memory_resource

An allocator is a handle to a heap

Arthur O’Dwyer
2017-07-26

C++17 adds `std::pmr::memory_resource`

- In the `<memory_resource>` header
- Currently not supported by any major vendor
  - not libc++ (LLVM/Clang)
  - not libstdc++ (GNU/GCC)
  - not Visual Studio (Microsoft/MSVC)
- Easy to implement yourself

And `std::pmr::polymorphic_allocator`

- Also in the `<memory_resource>` header
- Currently not supported by any major vendor
  - not libc++ (LLVM/Clang)
  - not libstdc++ (GNU/GCC)
  - not Visual Studio (Microsoft/MSVC)
- Not trivial, but still pretty easy to implement yourself

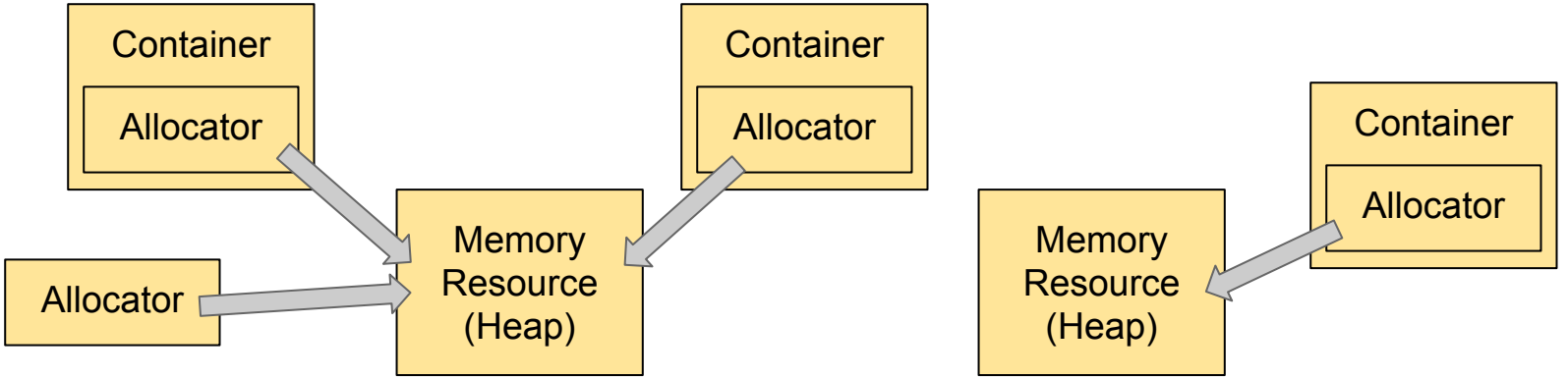
What are these new classes good for?

```
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);
}
```

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

   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(); }

   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)); }
   memory_resource *resource() const { return m_mr; }

   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(); }
```

Clarifies our thinking about allocators

- Old-style thinking: “an allocator represents a source of memory” — ***WRONG!***
- New-style thinking: “an allocator represents *a pointer to* a source of memory (plus some orthogonal bits).”



But what about stateless allocators?

A stateless allocator [e.g. `std::allocator<T>`] represents a pointer to a source of memory (plus some orthogonal bits) where that 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.

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;
}
```

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.
  - Might it ever make sense to *reference-count* a heap?
- Memory-resource types should generally be immobile.
  - A memory resource might allocate chunks out of a buffer stored inside itself as a data member.

Non-standard in-line buffer resource

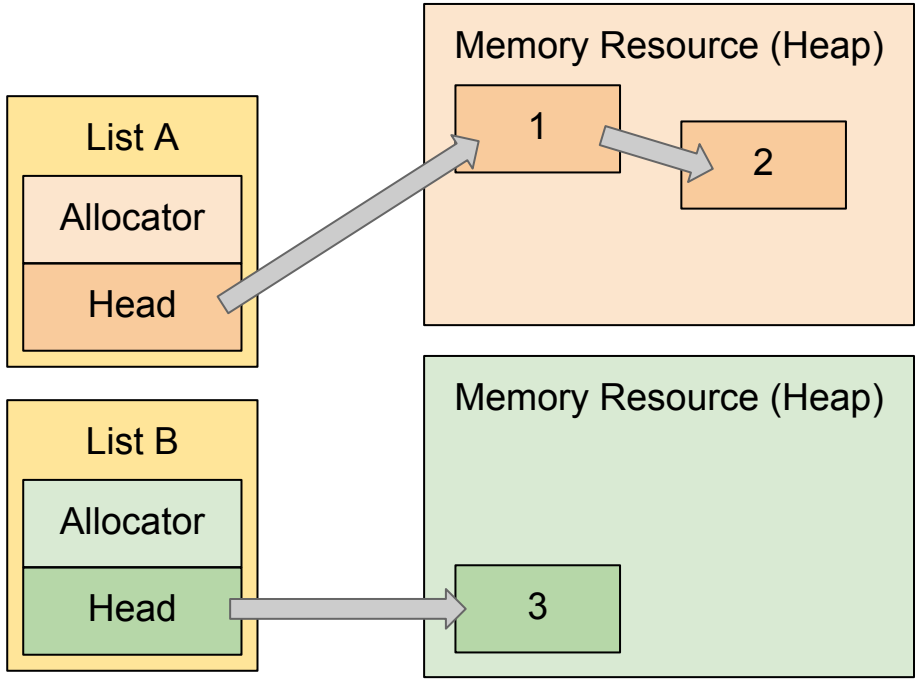
```
template<size_t capacity, size_t maxalign>
class inline_buffer_resource : public memory_resource {
    alignas(maxalign) char buffer[capacity];
    size_t index = 0;

    void *do_allocate(size_t bytes, size_t align) override {
        if (align > maxalign) throw bad_alloc();
        if ((-index % align) > (capacity - index)) throw bad_alloc();
        if (bytes > (capacity - index - (-index % align))) throw bad_alloc();
        index += (-index % align) + bytes;
        return buffer + (index - bytes);
    }

    void do_deallocate(void *, size_t, size_t) override {}
    bool do_is_equal(const memory_resource& rhs) const noexcept override {
        return (this == &rhs);
    }
};
```

(what we'd like to happen)

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);
```

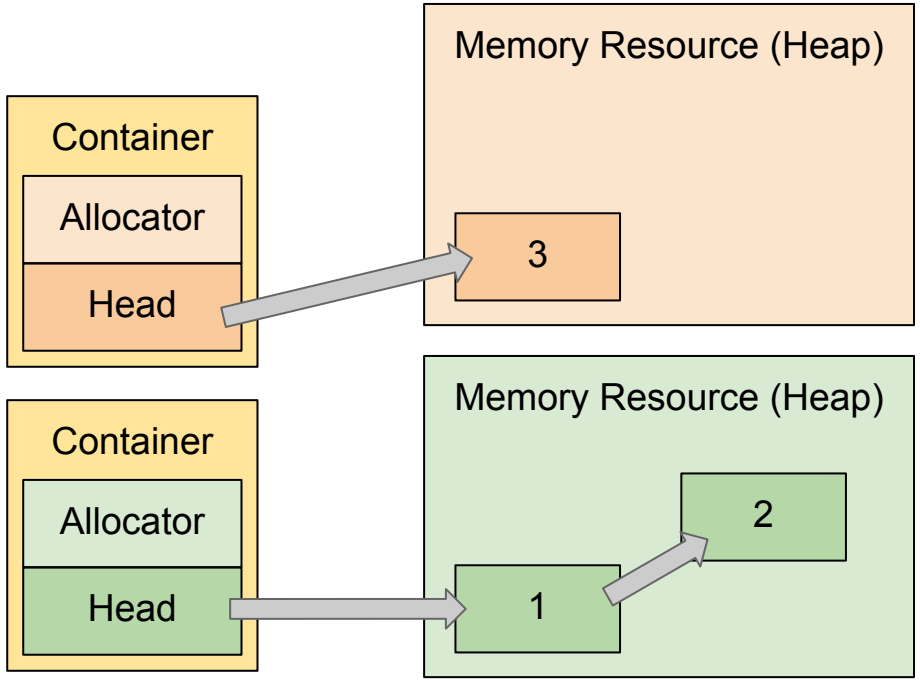


(what we'd like to happen)

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);
```

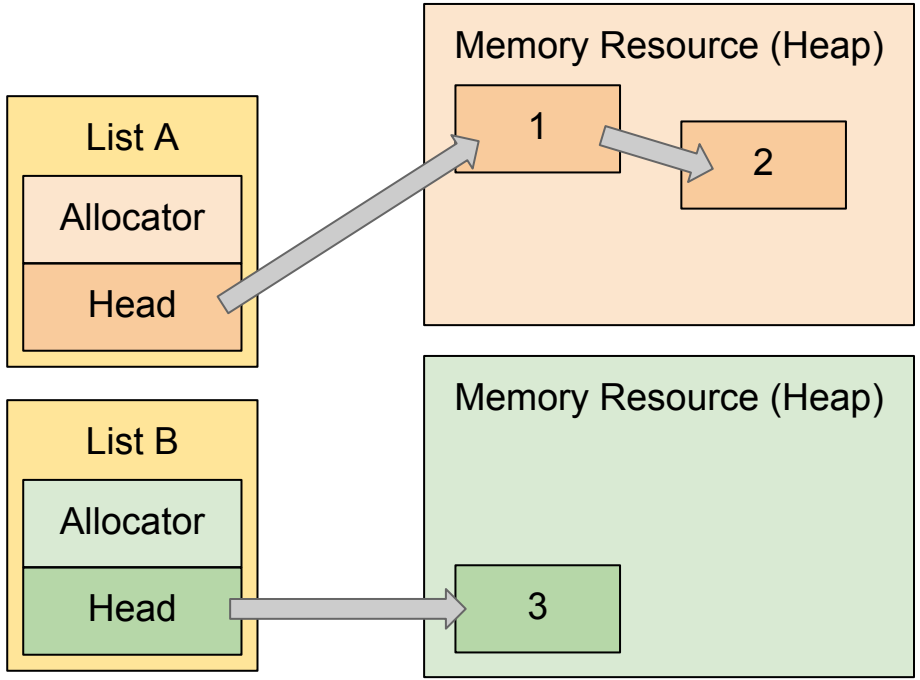


(what actually happens)

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);
```

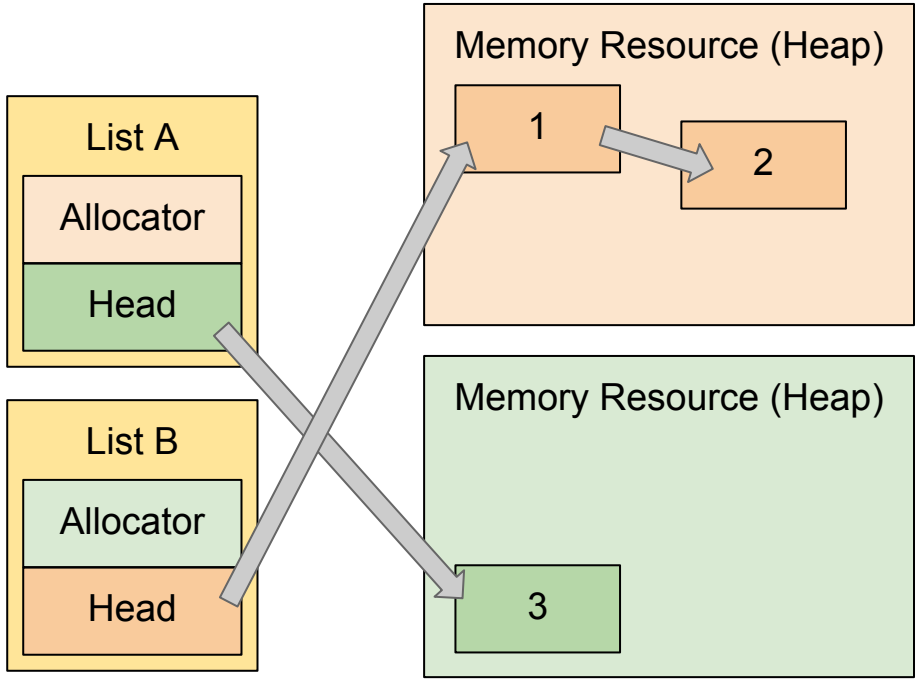


(what actually happens)

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);
```



Play with the new C++17 features

(and a lot of non-standard improvements)

https://github.com/Quuxplusone/from-scratch