Frictionless Allocators

Alisdair Meredith
Frictionless Allocators

ACCU Conference
March 2021

Alisdair Meredith
Senior Developer
Twitter @AlisdairMered
Outline of Talk

• Why Allocators?
• What do allocators look like in C++20?
• What causes friction?
• What should we do about it?
• What is our experience?
• What next?
What is an allocator?
What is an allocator?

A service that grants exclusive use of a region of memory to clients.
What is an allocator?

A service that grants exclusive use of a region of memory to clients

Nice clients will return that region to the service when no longer needed
Why Do We Want Allocators?
Why Do We Want Allocators?

Is the new operator not good enough?
Why Do We Want Allocators?

- Performance
- Performance
- Performance
- Instrumentation
- Special memory
Performance

• Well chosen allocators can greatly improve memory locality

  • See John Lakos talk: Value Proposition: Allocator Aware Software
    https://www.youtube.com/watch?v=ebn1C-mTFVk

  • Papers we wrote on measuring allocator performance:

  • Factor of 3-5 speedup common for allocation behavior, compared to new operator

  • Order of magnitude or more for extreme cases
Performance

• Well chosen allocators can greatly improve memory locality

• See John Lakos talk:
  Value Proposition: Allocator Aware Software
  https://www.youtube.com/watch?v=ebn1C-mTFVk

• Papers we wrote on measuring allocator performance:

• Factor of 3-5 speedup common for allocation behavior, compared to new operator

• Order of magnitude or more for extreme cases
A Faster Allocator

• General purpose allocator tries to minimize contention
  • A better/replacement `operator new`
• Avoid synchronization if we can guarantee all access from a single thread
• Simplified bookkeeping if we never reclaim memory
  • monotonic allocator simply advances a pointer through a buffer on each allocation
Performance

• Better memory locality improves runtime after allocation

• Keeping memory in L1/L2 cache has an enormous impact on runtime performance

  • although CPU is trying to manage cache to make this happen anyway, a local memory pool goes a long way to help

• Memory pools minimize the effect of diffusion on a single task

• Memory pools on the stack reduce fragmentation of long running processes

• No synchronization if allocation confined to a single thread
Performance

• Two common strategies to improve locality
  • Try to allocate on the thread stack
    • Typically from a pre-sized memory buffer
  • Manage a pool of memory to avoid needless trips back to the memory manager
    • This is commonly the implementation strategy of `operator new`, but specific pool for each data structure
Utility

- Custom allocators can add extra functionality in addition to supplying memory, such as instrumentation for:
  - Debugging
  - Logging
  - Profiling
  - Test drivers
Special Memory

• Special memory may be hardware specific, or has some other property, e.g., shared memory
  
  • Often requires a handle with more info than a native C++ pointer
    
    • e.g., boost::interprocess for shared memory containers
  
  • Some architectures provide different access to different regions of memory
    
    • VRAM on video cards?
Emery Berger

- Professor at University of Massachusetts Amherst
  
  - ACM Fellow: *For contributions in memory management and programming language implementation*
  
- Developed **Hoard**, first scalable general-purpose memory allocator
  
  - Algorithm incorporated into IBM & Mac OS X allocators
  
- Developed **DieHard** & **DieHarder** - reliable & secure memory allocators
  
  - Directly influenced Windows 7-8 allocator design
  
- Wrote technical paper evaluating custom allocators
  
  - *Reconsidering Custom Memory Allocation*, cited over 200 times, Most Influential Paper
Accelerating Programs via Custom Allocators

- **Demonstrate value** of custom allocation
  - Empirically measure opportunities (past successes)
    - Performance impact, space impact
- **Help programmers**
  - **Automatically** identify opportunities for custom allocation in legacy code
  - Provide tools to ensure **efficiency** for programs using custom allocation
Initial empirical results

• Replace specific custom allocator: **BufferedSequentialAllocator**

• Run with three benchmarks:

  • **stresstest** – artificial composition of sorts & filters; changes sort & filter criteria to make the engine rearrange static data based on random numbers

  • **moda_moda** – extraction of the DataLayer pipeline of some application that actually drives headline Terminal functions

  • **Single-stage**: benchmarks worksheet computation in a single stage pipeline. Exercises **WorksheetView** (Excel formulas)

  • **Multi-stage**: benchmarks worksheet computation in multiple stages composed by join views. Exercises **WorksheetView**, **UniqueView**, and **JoinView**
Early Results

-O3, TurboBoost on, replacing BufferManager

Roughly ~25% improvement using custom allocation
Allocators in C++20
template <class Alloc>
struct allocator_traits {
    using allocator_type = Alloc;
    using value_type = typename Alloc::value_type;
    using pointer = see below;
    using const_pointer = see below;
    using void_pointer = see below;
    using const_void_pointer = see below;
    using difference_type = see below;
    using size_type = see below;

    // ...
};
template <class Alloc>
struct allocator_traits {
  // ...

  template <class T>
  using rebind_alloc = see below;

  template <class T>
  using rebind_traits = allocator_traits<rebind_alloc<T>>;

  // ...
};
template <class Alloc>
struct allocator_traits {
  // ...
  static pointer allocate(Alloc& a, size_type n);
  static pointer allocate(Alloc& a, size_type n, const_void_pointer hint);
  static void deallocate(Alloc& a, pointer p, size_type n);

  template <class T, class... Args>
  static void construct(Alloc& a, T* p, Args&&... args);

  template <class T>
  static void destroy(Alloc& a, T* p);

  static size_type max_size(const Alloc& a);

  // ...
};
Allocator Traits
(since 2011)

template <class Alloc>
struct allocator_traits {
   // ...
   static pointer allocate(Alloc& a, size_type n);
   static pointer allocate(Alloc& a, size_type n, const_void_pointer hint);
   static void deallocate(Alloc& a, pointer p, size_type n);

   template <class T, class... Args>
   static void construct(Alloc& a, T* p, Args&&... args);

   template <class T>
   static void destroy(Alloc& a, T* p);

   static size_type max_size(const Alloc& a);

   // ...
};
template <class Alloc>
struct allocator_traits {
  // ...

  using propagate_on_container_copy_assignment = see below;
  using propagate_on_container_move_assignment = see below;
  using propagate_on_container_swap = see below;

  static Alloc select_on_container_copy_construction(const Alloc& rhs);
};
How did we improve support in C++17

• `is_always_equal`
  • Improved exception specifications on containers

• Polymorphic Memory Resources in namespace `std::pmr`

• `std::pmr` containers

• Constraint: fancy pointers must be *contiguous* iterators
std::allocator

C++17

template<class T>
struct allocator {
    using value_type = T;
    using size_type  = size_t;
    using difference_type = ptrdiff_t;
    using propagate_on_container_move_assignment = true_type;
    using is_always_equal = true_type;

    allocator() noexcept;
    allocator(const allocator&) noexcept;
    template<class U>
    allocator(const allocator<U>&) noexcept;
    ~allocator();
    allocator& operator=(const allocator&) = default;

    T* allocate(size_t n);
    void deallocate(T* p, size_t n);
};
template<class T>
struct allocator {
    using value_type = T;
    using size_type = size_t;
    using difference_type = ptrdiff_t;
    using propagate_on_container_move_assignment = true_type;
    using is_always_equal = true_type;

cnstexpr allocator() noexcept;
    cnstexpr allocator(const allocator&) noexcept;
    template<class U>
    cnstexpr allocator(const allocator<U>&) noexcept;
    cnstexpr ~allocator();
    cnstexpr allocator& operator=(const allocator&) = default;

    [[nodiscard]] cnstexpr T* allocate(size_t n);
    cnstexpr void deallocate(T* p, size_t n);
};
How does pmr work?

- Resource derives from `std::pmr::memory_resource`
  - Override the pure abstract members
- Clients store pointer to memory resource “like a vtable”
- Resource pointer never propagates
  - Scope resource object with a longer lifetime than consumers
  - Data structures “guarantee” all elements using same resource
- Alias templates for all standard containers to use this new scheme
class memory_resource {
   // For exposition only
   static constexpr size_t max_align = alignof(max_align_t);

public:
   virtual ~memory_resource();

   void* allocate(size_t bytes, size_t alignment = max_align);
   void deallocate(void* p, size_t bytes, size_t alignment = max_align);

   bool is_equal(const memory_resource& other) const noexcept;

protected:
   virtual void* do_allocate(size_t bytes, size_t alignment) = 0;
   virtual void do_deallocate(void* p, size_t bytes, size_t alignment) = 0;

   virtual bool do_is_equal(const memory_resource& other) const noexcept = 0;
};
class memory_resource {
    // For exposition only
    static constexpr size_t max_align = alignof(max_align_t);

public:
    virtual ~memory_resource();

    void* allocate(size_t bytes, size_t alignment = max_align);
    void dealloc(void* p, size_t bytes, size_t alignment = max_align);

    bool is_equal(const memory_resource& other) const noexcept;

protected:
    virtual void* do_allocate(size_t bytes, size_t alignment) = 0;
    virtual void do_deallocate(void* p, size_t bytes, size_t alignment) = 0;

    virtual bool do_is_equal(const memory_resource& other) const noexcept = 0;
};
class memory_resource {
   // For exposition only
   static constexpr size_t max_align = alignof(max_align_t);

public:
   virtual ~memory_resource();

   void* allocate(size_t bytes, size_t alignment = max_align);
   void deallocate(void* p, size_t bytes, size_t alignment = max_align);

   bool is_equal(const memory_resource& other) const noexcept;

protected:
   virtual void* do_allocate(size_t bytes, size_t alignment) = 0;
   virtual void do_deallocate(void* p, size_t bytes, size_t alignment) = 0;

   virtual bool do_is_equal(const memory_resource& other) const noexcept = 0;
};
memory_resource

class memory_resource {
   // For exposition only
   static constexpr size_t max_align = alignof(max_align_t);

public:
   virtual ~memory_resource();

   void* allocate(size_t bytes, size_t alignment = max_align);
   void  deallocate(void* p, size_t bytes, size_t alignment = max_align);

   bool is_equal(const memory_resource& other) const noexcept;

protected:
   virtual void* do_allocate(size_t bytes, size_t alignment) = 0;
   virtual void  do_deallocate(void* p, size_t bytes, size_t alignment) = 0;

   virtual bool do_is_equal(const memory_resource& other) const noexcept = 0;
};
class memory_resource {
    // For exposition only
    static constexpr size_t max_align = alignof(max_align_t);

public:
    virtual ~memory_resource();

    void* allocate(size_t bytes, size_t alignment = max_align);
    void deallocate(void* p, size_t bytes, size_t alignment = max_align);

    bool is_equal(const memory_resource& other) const noexcept;

protected:
    virtual void* do_allocate(size_t bytes, size_t alignment) = 0;
    virtual void do_deallocate(void* p, size_t bytes, size_t alignment) = 0;

    virtual bool do_is_equal(const memory_resource& other) const noexcept = 0;
};
class memory_resource {
    // For exposition only
    static constexpr size_t max_align = alignof(max_align_t);

public:
    virtual ~memory_resource();

    void* allocate(size_t bytes, size_t alignment = max_align);
    void  deallocate(void* p, size_t bytes, size_t alignment = max_align);

    bool is_equal(const memory_resource& other) const noexcept;

protected:
    virtual void* do_allocate(size_t bytes, size_t alignment) = 0;
    virtual void  do_deallocate(void* p, size_t bytes, size_t alignment) = 0;

    virtual bool do_is_equal(const memory_resource& other) const noexcept = 0;
};
Standard Resources

- `memory_resource* new_delete_resource() noexcept;`
- `memory_resource* null_memory_resource() noexcept;`
- `memory_resource* get_default_resource() noexcept;`
Standard Resources

- `memory_resource* new_delete_resource() noexcept;`
- `memory_resource* null_memory_resource() noexcept;`
- `memory_resource* get_default_resource() noexcept;`
- `class monotonic_buffer_resource;`
- `class synchronized_pool_resource;`
- `class unsynchronized_pool_resource;`
Idiom and usage of `pmr`

- Memory resources are objects, typically scoped to a function, with a lifetime longer than their clients below them.

- Default, object, and global “allocators”
  - System-wide default used for all objects unless otherwise specified.
  - Use the default for function-scope objects within an allocator-aware class.
  - Use object allocator (supplied at construction) only (and always) for data that is part of the object data structure, that persists beyond the function call.
  - Use another “global allocator” for any object with static or thread-local storage duration, as may outlive the default resource after `main`. 
Idiom and usage of pmr

• Memory resources are objects, typically scoped to a function, with a lifetime longer than their clients below them

• Default, object, and global “allocators”
  • System-wide default used for all objects unless otherwise specified
  • Use the default for function-scope objects within an allocator-aware class
  • Use object allocator (supplied at construction) only (and always) for data that is part of the object data structure, that persists beyond the function call
  • Use another “global allocator” for any object with static or thread-local storage duration, as may outlive the default resource after main

• No support for fancy pointers
Quick Example

```cpp
pmr::string make_string(const char *s) {
    pmr::monotonic_resource res;
    pmr::string x(s, &res);
    return x;
}
```
Quick Example

```cpp
pmr::string make_string(const char *s) {
    pmr::monotonic_resource res;
    pmr::string x(s, &res);
    return x;
}
```

String `x` is declared after resource `res`, so C++ lifetimes should avoid dangling references.
Quick Example

```cpp
pmr::string make_string(const char *s) {
    pmr::monotonic_resource res;
    pmr::string x(s, &res);
    return x;   // potential RVO
}
```
Quick Example

```cpp
pmr::string make_string(const char *s) {
    pmr::monotonic_resource res;
    pmr::string x(s, &res);
    return x;    // C++17 guarantees RVO
};
```
Quick Example

```cpp
pmr::string make_string(const char *s) {
    pmr::monotonic_resource res;
    pmr::string x(s, &res);
    return {x};
};
```

Force creation of a temporary, using the default allocator
Scoped Allocator Model

- Simple idea: every element in the data structure uses the same allocator/memory-resource

- Class design: every member of the object graph (all bases and members) use the same allocator

- Key benefit: underpins performance when looking to avoid diffusion and fragmentation

- Important benefit: easy to guarantee allocator/resource has a longer lifetime than its clients

- Implication: allocators can never propagate, or else any swap or assignment could invalidate the whole system
Scoped Allocator Model

- Simple idea: every element in the data structure uses the same allocator/memory-resource
- Class design: every member of the object graph (all bases and members) use the same allocator
- Key benefit: underpins performance when looking to avoid diffusion and fragmentation
- Important benefit: easy to guarantee allocator/resource has a longer lifetime than its clients
- Implication: allocators can never propagate, or else any swap or assignment could invalidate the whole system
namespace std::pmr {
    template<class Tp = byte>
    class polymorphic_allocator {
        memory_resource* memory_rs;
    // exposition only
    public:
        using value_type = Tp;

        // 20.12.3.1, constructors
        polymorphic_allocator() noexcept;
        polymorphic_allocator(memory_resource* r);
        polymorphic_allocator(const polymorphic_allocator& other) = default;

        template<class U>
        polymorphic_allocator(const polymorphic_allocator<U>& other) noexcept;

        polymorphic_allocator& operator=(const polymorphic_allocator&) = delete;
    }
namespace std::pmr {
    template<class Tp = byte>
    class polymorphic_allocator {
        memory_resource* memory_rsrc;    // exposition only
    public:
        using value_type = Tp;

        // 20.12.3.1, constructors
        polymorphic_allocator() noexcept;
        polymorphic_allocator(memory_resource* r);
        polymorphic_allocator(const polymorphic_allocator& other) = default;
        template<class U>
        polymorphic_allocator(const polymorphic_allocator<U>& other) noexcept;

        polymorphic_allocator& operator=(const polymorphic_allocator&) = delete;

        // ...
    }
// ...  

// 20.12.3.2, member functions
[[nodiscard]] Tp* allocate(size_t n);
void deallocate(Tp* p, size_t n);

template<class T, class... Args>
  void construct(T* p, Args&&... args);

template<class T>
  void destroy(T* p);

polymorphic_allocator select_on_container_copy_construction() const;
memory_resource* resource() const;

// ...
// ...

[[nodiscard]] void* allocate_bytes(size_t nbytes,
        size_t alignment = alignof(max_align_t));
void deallocate_bytes(void* p, size_t nbytes,
        size_t alignment = alignof(max_align_t));

template<class T>
[[nodiscard]] T* allocate_object(size_t n = 1);
template<class T>
void deallocate_object(T* p, size_t n = 1);
template<class T, class... CtorArgs>
[[nodiscard]] T* new_object(CtorArgs&&... ctor_args);
template<class T>
void delete_object(T* p);
Limitations of pmr

• Solves the vocabulary problem, but only if used consistently (i.e., the vocabulary problem!)

• No support for fancy pointers / special memory regions

• Storing an extra pointer in every object, repeatedly through the whole data structure

• Cost of dynamic dispatch

  • (see Lakos talk for why this may be negligible)
Bloomberg Experience

- Using the progenitor for pmr allocators for a decade or more

- pmr style allocators can be a big win
  - Performance-critical code benefits significantly
  - Instrumentation helpful, especially in test drivers

- Users still bridle at code complexity
What Causes Friction with pmr Allocators?
Unsupported use cases

• Without user-supplied constructors, some types cannot support the scoped allocator model:

  • `pmr::string data[42];`
  • `std::array<pmr::string, 42> more_data;`

• Lambda objects

• Structured bindings

• Default member initializers

• Problem is recursive, `vector<array<string, 10>>`
Allocator Propagation

- Allocator is bound at construction
- Should allocator be rebound on assignment?
  - Assignment copies data
  - Allocator is orthogonal, specific to each container object
- Traits give control of the propagation strategy
  - Default is to never propagate
Complexity of Propagation

- 3 (or 4?) fine-grained traits is too many dimensions to reasonably support

- What does it mean to propagate on `swap`, but not on move assignment? Or vice-versa?

- Trait for copy construction is actually a function call?!
Syntactic overhead is high

• Mandatory construction through traits looks like expert-level code

• Typically double the number of constructors to allow for optional allocator
  
  • Cannot have a constructor with multiple defaulted arguments, as need an optional allocator for each subset

  • E.g., for `unordered_map`

    • C++11: 8 constructors

    • C++17: 15 constructors (many delegating to original 8)

• Inconsistent argument order: allocator is final argument, or `MyType(allocator_arg, alloc, ...)"
Copy Constructor Issue

- With the trait to select allocator on copies, behavior was unpredictable with optional copy-elision rules
- C++17 nails down mandatory copy elision in the important cases
- Risk of returning an object with a reference to a memory resource that is about to leave scope
- Compiler warnings may help in the future
Copy Constructor Issue

- With the trait to select allocator on copies, behavior was unpredictable with optional copy-elision rules

- C++17 nails down mandatory copy elision in the important cases

  - Risk of returning an object with a reference to a memory resource that is about to leave scope

- Compiler warnings may help in the future
Reducing Friction
Ideal Model

- No allocator spam in the interface
- A single data structure uses the same allocator throughout
  - e.g., container and its elements
  - e.g., a graph, its nodes, and their contents, etc.
- If a type manages dynamic memory, it always supports an allocator
- “allocator aware” types are known to the type system
  - can query if a type is allocator aware
  - can query which allocator an object uses
Allocate Awareness

A type is *explicitly* allocator aware if:
- it says so (need a way to mark a class)

A type is *implicitly* allocator aware if:
- it derives from an allocator-aware class
- it has data members that are allocator aware
  - “viral” on members as well as bases
Why Implicit from Members?

- Could just make allocator aware classes derive from a base class with the right behavior
  - Generic code would want conditional bases
  - Arrays, aggregates, etc., need implicit behavior
  - Forces extra vtable pointer in all cases
  - Too much syntax for common/essential usage
Why Implicit from Members?

- Could just make allocator aware classes derive from a base class with the right behavior

  - Generic code would want conditional bases

  - Arrays, aggregates, etc., need implicit behavior

- Forces extra vtable pointer in all cases

- Too much syntax for common/essential usage
Allocator Aware Properties

• An allocator-aware class will use its supplied allocator to acquire all memory for the data structure’s persistent needs

• There is a consistent (customizable) API to query which allocator an object is using

• The allocator for an object will not change during its lifetime

  • i.e., allocators do not propagate
Why querying matters

• Some operations require allocators to be the same, e.g., move and swap

• Make a temporary with the same allocator if we expect to move into an existing object

• swap should either have a precondition that allocators are the same, or make potentially throwing “copies” with appropriate allocator for swaps

• Users need a means to detect allocator compatibility if they are to avoid violating such preconditions
Simplifying Construction

• Do *not* add allocator overloads to every constructor

• When user wants to supply an allocator, pass it out-of-band from the initializer list with a new syntax

```cpp
multipool_resource res;
set<string>() x{ "hello", "world"} using res;
```
class Object {
    std::pmr::string d_name;

public:
    using allocator_type = std::pmr::polymorphic_allocator<>;

    explicit Object(allocator_type a = {}) : d_name("<UNKNOWN>", a) {}

    Object(const Object& rhs, allocator_type a = {}) : d_name(rhs.d_name, a) {}

    Object(Object&&) = default;
    Object(Object&& rhs, allocator_type a) : d_name(std::move(rhs.d_name), a) {}

    // Apply rule of 6
    ~Object() = default;
    Object& operator=(const Object& rhs) = default;
    Object& operator=(Object&& rhs) = default;
};
class Object {
    std::pmr2::string d_name;

public:
    // using allocator_type = std::pmr::polymorphic_allocator<>;

    Object() : d_name("<UNKNOWN>") {} // no longer explicit

    Object(const Object& rhs) = default;
    Object(Object&&) = default;
    // Object(Object&& rhs, allocator_type a);

    // Apply rule of 6
    ~Object() = default;
    Object& operator=(const Object& rhs) = default;
    Object& operator=(Object&& rhs) = default;
};
class Object {
    std::pmr2::string d_name = "<UNKNOWN>";

public:

    Object() = default;

    Object(const Object& rhs) = default;

    Object(Object&&) = default;

    // Apply rule of 6
    ~Object() = default;
    Object& operator=(const Object& rhs) = default;
    Object& operator=(Object&& rhs) = default;
};
class Object {
    std::pmr2::string d_name = "<UNKNOWN>";

public:

    // Rule of zero !!
};
class Object {
    std::pmr2::string d_name = "<UNKNOWN>";

public:

    // Rule of zero !!

};

pmr::multipool_resource res;
Object x{"Hello world"} using res;
Implementing Awareness

• Stash an allocator pointer at construction, much like a vtable pointer
  • does not vary through constructing a hierarchy though
• Customization API to give precise control of storage if needed
  • e.g., optional object needs to stash allocator when empty, but can re-use the storage for the missing object
• If awareness is *implicit*, access the allocator through the entity granting awareness
  • do not pay to store excess copies of the pointer
  • Leaf nodes of data structures will always need a pointer though
Implicit Awareness

• In most cases, allocator awareness will be implicit, greatly reducing the implementation cost for user code

• Implicit awareness clearly implies a new language feature

• Implicit awareness can be supported by C++11 containers, using an allocator-aware allocator object (i.e., the allocator template parameter)
Allocate Injection

- Inject an allocator at object creation time, in addition to constructor arguments
  - needs language support with an extension syntax (such as using)
  - new operator one obvious customization, but need local object support too
- An implicit extra argument for every constructor
  - no constructor spam with allocator overloads
  - process-wide default is provided if not supplied by the caller
    - note: the move constructor is special
- Implicitly propagate that injection through member initializers for all bases and members
  - but not into constructor body
Early Experience
Sean Baxter and Circle

- Sean Baxter has written his own C++20 compiler with an LLVM back-end over the last 3 1/2 years
- Designed for rapid prototyping and language evolution, ultimately to advance his own post-C++ language, Circle
Sean Baxter and Circle

• Sean Baxter has written his own C++20 compiler with an LLVM back-end over the last 3 1/2 years

• Designed for rapid prototyping and language evolution, ultimately to advance his own post-C++ language, Circle

• After seeing previous talks online, implemented allocator injection through using in around a week...
Injecting Allocators with *using*

- Works well on existing standard library
- Relies on existing allocator_type mark-up, and existing allocator-aware constructors
  - No implicit generation (yet)
- CTAD support fell out for free
  (Constructor Template Argument Deduction)
- Supports full C++20 allocator model, *not* limited to pmr
// allocator-specifier in initializers and postfix-expressions

#include <list>
#include "logger.hxx"

int main() {
    logging_resource_t logger("logger");

    // using-allocator in a braced initializer for a declaration.
    // creates a PMR list when the allocator expression derives
    // memory_resource.
    pmr::list<int> my_list { 1, 2, 3 } using logger;

    // using-allocator in a braced initializer on an expression.
    auto my_list2 = pmr::list<int> { 4.4, 5.5, 6.6 } using logger;
}
Factory Functions

• Functions that return a new object by value
  • e.g., `std::make_shared`, `std::to_string`

• How should we provide an allocator for the return value?
  • Pass an extra function argument?
  • Add using support?
Factory Functions

• Functions that return a new object by value
  • e.g., `std::make_shared`, `std::to_string`

• How should we provide an allocator for the return value?
  • Pass an extra function argument
  • **Add using support**
Factory Functions

- Functions that return a new object by value
  - e.g., `std::make_shared`, `std::to_string`
  - How should we provide an allocator for the return value?

- Add using support (explicit for overloading, or implicit)
Factory Functions

• Functions that return a new object by value

• e.g., `std::make_shared`, `std::to_string`

• How should we provide an allocator for the return value?

• Add using support

• How do we avoid redundant allocations with default allocator inside the factory?
The Conundrum

What should we do with an extended move constructor?

template <class T>
struct NamedValue {
    std::string name;
    T value;

    NamedValue(NamedValue&&) = default;
    NamedValue(NamedValue&&) using Alloc = default;
};
Option 1

• Member initializers call `extended-move-with-allocator` for each base and member

• The classes that handle memory allocation directly (i.e., vector, rather than class with a vector member) will implement the custom logic to test the allocator, and move or make copies as needed

• Supports move-only types as members, as long as they manage any extended-move logic themselves, which is trivial by default for non-allocator aware types
Option 2

• Test whether the allocators are compatible

• If compatible, delegate directly to the regular move constructor

• If incompatible, delegate to the extended copy constructor

• Ill-formed if the extended copy constructor is not available

• (ideally) provide custom overload to handle non-default cases.
Option 2

• Test whether the allocators are compatible

• If compatible, delegate directly to the regular move constructor

• If incompatible, delegate to the extended copy constructor

• Ill-formed if the extended copy constructor is not available

• (ideally) provide custom overload to handle non-default cases. [we will need more syntax]
struct highlight {
    std2::vector<int> d_values;
    int * d_focus;  // invariant: points to an element in d_values

    highlight(highlight const &) = delete;
    highlight(highlight&&)       = default;
};
struct highlight {
    std2::vector<int> d_values;
    int * d_focus;  // invariant: points to an element in d_values

    highlight(highlight const &) = delete;
    highlight(highlight&&)       = default;
};

Copying d_focus would violate invariant so delete the copy constructor. Alternatively, implement with a look-up to fix up pointer to the corresponding element.
struct highlight {
    std2::vector<int> d_values;
    int * d_focus; // invariant: points to an element in d_values

    highlight(highlight const &); // user provided
    highlight(highlight&&) = default;
};

Copying d_focus would violate invariant unless allocators match. Option 2 is the only safe default: delegate to copy if allocators do not match, and using construction (but not regular move) is deleted if copy constructor is deleted.
struct highlight {
    std2::vector<int> d_values;
    int * d_focus; // invariant: points to an element in d_values

    highlight(highlight const &other) :
        d_values(other.d_values),
        d_focus(nullptr)
    {
        // maybe find new address for d_focus in d_values
    }

    highlight(highlight&& other) = default;

    highlight(highlight&& other) [[?]] // magical using overload
        : highlight(copy_or_move(other)) {} // delegating constructor

private:
    // factory function
    static highlight copy_or_move(highlight& other) {
        if (allocator_of(other) == allocator_of(copy_or_move)) {
            return std::move(other);
        }
        else {
            return other;
        }
    }
};
struct highlight {
    std2::vector<int> d_values;
    int * d_focus; // invariant: points to an element in d_values

    highlight(highlight const & other)
        : d_values(other.d_values),
          d_focus(nullptr)
    {
        // maybe find new address for d_focus in d_values
    }

    highlight(highlight&& other) = default;

    highlight(highlight&& other) [[?]] // magical using overload
        : highlight(copy_or_move(other)) {} // delegating constructor

private:
    // factory function
    static highlight copy_or_move(highlight& other) {
        if (allocator_of(other) == allocator_of(copy_or_move)) {
            return std::move(other);
        }
        else {
            return other;
        }
    }
};
struct highlight {
    std2::vector<int> d_values;
    int * d_focus; // invariant: points to an element in d_values

    highlight(highlight const &other)
    : d_values(other.d_values), d_focus(nullptr)
    {
        // maybe find new address for d_focus in d_values
    }

    highlight(highlight&& other) = default;

    highlight(highlight&& other) noexcept // magical using overload
    : highlight(copy_or_move(other)) {} // delegating constructor

private:
    // factory function
    static highlight copy_or_move(highlight& other) {
        if (allocator_of(other) == allocator_of(copy_or_move)) {
            return std::move(other);
        }
        else {
            return other;
        }
    }
};
struct highlight {
    std2::vector<int> d_values;
    int * d_focus;  // invariant: points to an element in d_values

    highlight(highlight const &other)
        : d_values(other.d_values)
          , d_focus(nullptr)
    {
        // maybe find new address for d_focus in d_values
    }

    highlight(highlight&& other) = default;

    highlight(highlight&& other) [[?]]  // magical using overload
        : highlight(copy_or_move(other)) {} // delegating constructor

private:
    // factory function
    static highlight copy_or_move(highlight& other) {
        if (allocator_of(other) == allocator_of(copy_or_move)) {
            return std::move(other);
        } else {
            return other;
        }
    }
};
Next Step
Basic Feature Set

- Pass allocators to *factory* functions and initializers through extra *using* argument

- A special allocator type that imbues enclosing classes as allocator aware
  - Fundamental type to avoid specifying customization interface
  - Acts like a reference to a `pmr::memory_resource`

- *All* constructors of allocator aware type are implicitly allocator aware
  - Move constructor is special and split in two
  - `allocator_of` implicit hidden friend function

- *Implicit* factory functions (only)
Implicit Factory Functions

- Support a using when return type is allocator aware
  - Will explore what it means when calling from templates
- Implicitly supply allocator to return value
- Where guaranteed (N)RVO applies, supply allocator to variable declarations
  - P2025 Guaranteed Copy Elision for Named Return Objects
Validation Tests

- `std2::string`
- `std2::vector<T>`
- `std2::vector<std2::string>`
- aggregate classes
- local arrays
- `std::array`
- `std::tuple`
- `std::pair`
- lambda expressions
Open Questions

• Argument passing in factory functions
• Explicit factory functions
• Overloading constructors on allocator
• More customization points
  • e.g. overloading allocator_of to avoid redundant copy in optional
• using in more places, such as function arguments?
• Unions and variant
Fin