Getting Allocators Out Of Our Way
Language support for scoped allocators

P3240 presented to EWGI in Tokyo
March 21, 2024

Alisdair Meredith
ameredith1@bloomberg.net

TechAtBloomberg.com
Presentation Goals

• Seek feedback on the scope of a proposal that would best progress work in this group
  • Do we need a complete solution to all known issues?
  • Should we take an MVP approach like the contract work?

• Order of presentation
  • Motivate the problems to be solved
  • Present our current understanding and goals for language support for the scoped allocator model
  • Present known design questions that are left open pending feedback
Why Allocators Matter

Motivation

• Memory is a special resource consumed by every object in the system

• Memory access patterns (locality of reference) can be a critical factor of system performance, and control of memory allocation is our best known way to handle that

• Long lived applications suffer from memory fragmentation and diffusion without careful control of memory allocation

• Additional utility in the form of telemetry, support for testing, etc.
Why Allocators Are Not Used

Demotivation

• Library support is very intrusive
• Is not an optional part of the design
  • Must be integrated from the start
  • Hard to retro-fit
• Cannot support all types
  • Aggregates, arrays, lambas, …
Simplifying the Problem
Building on experience

- Building on `pmr` memory resources
  - Building on Bloomberg experience beyond the standard library
  - Preferring std library in our examples for familiar vocabulary
- Looking to generalize in the future
  - Extensions to support non-memory resource allocators
  - Extensions to support non-allocator protocols
What is the Scoped Allocator Model

- The scoped allocator model supports enforcing the same allocator is used for all members of the same data structure, notably for containers such as `vector` and `map`
  - i.e., all elements of the container use the same allocator as the container
  - This is the model used by `pmr::polymorphic_allocator`
What is Allocator Propagation

• A container is given an allocator at construction, and that allocator never changes

• In particular, it is not replaced by assignment or swap

• Propagate is a confusing term — we do not propagate the allocator through assignment and swap to objects outside the container, but do push the allocator to every element inside the container, and that sounds a lot like a different form of propagation
Allocator for Construction in pmr Model

- If no allocator is explicitly supplied, use the `default_memory_resource`, even for copies and temporaries

- Unless it is the specific special case of the move constructor
Problems to Solve for Users of \texttt{pmr}

Current state of the art

- Cannot reach all parts of the language
  - Aggregates
  - Arrays (technically an aggregate)
  - Lambdas
- Objects with static storage duration require special attention
Problems to Solve for Library Implementers

Current state of the art

- Implementation and maintenance of the scoped semantic is expensive
  - Many constructor overloads requiring an allocator argument
  - Must pay careful attention to non-propagation of the allocator
  - Finding the allocator an object uses needs a convention not described by the standard allocator traits
Towards a Solution
Related Work

Papers that are assumed as they solve related problems

- P2025 Guaranteed NRVO (EWG, paper stalled)
- P2786 Trivial relocation (passed EWG this meeting)
- P2959 Relocation within a container (LEWG, not yet seen)
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;
};
Worked Example

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 !!
Worked Example

class Object {
   std::pmr2::string d_name = "<UNKNOWN>";

public:

   // Rule of zero !!

};

pmr::multipool_resource res;
Object x{"Hello world"} using res;
Supporting Language Constrained Types

• Type is *allocator enabled* if it has any allocator-enabled bases or non-static data members
  • New fundamental type provides basic hook to be allocator enabled
  • New type acts like `pmr::memory_resource&`

• Allocator propagation cannot depend on user provided functions
  • Propagation rules must be implicit and implemented by the compiler
  • Natural behavior when the new type behaves like a reference — does not rebind
Supplying an Allocator

- Allocators must be supplied my a mechanism that is not a constructor argument
  - Addresses getting allocators into aggregates, arrays, and lambdas
- Suggested syntax: using after variable initializers
  - Using-initialization supported only for allocator-enabled types
  - Not usable with member initializers, as class must have consistent allocator
  - Uses the default memory resource if not supplied by user, but...
    - See later for initializing objects with static storage duration
Aggregates do not support `pmr`  
Correct-looking usage does not propagate allocator to strings

```
struct Aggregate {
    std::pmr::string data1;
    std::pmr::string data2;
    std::pmr::string data3;
};

std::pmr::test_resource tr;
std::pmr::polymorphic_allocator ta(&tr);
Aggregate ag = {"Hello", ta}, {"World", ta}, {"!", ta};

std::pmr::vector<Aggregate> va(ta);
va.emplace_back(std::move(ag));  // Correct allocator is retained by moves
va.emplace_back(ag);             // Error, copied lvalue uses default resource
va.resize(5);                    // Error, new elements use default resource
va.resize(1);                    // OK, remove all objects with bad allocators
```
Aggregate Support becomes Implicit
Simpler syntax, and behaves correctly

```cpp
struct Aggregate {
    std2::string data1;
    std2::string data2;
    std2::string data3;
};

std::pmr::test_resource tr;
Agggregate ag using tr = {"Hello", "World", "!"};

std2::vector<Aggregate> va using tr;
va.emplace_back(std::move(ag)); // Correct allocator is retained by moves
va.emplace_back(ag); // Scoped allocator is applied to copied element
va.resize(5); // All elements use scoped allocator
va.resize(1); // OK
```
Exposing the Allocator

- All allocator enabled objects have a “hidden friend” `allocator_of` function
  - Returns a reference to the memory resource used by the object
  - Allows testing for whether two objects have the same allocator
  - Call `allocator_of(*this)` to find your own allocator
- Implicit implementation looks for first allocator-enabled member (including base member objects) and forwards the call
  - This implicit implementation will resolve support for native arrays
int main() {
    using namespace std;
    pmr::monotonic_buffer_resource tr;

    pair<pmr::string, pmr::string> p2 = {
        piecewise_construct
        , tuple{pmr::string("Hello", &tr)}
        , tuple{pmr::string("world", &tr)}
    };

    tuple t4 = {
        allocator_arg, pmr::polymorphic_allocator<>{&tr}
        , pmr::string("Bonjour")
        , pmr::string("tout")
        , pmr::string("le")
        , pmr::string("mond")
    };

    // assert(p2.get_allocator() == &tr);    // No equivalent
    // assert(t4.get_allocator() == &tr);    // No equivalent

    assert(get<0>(p2).get_allocator() == &tr);
    assert(get<1>(p2).get_allocator() == &tr);

    assert(get<0>(t4).get_allocator() == &tr);
    assert(get<1>(t4).get_allocator() == &tr);
    assert(get<2>(t4).get_allocator() == &tr);
    assert(get<3>(t4).get_allocator() == &tr);
}
Easy to Extract Allocator, Even From Existing Templates

```cpp
int main() {
    using namespace std2::string_literals;
    std2::test_resource tr;

    std::pair p2 using tr = { "Hello"s, "world"s };   
    std::tuple t4 using tr = { "Bonjour"s, "tout"s, "le"s, "mond"s };

    assert(allocator_of(p2) == tr);  
    assert(allocator_of(t4) == tr);

    assert(allocator_of(get<0>(p2)) == tr);  
    assert(allocator_of(get<1>(p2)) == tr);

    assert(allocator_of(get<0>(t4)) == tr);  
    assert(allocator_of(get<1>(t4)) == tr);  
    assert(allocator_of(get<2>(t4)) == tr);  
    assert(allocator_of(get<3>(t4)) == tr);
}
```
Factory Functions
Passing allocators for the return value

• A *factory function* is any function that returns an allocator-enabled object by value

• Factory functions support a using argument to supply an allocator

• Return expressions implicitly use the allocator supplied to the function

• Local variables that are guaranteed to RVO implicitly use the supplied allocator

  • Hence desire for the proposal for some NRVO guarantees
std2::string make(char const * s) { return s; }

std2::string join(char const * s1, char const * s2) {
    using std2::string;
    return string{s1} + string{" "} + string{s2};
}

std2::string join2(std2::string s1, std2::string s2) {
    return s1 + " " + s2;
}

int main() {
    std::pmr::test_resource ta;
    auto hw = make("Hello world!") using ta;
    hw = join("Hello", "world!") using ta;

    std2::string hello using ta = "Hello";
    std2::string world using ta = "world";

    hw = join2(hello, world) using allocator_of(hw);  // temporaries use pa
A Generic Factory Function

Missing standardese is at least another 10 slides to show…

// make_from_tuple is 1/2 page of C++23 specification
// uses_allocator_construction is 2 1/2 pages of C++23 specification

template<class T, class Alloc, class... Args>
constexpr
T make_obj_using_allocator(const Alloc& alloc, Args&&... args) {
    return make_from_tuple<T>(uses_allocator_construction_args<T>(
        alloc, std::forward<Args>(args)...));
}
Simplified Generic Factory Function

// make_from_tuple is 1/2 page of C++23 specification
// uses_allocator_construction is 2 1/2 pages of C++23 specification

template<class T, class Alloc, class... Args>
constexpr
T make_obj_using_allocator(const Alloc& alloc, Args&&... args) {
    return {std::forward<Args>(args)...};
}
Move Semantics

- Allocators do not propagate on move-assignment, as we do not rebind/replace an existing allocators

- Allocators do propagate on move-construction or else moves would become allocating copies
  - For construction, an object does not yet have an allocator installed, so choose the same one as the object that is moving

- Move-constructuct using allocator uses the supplied allocator by delegating to
  - if the using allocator matches \texttt{allocator\_of}(rvalue), the move constructor
  - Otherwise the copy constructor, so class invariants are managed in one place
Accessing Memory Resources outside their Lifetime

- Basic pmr usage is addressed by C++ object lifetime
  - (local) memory resource must be declared before (local) object that uses it
- Static initialization cannot use the default memory resource specified by main
  - Support for a static duration global resource
  - Global resource given by a replaceable function
Allocating Memory
Leaving the least interesting case until last

• Allocate and release memory directly with a memory resource
  • Retrieve memory resources from objects using `allocator_of`
• Provide an allocator type within the standard library
  • Analogous to `std::pmr::polymorphic_allocator<>`
  • Call `a.new_object<TYPE>(args...)` to allocate and construct
  • Call `a.delete_object(ptr)` to destroy and deallocate
• Provides the initial allocator
  • The new fundamental type is never exposed to the user
Open Design Questions
Unresolved Design Concerns
Each of the topics below needs to be explored in detail

• Explicit factory functions (providing an allocator/object for function use only)
• Providing allocator/objects to initialize function arguments
• Providing allocator/objects to whole expressions, or subexpressions
• Providing explicit (and different) allocator/objects to different member initializers
• Accessing using argument to constructor/factory function
• Customising the move constructor (pair<string, unique_ptr> problem)
• Customisation API to optimize storage, e.g., for any/optional
Next Steps…
Future Work
Currently planned next steps

• Progress the “related papers” on trivial relocation
  • Pick up the paper on guaranteed NRVO
• Rewrite paper P2685 using P3004 Principled Design
• Reconsider how much can be simplified with reflection, P2996
• Establish how much of the design space must be solved for a minimal feature open to future extensions (the Contracts MVP approach)
  • Expect the focus to be on Viable, rather than Minimal
• Semi-related: P1160 Test Resource becomes much more useful