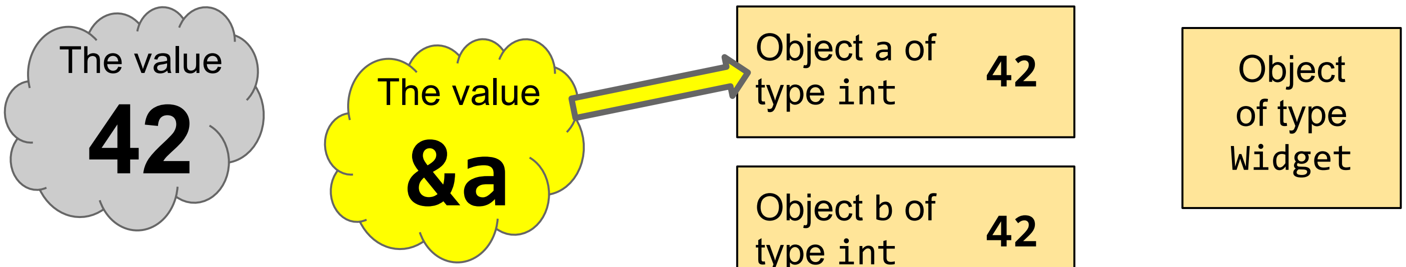
What is an object?

- An object, unlike a (pure) value, has an ***address***.
- Address, pointer, name, unique identifier, handle — all synonymous for our purposes.



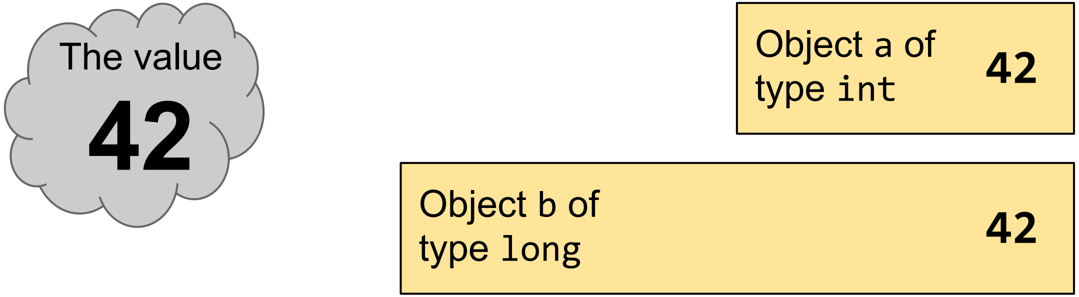
What is an object?

- An object, unlike a (pure) value, has an ***address***.
- Address, pointer, name, unique identifier, handle — all synonymous for our purposes.
- The name of an object *is itself* a value.



What is an object?

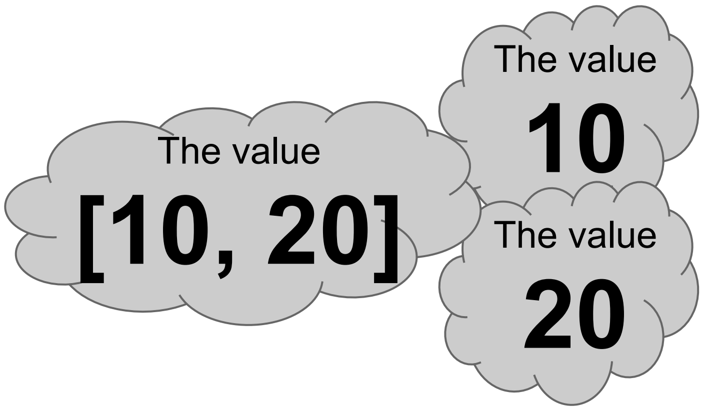
Where it gets confusing (for me at least): A C++ object is defined in part by its **in-memory representation**. And there is some sense in which some kinds of objects can "have" a value at any given moment.

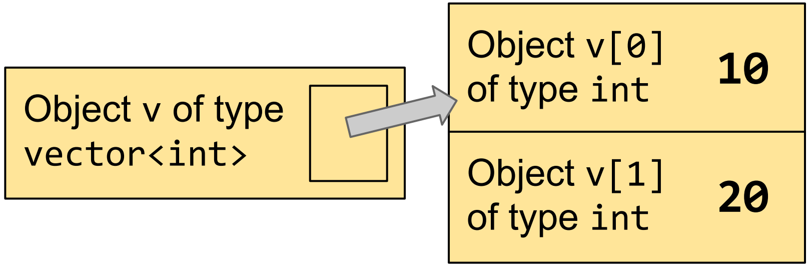


What is a (sequence) container?

A container is a **value**, containing sub-values, which are called its elements.

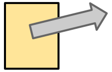
A container is an **object**, that holds and manages its elements, which are also objects.

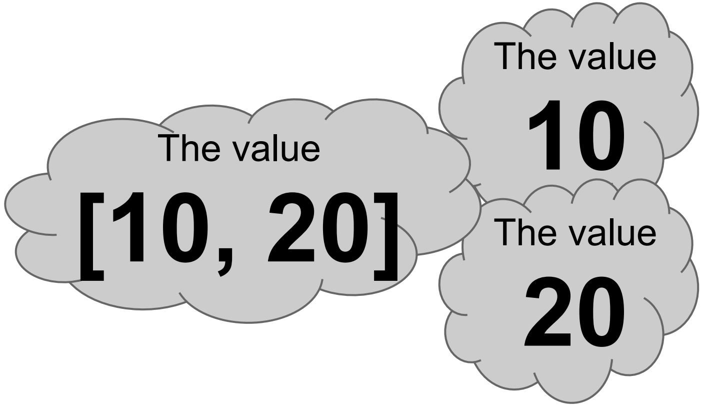


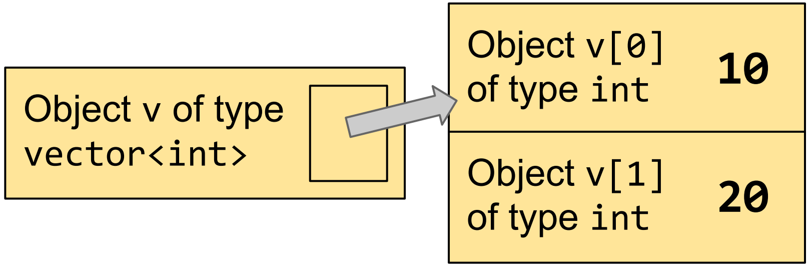


What is an allocator?

The classic C++ allocator model answers the questions implied by the object diagram on the right.

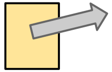
- Where does the memory for `v[i]` come from?
- What is the thing represented by  in our diagram?





What is an allocator?

`std::vector` is parameterized on an allocator type `A` as well as on `T`.

- Where does the memory for `v[i]` come from?
  - It comes from `A::allocate(n)`.
- What is the thing represented by  in our diagram?
  - It's an object of type `A::pointer`.

The container holds an *instance* of the allocator type `A` within itself. Anything that can be funneled through that instance, is funneled.

What can we put inside the allocator instance?

What goes into an allocator?

The only allocator type in C++03 / 11 / 14 is `std::allocator`.

`std::allocator` is stateless.

This led to people's trying to implement the wrong kind of state**ful** allocators.

```
template<class T>
struct Bad {
    alignas(16) char data[1024];  // "state" = big buffer to allocate from
    size_t used = 0;
    T *allocate(size_t n) {
        auto k = n*sizeof(T); used += k;
        return (T*)(data + used - k);
    }
};
```

std::pmr::polymorphic_allocator

C++17 adds `std::pmr::polymorphic_allocator`.

- Basically a pointer to a `std::pmr::memory_resource`.
- All the shared state goes into the `memory_resource`.

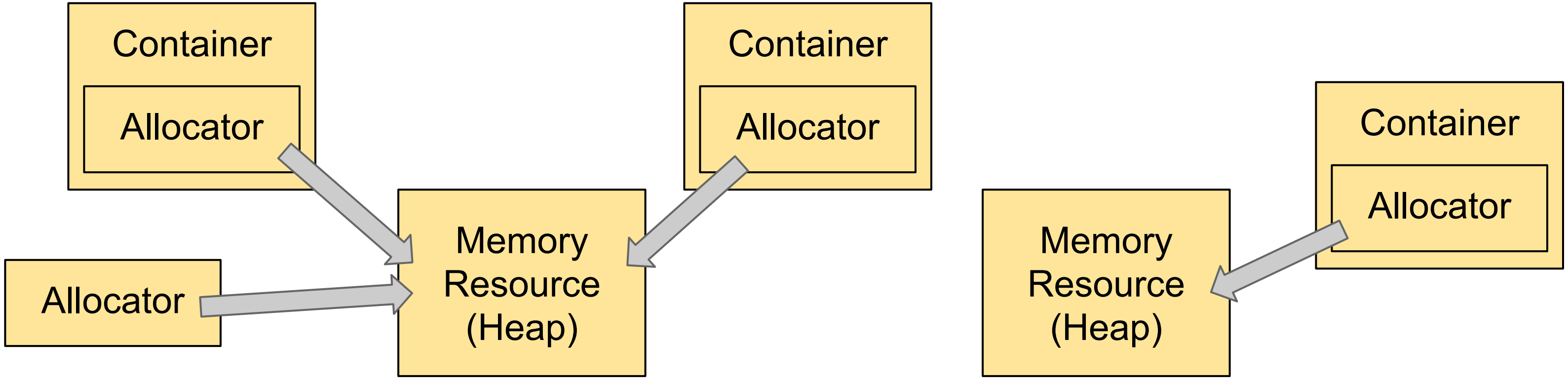
```
template<class T>
struct TrivialResource : std::pmr::memory_resource {
    alignas(16) char data[1024];  // "state" = big buffer to allocate from
    size_t used = 0;
    T *allocate(size_t n) {
        auto k = n*sizeof(T); used += k;
        return (T*)(data + used - k);
    }
};
TrivialResource<int, 10> mr;
std::vector<int, polymorphic_allocator<int>> fcvec(&mr);
```

“Object-like” bad; “value-like” good

`struct Bad`	`std::pmr::polymorphic_allocator`
Object-like, contains mutable state	Value-like, contains only immutable state
“Source of memory” is stored directly in the object	“Source of memory” is shared among all copies of the allocator with the same *value*
`allocate` and `deallocate` are non-const	`allocate` and `deallocate` could be const
Moving/copying the allocator object is likely to cause bugs	Moving/copying the allocator is safe and even encouraged
There is no shared state	Need to think about lifetime of the shared state

Clarifies our thinking about allocators

- Old-style thinking: “an allocator *object* represents a source of memory” — ***WRONG!***
- New-style thinking: “an allocator *value* represents *a handle to* a source of memory (plus some other, orthogonal pieces).”



But what about stateless allocators?

A stateless allocator [e.g. `std::allocator<T>`] represents a handle to a source of memory (plus some orthogonal pieces) where the source of memory is a global singleton [e.g. the `new`/`delete` heap].

- A datatype with k possible values needs only $\log_2 k$ bits.
- A pointer to a global singleton (with 1 possible value) needs $\log_2 1 = 0$ bits.

```cpp
class memory_resource {
public:
   void *allocate(size_t bytes, size_t align = alignof(max_align_t)) {
       return do_allocate(bytes, align);
   }
   void deallocate(void *p, size_t bytes, size_t align = alignof(max_align_t)) {
       return do_deallocate(p, bytes, align);
   }
   bool is_equal(const memory_resource& rhs) const noexcept {
       return do_is_equal(rhs);
   }
   virtual ~memory_resource() = default;
private:
   virtual void *do_allocate(size_t bytes, size_t align) = 0;
   virtual void do_deallocate(void *p, size_t bytes, size_t align) = 0;
   virtual bool do_is_equal(const memory_resource& rhs) const noexcept = 0;
};

bool operator==(const memory_resource& a, const memory_resource& b) noexcept {
    return (&a == &b) || a.is_equal(b);
}
```

Standard new_delete_resource()

```
class singleton_new_delete_resource : public memory_resource {
   void *do_allocate(size_t bytes, size_t align) override {
       return ::operator new(bytes, std::align_val_t(align));
   }
   void do_deallocate(void *p, size_t bytes, size_t align) override {
       ::operator delete(p, bytes, std::align_val_t(align));
   }
   bool do_is_equal(const memory_resource& rhs) const noexcept override {
       return (this == &rhs);
   }
};

inline memory_resource *new_delete_resource() noexcept {
   static singleton_new_delete_resource instance;
   return &instance;
}
```

```cpp
template<class T> class polymorphic_allocator {
    memory_resource *m_mr;
public:
    using value_type = T;

    polymorphic_allocator(memory_resource *mr) : m_mr(mr) {}
    template<class U>
    explicit polymorphic_allocator(const polymorphic_allocator<U>& rhs) noexcept
                                    { m_mr = rhs.resource(); }
    polymorphic_allocator()         { m_mr = get_default_resource(); }

    memory_resource *resource() const { return m_mr; }
    T *allocate(size_t n)           { return (T*)(m_mr->allocate(n * sizeof(T), alignof(T))); }
    void deallocate(T *p, size_t n)  { m_mr->deallocate((void*)(p), n * sizeof(T), alignof(T)); }

    polymorphic_allocator select_on_container_copy_construction() const
                                    { return polymorphic_allocator(); }
};

template<class A, class B>
bool operator==(const polymorphic_allocator<A>& a, const polymorphic_allocator<B>& b) noexcept
{ return *a.resource() == *b.resource(); }
```

This stateful allocator of size **64 bits** can “point to” any of **2^{64}** different memory resources.

```cpp
static atomic_refcounted_ptr<memory_resource> s_table[256];

template<class T> class one_byte_allocator {
    uint8_t m_index = 0;
public:
    using value_type = T;

    one_byte_allocator() = delete;
    one_byte_allocator(memory_resource *mr) {
      /* find an empty slot in s_table, insert mr, and inc_ref it */
    }
    one_byte_allocator(const one_byte_allocator& rhs) noexcept
                                   { m_index = rhs.m_index; s_table[m_index].inc_ref(); }
    ~one_byte_allocator() { s_table[m_index].dec_ref(); }

    template<class U>
    explicit one_byte_allocator(const one_byte_allocator<U>& rhs) noexcept
                                   { m_index = rhs.m_index; s_table[m_index].inc_ref(); }

    memory_resource *mr() const { return s_table[m_index].get(); }
    T *allocate(size_t n)          { return (T*)(mr()->allocate(n * sizeof(T), alignof(T))); }
    void deallocate(T *p, size_t n)   { mr()->deallocate((void*)(p), n * sizeof(T), alignof(T)); }
};
```

This stateful allocator of size **8 bits** can “point to” any of **256** different memory resources.

```cpp
template<class T> class zero_byte_allocator {
    /* no state */
public:
    using value_type = T;

    zero_byte_allocator() = default;

    template<class U>
    explicit zero_byte_allocator(const zero_byte_allocator<U>& rhs) noexcept
                                            { }

    memory_resource *mr() const { return std::pmr::new_delete_resource(); }

    T *allocate(size_t n)                   { return (T*)(mr()->allocate(n * sizeof(T), alignof(T))); }
    void deallocate(T *p, size_t n)         { mr()->deallocate((void*)(p), n * sizeof(T), alignof(T)); }
};
```

This stateless allocator of size **0 bits** can "point to" any of **1** different memory resources.

This is essentially `std::allocator`!

Corollaries to the new way of thinking

- Allocator types should be copyable, just like pointers.
    - This was always true but now it's more obvious.
- Allocator types should be cheaply copyable, like pointers.
    - They need not be *trivially* copyable.
- Memory-resource types should generally be immobile.
    - A memory resource might allocate chunks out of a buffer stored inside itself as a data member.

Allocators must “do as the pointers do”

in one more subtle way...

```cpp
static atomic_refcounted_ptr<memory_resource> s_table[256];

template<class T> class one_byte_allocator {
    uint8_t m_index = 0;
public:
    using value_type = T;

    one_byte_allocator() = delete;
    one_byte_allocator(memory_resource *mr) {
      /* find an empty slot in s_table, insert mr, and inc_ref it */
    }
    one_byte_allocator(const one_byte_allocator& rhs) noexcept
                                  { m_index = rhs.m_index; s_table[m_index].inc_ref(); }
    ~one_byte_allocator() { s_table[m_index].dec_ref(); }

    template<class U>
    explicit one_byte_allocator(const one_byte_allocator<U>& rhs) noexcept
                                  { m_index = rhs.m_index; s_table[m_index].inc_ref(); }

    T *allocate(size_t n)         { return (T*)(mr()->allocate(n * sizeof(T), alignof(T))); }
    void deallocate(T *p, size_t n)   { mr()->deallocate((void*)(p), n * sizeof(T), alignof(T)); }
    memory_resource *mr() const { return s_table[m_index].get(); }
};
```

Did you notice that this allocator type is copyable but not (efficiently) moveable?

Allocators must be “copy-only” types

This was LWG issue 2593.

Expression	Return type	Assertion/note/pre-/post-condition
`X u(a);` `X u = a;`		Shall not exit via an exception. *Postconditions:* `u == a`.
`X u(std::move(a));` `X u = std::move(a);`		Shall not exit via an exception. *Postconditions:* The value of a is unchanged and is equal to u.

```
vector<int, A<int>> v1;
vector<int, A<int>> v2 = std::move(v1);  // If we move-from A...
v1.clear(); v1.push_back(42);  // ...this will allocate() using the moved-from A!
```

Allocators must be “copy-only” types

Given that we are not allowed to move allocators any more cheaply than we copy them, how worried do we have to be about the *cost* of a copy?
I wrote a little test to find out. https://wandbox.org/permlink/mHrj7Y55k3Gqu4Q5

Expression	Allocator copies+moves on libc++ (— if stateful)	Allocator copies+moves on libstdc++ (— if stateful)
`list b(a);`	2+2 — 2+2	1+1 — 1+1
`list b = move(a);`	1+2 — 1+2	0+1 — 0+1
`vector b(a);`	2+0 — 2+0	2+0 — 2+0
`vector b = move(a);`	1+0 — 1+0	0+1 — 0+1

We see these extra copies/moves due to *rebinding*.

Allocators are “rebindable family” types

The type-to-allocate is baked into the allocator type.

- `Alloc<int>{}.allocate(2)` means “allocate 2 **`int`**`s`.”
- This works great for `std::vector` — and *only* for `std::vector`.
- `std::list<T, Alloc<T>>` wants to allocate `__node<T>`, not `T`.
- `std::map<K, V, Alloc<pair<const K, V>>>` certainly doesn’t want to allocate that messy type!

Every “allocator type” is really a whole family of related types: `Alloc<int>` and `Alloc<double>` and `Alloc<void***>` and so on. An allocator value which is representable in *one* of the family’s types must be representable in *all* of its types.

Allocators are “rebindable family” types

Rebinding is useful whenever your generic algorithm requires a “`Foo` of `T`,” where *you* provide the `T`(s) but your *user* provides the (single) `Foo`.

```
// Rebinding, preferred by the STL
template<class T, class A_of_T>
class myContainer {
    using actualAllocator = allocator_traits<A_of_T>::rebind_alloc<U>;
};
```

```
// Template template parameters, unused in the STL
template<class T, template<class> class A>
class myContainer {
    using actualAllocator = A<U>;
};
```

Allocators are “rebindable family” types

Other “rebindable families” in C++:

- Allocator types
  - `allocator_traits<Alloc<T>>::rebind_alloc<U> == Alloc<U>`
- Pointer types
  - `pointer_traits<Ptr<T>>::rebind<U> == Ptr<U>`
- Smart-pointer types
  - `decltype(reinterpret_pointer_cast<U>(Sptr<T>{})) == Sptr<U>`

Each “rebindable family” has a prototype

or “representative” of the equivalence class —

- Pointer and smart-pointer families have a “void pointer” type
    - `Ptr<void>`, `Sptr<void>`
- Allocator families have* a “proto-allocator” type
    - `Alloc<void>`

* — Caveat. This “proto-allocator” concept is in the Networking and Executors TSes but not yet in the Standard. And somebody was recently agitating for the proto-allocator type to be spelled `Alloc<std::byte>` instead of `Alloc<void>`! But I hope that’ll get cleared up soon.

If we were designing the STL today...

...we'd use the "proto-allocator" type in our interfaces to save on instantiations.

Current STL:

```
std::vector<int, Alloc<int>>
std::list<int, Alloc<int>>
std::map<int, int, Alloc<pair<const int, int>>>
```

Better, DRYer STL:

```
std::vector<int, Alloc<void>>
std::list<int, Alloc<void>>
std::map<int, int, Alloc<void>>
```

We seem to be circling back to `std::pmr`

With the “proto-allocator” type, the container must rebind its allocator before any operation:

```
ProtoAlloc m_alloc = ...;

using AllocT = allocator_traits<ProtoAlloc>::rebind_alloc<T>;
auto ptr = allocator_traits<AllocT>::allocate(
    static_cast<AllocT>(m_alloc), capacity
);
```

So why don’t we just standardize on this, instead?

```
memory_resource *m_res = ...;
auto ptr = m_res->allocate(capacity * sizeof(T));
```

Because an Allocator is not *merely* a pointer to a MemoryResource

Allocator > source of memory

```
auto ptr = allocator_traits<AllocT>::allocate(
    static_cast<AllocT>(m_alloc), capacity
);
```

`ptr` will be of type `allocator_traits<AllocT>::pointer`

- *probably* `T*`
- but *could* be something fancier, such as `boost::interprocess::offset_ptr<T>`

It's completely up to the allocator to decide how its pointers are represented!

Allocator > source of memory

```
auto ptr = allocator_traits<AllocT>::allocate(
    static_cast<AllocT>(m_alloc), capacity
);
```

`ptr` will be of type `allocator_traits<AllocT>::pointer`

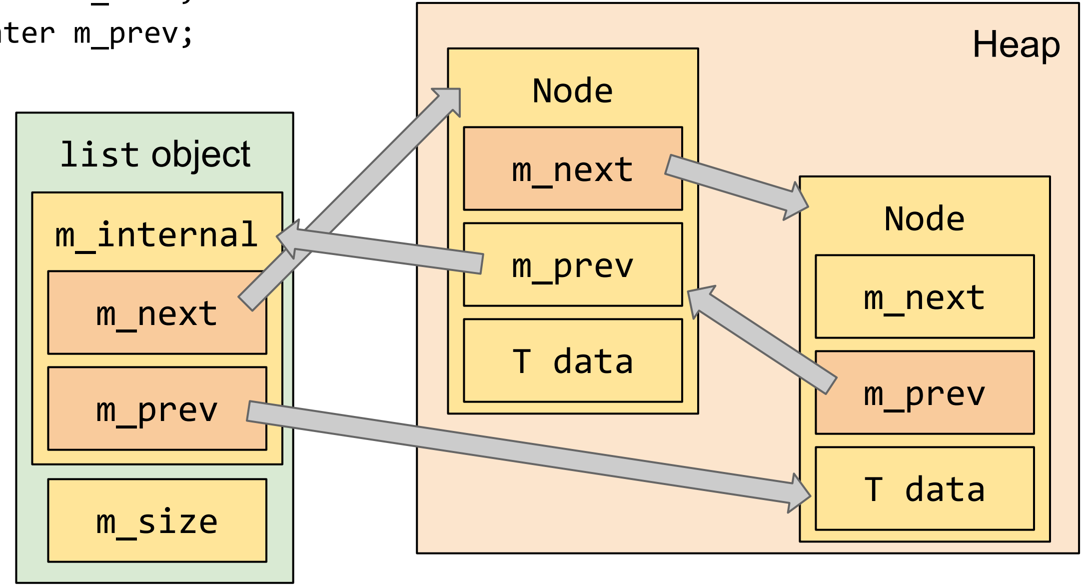
- *probably* `T*`
- but *could* be something fancier, such as `boost::interprocess::offset_ptr<T>`

It’s completely up to the allocator to decide how its pointers are represented!

But “**its** pointers” is awfully vague...

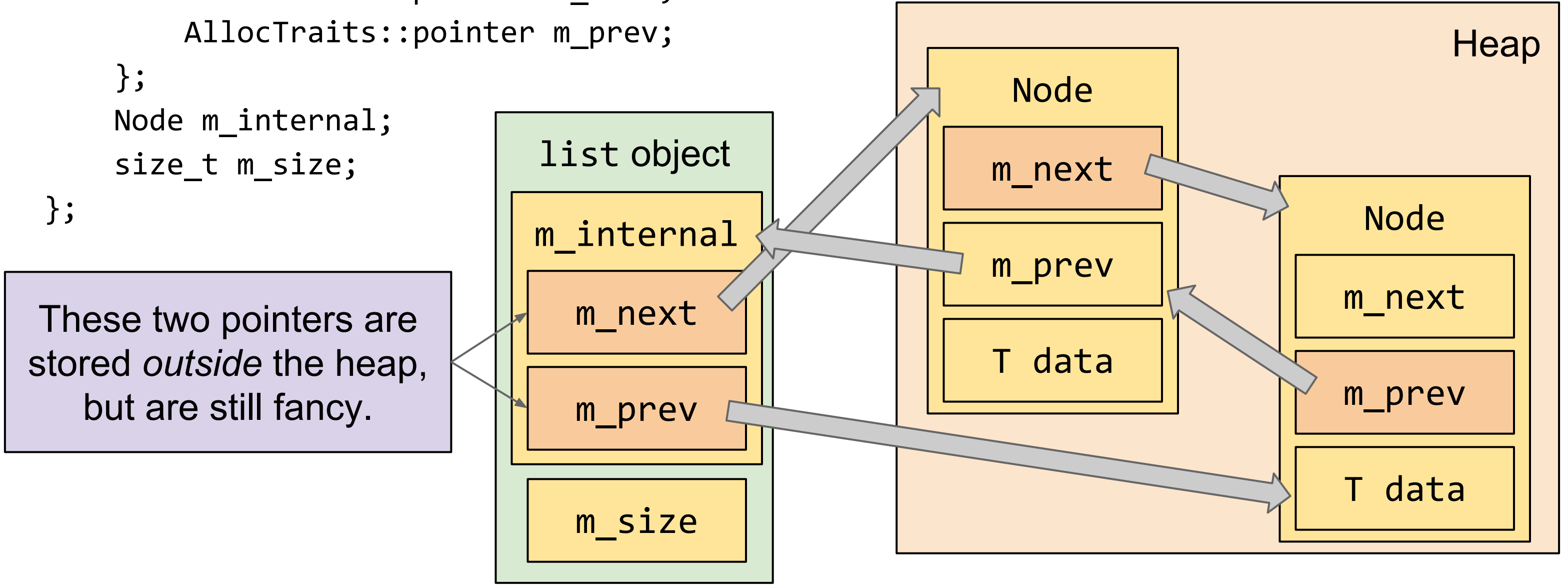
Container uses pointer for all allocations

```
template<class T, class A>
class list {
    struct Node {
        using AllocTraits = allocator_traits<A>::rebind_traits<Node>;
        AllocTraits::pointer m_next;
        AllocTraits::pointer m_prev;
    };
    Node m_internal;
    size_t m_size;
};
```



Container uses pointer for all allocations

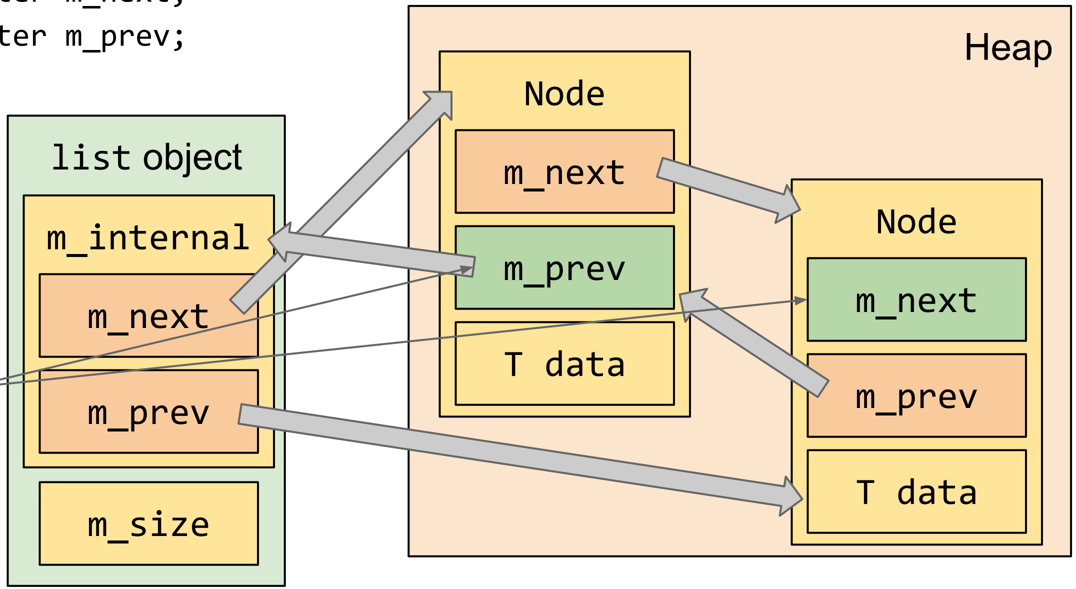
```
template<class T, class A>
class list {
    struct Node {
        using AllocTraits = allocator_traits<A>::rebind_traits<Node>;
        AllocTraits::pointer m_next;
        AllocTraits::pointer m_prev;
    };
    Node m_internal;
    size_t m_size;
};
```



Container uses pointer for all allocations

```
template<class T, class A>
class list {
    struct Node {
        using AllocTraits = allocator_traits<A>::rebind_traits<Node>;
        AllocTraits::pointer m_next;
        AllocTraits::pointer m_prev;
    };
    Node m_internal;
    size_t m_size;
};
```

These pointers are stored *within* the heap, but point to objects living outside the heap.

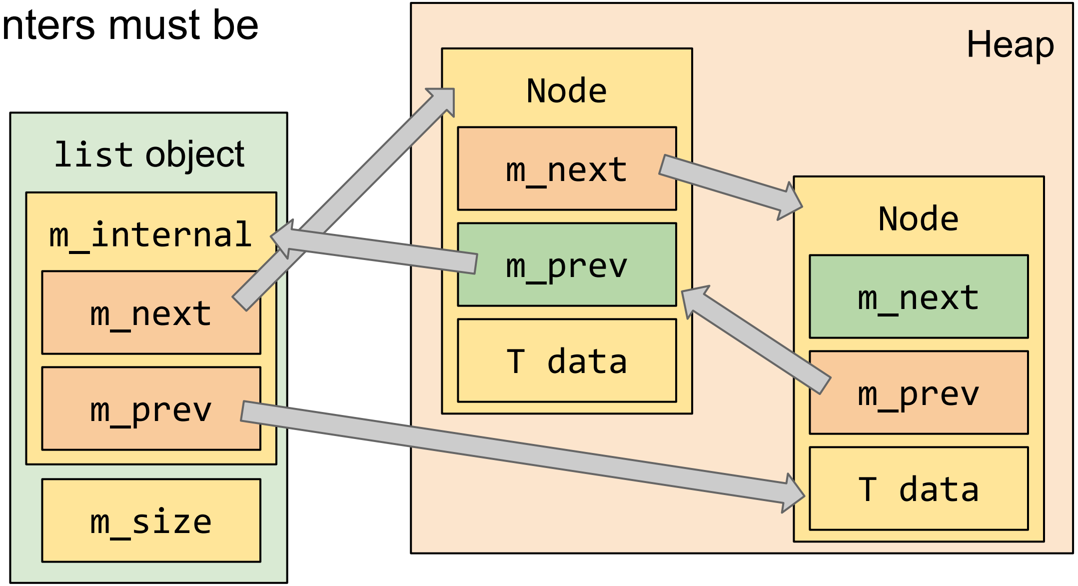


Fancy pointers' range = raw pointers' range

We must be able to convert fancy pointer m_next into native reference `*m_next`, and then into native pointer `this`.

We must be able to convert native pointer `&m_internal` into fancy pointer `m_prev`.

So fancy and native pointers must be ***interconvertible*** (must have the same range of values).



So are fancy pointers just native pointers?

"Interconvertible" = same possible values. So are they the same type?

So are fancy pointers just native pointers?

“Interconvertible” = same possible values. So are they the same type?

- No, because C++ type also involves object representation.
  - Boost `offset_ptr`
  - Bob Steagall’s “synthetic pointers”
  - C++ conflates valueish and objectish attributes: good idea or bad idea?

So are fancy pointers just native pointers?

“Interconvertible” = same possible values. So are they the same type?

- No, because the fancy type might be augmented with extra data.
  - “Segmented” pointers carry metadata (e.g. slab number) for the deallocator
  - “Fat” pointers carry metadata (e.g. array bounds) used during dereferences
  - Vendor extensions support these cases only inconsistently / accidentally. P0773R0 suggests that they should be supported by the Standard.

A C++ allocator is...

- Runtime source of memory (i.e., handle to a memory resource)

A C++ allocator is...

- Runtime source of memory (i.e., handle to a memory resource)
- Compile-time decider of the `pointer` type

A C++ allocator is...

- Runtime source of memory (i.e., handle to a memory resource)
- Compile-time decider of the `pointer` type
- Compile-time decider whether the source of memory should move with the container *value*, or stick to the original container *object* (POCCA, POCMA, POCS)
    - Bob calls this “lateral propagation”; I call it “stickiness”

A C++ allocator is...

- Runtime source of memory (i.e., handle to a memory resource)
- Compile-time decider of the `pointer` type
- Compile-time decider whether the source of memory should move with the container *value*, or stick to the original container *object* (POCCA, POCMA, POCS)
    - Bob calls this “lateral propagation”; I call it “stickiness”
- Runtime decider of how containers’ sub-objects (elements) should be constructed (`allocator_traits::construct`)
    - Bob calls this “vertical propagation”; I call it “`scoped_allocator_adaptor` is why we can’t have nice things”
    - Cf. `AT::has_trivial_construct_and_destroy` from “The Best Type Traits”

A handle to a heap *plus orthogonal pieces*

- Runtime source of memory (i.e., handle to a memory resource)
- Compile-time decider of the `pointer` type
- Compile-time decider whether the source of memory should move with the container *value*, or stick to the original container *object* (POCCA, POCMA, POCS)
    - Bob calls this “lateral propagation”; I call it “stickiness”
- Runtime decider of how containers’ sub-objects (elements) should be constructed (`allocator_traits::construct`)
    - Bob calls this “vertical propagation”; I call it “`scoped_allocator_adaptor` is why we can’t have nice things”
    - Cf. `AT::has_trivial_construct_and_destroy` from “The Best Type Traits”

We might separate some of these pieces

Imagine:

- nonsticky_allocator_adaptor (changing POCCA/POCMA without affecting source-of-memory)
- fancy_allocator_adaptor (changing pointer representation or fatness without affecting source-of-memory)
  - Boost.Interprocess has an allocator that deals in offset_ptr, but it gets its memory from a segment_manager. Getting offset_ptrs from an arbitrary heap wouldn't be useful here.
  - Grafting fat pointers onto an arbitrary heap sounds potentially useful.
- Should std::allocator be static_cast-able to std::pmr::polymorphic_allocator?

How are analogous “handle” types handled in other areas of C++?

Other “a Y is a handle to an X” in C++

An **allocator** is a cheaply copyable handle to a **memory resource**

- Plus some other bits, like `pointer` typedef and stickiness.

An **iterator** is a cheaply copyable handle to a container’s contents

- Plus some other bits, like iteration-direction (`reverse_iterator`).

Other “a Y is a handle to an X” in C++

An **allocator** is a cheaply copyable handle to a **memory resource**

- Plus some other bits, like `pointer` typedef and stickiness.

An **iterator** is a cheaply copyable handle to a container’s contents

- Plus some other bits, like iteration-direction (`reverse_iterator`).

> Boost has `iterator_facade` and `iterator_adaptor`.
>
> `iterator_facade` means “you implement a complete set of primitive functions (and data members); then inherit from `iterator_facade` to provide the standard zoo of operators.”
>
> `iterator_adaptor` means “you override some but not all the primitive functions; then inherit from `iterator_adaptor<Base>` to provide the missing parts.”

Other “a Y is a handle to an X” in C++

An **allocator** is a cheaply copyable handle to a **memory resource**

- Plus some other bits, like `pointer` typedef and stickiness.

An **iterator** is a cheaply copyable handle to a container’s contents

- Plus some other bits, like iteration-direction (`reverse_iterator`).

P0443: An **executor** is a cheaply copyable handle to an **execution context**

- Plus some other bits, like bulkness.

An executor is a handle (Executors TS)

P0443r5, also P0737r0

> The context_t property ... returns the **execution context** associated with the **executor**. An **execution context** is a program object that represents a specific collection of execution resources and the execution agents that exist within those resources. Execution agents are units of execution ...

P0443 proposes `static_thread_pool` as an example of an ExecutionContext.

P0443 std::executor is like std::function

P0443 proposes `std::executor` as an example of an Executor.

`std::executor` is “polymorphic” in that it can hold (a shared copy of) any kind of Executor, period. This is like `std::function` or `std::any`.

OTOH, `std::pmr::polymorphic_allocator` can hold merely (a non-owning pointer to) a concrete `std::pmr::memory_resource`, which is just one model of the MemoryResource concept.

“Truly polymorphic allocator”

```
template<class T>
class executor_style_allocator {
    shared_ptr<memory_resource> sptr;
public:
    using value_type = T;

    template<class Alloc, class = enable_if_t<...>>
    executor_style_allocator(Alloc a) :
        sptr(std::make_shared<resource_adaptor<Alloc>>(std::move(a))) {}

    T *allocate(size_t n) const {
        return (T*)sptr->allocate(n * sizeof(T), alignof(T));
    }
    void deallocate(T *p, size_t n) const {
        sptr->deallocate((void*)p, n * sizeof(T), alignof(T));
    }
};
```

“Truly polymorphic allocator”

```
template<class T>
class executor_style_allocator {
    shared_ptr<memory_resource> sptr;
public:
    using value_type = T;

    template<class Alloc, class = enable_if_t<...>>
    executor_style_allocator(Alloc a) :
        sptr(std::make_shared<resource_adaptor<Alloc>>(std::move(a))) {}

    T *allocate(size_t n) const {
        return (T*)sptr->allocate(n * sizeof(T), alignof(T));
    }
    void deallocate(T *p, size_t n) const {
        sptr->deallocate((void*)p, n * sizeof(T), alignof(T));
    }
};
```

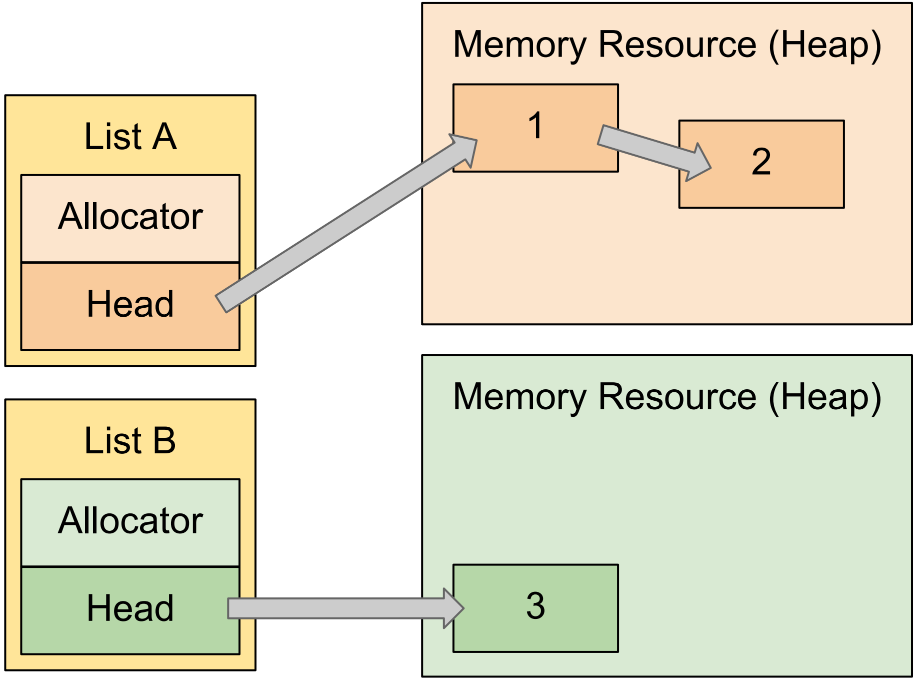
Lib Fundamentals TS, somehow omitted from C++17

Questions?

Peeve: swapping stateful allocators

Propagating and/or swapping *stateful* allocators is still broken in C++17; don't do it.

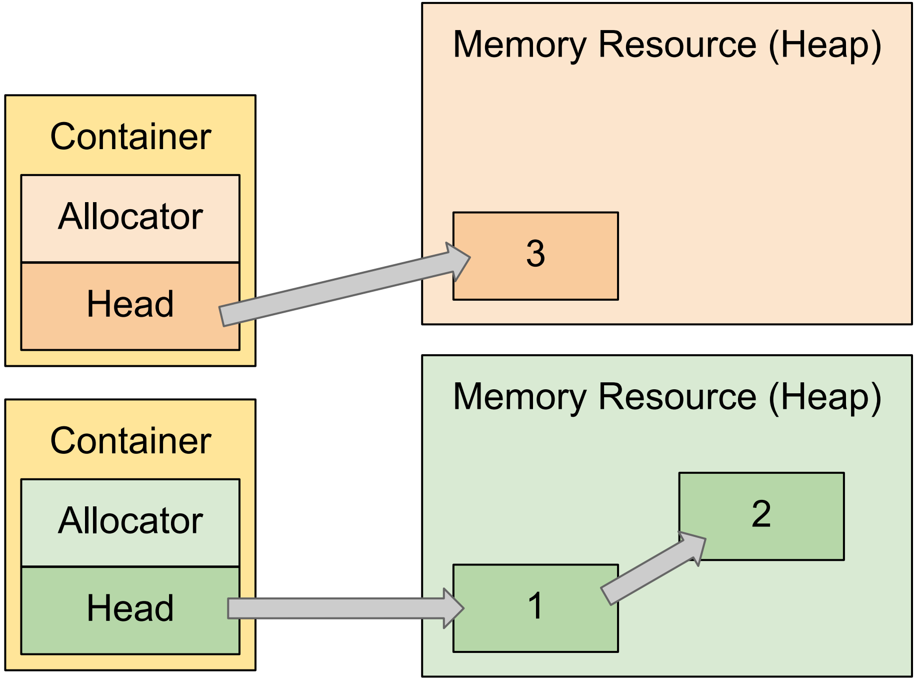
```
inline_buffer_resource<99> mr1;
inline_buffer_resource<99> mr2;
std::pmr::list<int> A{&mr1};
std::pmr::list<int> B{&mr2};
a.assign({1, 2});
b.assign({3});
swap(a, b);
```



Peeve: swapping stateful allocators

Propagating and/or swapping *stateful* allocators is still broken in C++17; don't do it.

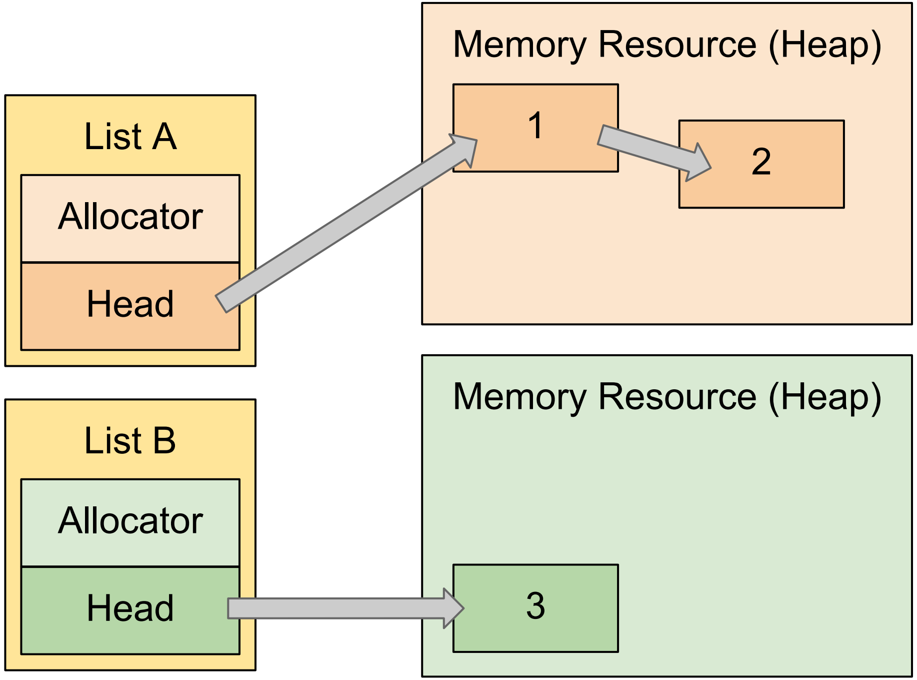
```
inline_buffer_resource<99> mr1;
inline_buffer_resource<99> mr2;
std::pmr::list<int> A{&mr1};
std::pmr::list<int> B{&mr2};
a.assign({1, 2});
b.assign({3});
swap(a, b);
```



Peeve: swapping stateful allocators

Propagating and/or swapping *stateful* allocators is still broken in C++17; don't do it.

```
inline_buffer_resource<99> mr1;
inline_buffer_resource<99> mr2;
std::pmr::list<int> A{&mr1};
std::pmr::list<int> B{&mr2};
a.assign({1, 2});
b.assign({3});
swap(a, b);
```



Peeve: swapping stateful allocators

Propagating and/or swapping *stateful* allocators is still broken in C++17; don't do it.

```
inline_buffer_resource<99> mr1;
inline_buffer_resource<99> mr2;
std::pmr::list<int> A{&mr1};
std::pmr::list<int> B{&mr2};
a.assign({1, 2});
b.assign({3});
swap(a, b);
```