Language Support For Scoped Objects

Document #: P2685R1
Date: 2023-05-04
Project: Programming Language C++
Audience: EWGI
Reply-to: Alisdair Meredith
<ameredith1@bloomberg.net>
Joshua Berne
<jberne4@bloomberg.net>

Contents

1 Abstract

The C++11 Standard Library introduced support for a new object model, hereafter referred to as the *sscoped object model*, to support correct use of stateful allocators. Key properties of the scoped object model are that once a scoped object is imbued at the construction of object `x`, then it cannot change throughout the lifetime of object `x`; this leads to the concern that scoped objects do not change during assignment or swap operations, leading to *propagation traits* that recognize this behavior. The next key property is that every valid form of construction of object `x` would need to accept a user supplied scoped object, and use a default where no scoped object is supplied by the caller; this property leads to the inability to support arrays and other aggregates, and a similar lack of support for lambda objects. The final key behavior is that all elements participating in the same data structure, such as base classes, data members, or dynamically allocated owned objects, must use the same scoped object as the parent object.

Our experience with the library emulation of scoped object types has proved the method to be cumbersome, complex, with much boilerplate code — particularly for folks who are not intending to make use of the available customizations themselves. We believe this has discouraged further use of the idiom to manage resources other than memory.

The design we present for language-based support for scoped objects is not yet complete; we are, however actively seeking early feedback on outstanding questions that must be resolved before bringing this work back as a full proposal.

This paper aims to help us gauge interest, understand the full scope of potential users, and find collaborators.

2 Revision History

- R1
  - Migrated source to use Michael Park's Pandoc template
  - Targeted the proposal more clearly on the scoped object model
  - Consistent use of em dashes and en dashes — these things matter!
  - Added open question on `constexpr` polymorphic allocation
  - Renumbered and reorganized subsections
- Original
  - Original version of the paper for the 2022 October (pre-Kona) mailing.

3 Introduction

Today's standard library has nearly full support for writing types that make effective use of (scoped) *polymorphic allocators*. The library components provided in `std::pmr` (C++17) bring to fruition the results of decades of effort by many parties to realize the local (*runtime* polymorphic) allocator model, pioneered by John Lakos at Bear Stearns (c. 1997–2001) and refined and used heavily by the BDE team at Bloomberg (c. 2002–). With `std::pmr` the decades of benefit that were enjoyed locally are now readily accessible to all users of standard C++.

The use of `pmr` scoped allocators, however, still comes with significant complexities, development and maintenance costs, and limitations in interoperability with other C++ language constructs. Fortunately such complexities, costs, and limitations are not (or need not be) endemic to custom allocators. As this paper aims to demonstrate, the "everyone's is a winner!" solution to the (optional) use of custom allocators lies with a *language-based* solution.

The difficulties inherent in using scoped allocators today arise from two primary sources, (1) excessive boilerplate source code and (2) lack of interoperability with several other features in the language. First, any object that contains an *allocator-aware (AA)* base class or member will need to provide significant boilerplate constructors and operators, which are both costly and error prone to write by hand. In particular, the *rule of zero (RoZ)* cannot be applied to such types. Second, objects for which users choose not to (e.g., aggregates) or cannot (e.g., lambdas) write their own constructors *cannot* be made allocator-aware, removing the ability to leverage many modern C++ features when writing allocator-aware types. On top of those two concerns there are many challenging edge cases in type implementation that are made more verbose and error-prone for allocator-aware types, just to handle the varied ways in which allocators can be passed and must be managed during an object's lifetime. We will dig into these issues in much more detail as part of delineating the specific problems we hope to solve.

The central facet of our proposed solution is to make an addition to the syntax of initialization, leveraging the `using` keyword, that allows for the notion of a *side channel* to pass the allocator through any construction of a type that contains such a *scoped property* (which we generalize below). This initialization back channel transitively and automatically passes a consistent value to initialize the value of the scoped property to the initialization of all base classes and members, enabling the maintenance of the fundamental invariants of the scoped model with no need for additional code.

As part of this initial presentation, we'll also explore many of the corners of the language that will need to be addressed to complete support of this kind of scoped property, and importantly we will show how the availability of these features would drastically simplify the design, specification, and implementation of various common types that currently support generic allocators.

Finally, in addition to having a quarter century's experience working with local allocators in production, the folks at Bloomberg — this author in particular — have been researching this topic for better than five years now. Though we have long since settled on a general syntax and come to many conclusions as to where and how this feature would best integrate into a more cohesive version of the C++ language, we still have some open issues, and our lowest-level design for language-based allocators is still a work in progress. Hence, we have chosen below to pose a number of open questions in lieu of providing a complete proposal that we ourselves know is not yet fully baked. We hope to evolve this proposal into something more general that tackles all known potential use cases cleanly and intuitively.

Our final proposal will facilitate making use of objects having *scoped allocators* without needing to write any boilerplate to support them in any layers above those that do raw memory allocation directly — most of which will be provided by the standard library. Once complete, the *rule-of-zero (RoZ)* will at last be applicable to *allocator-enabled (AE)* types, many language-generated objects will naturally support allocators if and as needed, and writing allocator-aware software will become seamless, much easier to write, and drastically less error-prone. Moreover, we hope to generalize the support needed for such *scoped properties* to enable new designs beyond allocators to other domains, such as executors or loggers, where a side channel with well-defined automatic scoping could be beneficial.

4 State the Problem

The C++ Standard Library has long supported custom allocators, but in practice their use has been largely limited to stateless allocators that are bound at compile-time, due to a variety of issues that arise once allocator state is embraced. C++11 improved support for stateful allocators through the use of `std::allocator_traits`, which first tackled the tricky issue of allocator propagation, which is particular to stateful allocators.

C++17 further improved support for making use of a single, general-purpose stateful scoped allocator with the addition of `std::pmr::polymorphic_allocator`. By specifying specific choices for the complex set of behaviors available to a fully-generic allocator template argument it became possible to write significant code that separated the choice of allocation strategy from the implementation of a type and made it something that could be chosen optimally at runtime. In particular, the availability of a common vocabulary for a polymorphic allocator type allows for choosing distinct allocation algorithms for distinct tasks, often for the same type within the same program.

One of the key principles underlying the use of `std::pmr::polymorphic_allocator` is the *scoped object model*, which it implements. The idea of a scoped object is that some key aspect of a data structure must be accessible at any point within the scope of that data structure. Hence, any base object, data member, or dynamically allocated member of the data structure that has behavior depending on that aspect will store a reference to an implementation of such. All parts of the data structure must refer to that same implementation, for consistent behavior. Once an object is initialized with such a reference, it cannot be rebound, or else we lose the guarantee that all parts of the data structure have the same behavior. This leads to the notion that scoped objects do not propagate their scoped aspect to another object on copy, assignment, swap, etc., which is counter-intuitive to the traditional C++ object model where, by default, all members would transfer on a copy, move, or swap. Much of this proposal deals with making the use of scoped objects more intuitive in the language.

4.1 Scoped Allocator Boilerplate

There is a lot of repetitive code that must be written to make a type properly allocator-aware, supporting the stateful scoped allocator model. Every such class must have a type alias named `allocator_type`, which is detected by the `uses_allocator` type trait, and an allocator-accepting overload of every constructor. The latter is often accomplished using default arguments, but there may need to be many new constructor overloads if there are already constructors relying on further default arguments. For example this was missed in the initial specification of unordered containers, and took a couple of subsequent standards to get right. See below for an exploration of the excessive constructors in the unordered containers.

The additional overloads for trailing default arguments can be mitigated by moving the allocator the to the front of the parameter list, preceded by `std::allocator_arg_t`, and this is the preferred choice for new types. However, this now means that uses-allocator construction must be aware of both conventions, and use some form of reflection to determine which rule to follow for a given initialization. That reflection is achieved today with some primitive, but otherwise needlessly expensive, template metaprogramming. For example, take a look at the implementations of `make_obj_using_allocator` and `uninitialized_construct_using_allocator` in your favorite standard library.

The addition of the allocator parameter means that a type cannot simply default its default, move, and copy constructors, but must define the allocator-accepting overloads, even when the class does not directly store the allocator to allocate memory itself, and the constructors merely pass the allocator along to relevant bases and members.

Similarly, if the class does directly store a copy of the allocator itself, the assignment operators must also be user provided, to handle correct behavior of not assigning to the allocator member. The same issue arises for defaulted comparison operators which must not consider the allocator as a constituent part of the object's value.

4.2 Scoped Object Propagation

An important property of scoped objects is that once they are imbued into an object, they are immutable and cannot change, just as if they were held by a reference. Note that reference members do not rebind on assignment or `swap`, and in fact make those functions ill-formed unless the user provides their own assignment operator. The only time a scoped object will propagate from one object to another is when the move constructor is called without an explicitly supplied scoped object; in such cases the scoped object *must* propagate in order to properly respect move semantics. In all other initializations, if no scoped object is implicitly supplied then some kind of system default will be provided by the environment instead. This applies even to the copy constructor.

The other imporant "scoped" property is that all objects participating in the same data structure, i.e., base classes and non-static data members, but also dynamically allocated object such as nodes in a graph and elements in a `vector`, must be consistent in their scoped objects so that interoperability can be assumed.

Our example in the Standard allocator model addresses allocator propagation through `allocator_traits` with the following separately specializable traits:

— `propagate_on_container_copy_assignment` (POCCA)
— `propagate_on_container_move_assignment` (POCMA)
— `propagate_on_container_swap` (POCS)

as well as the function `select_on_container_copy_construction()` (SOCCC) to control such propagation in the copy constructor. This demonstrates some design mistakes that are now hard to address, trying to provide too much generality for many different kinds of allocator models. In practice, issues arise if the three POCCA, POCMA, and POCS traits are not consistent. For example, we must suddenly know exactly which function will be called by any function implementation; this was not an issue when we could assume consistency, i.e., either all `true` or all `false`. How should you implement `swap` for a type whose allocators propagate on `swap` but not on move assignment, or vice-versa; the default implementation of the `swap` function that is ignorant of `allocator_traits` will always use move-assignment, breaking the contract for one of the inconsistent traits. Thus, the ADL `swap` function must always be defined for a class template that takes any Standard conforming allocator.

The scoped object model discards all of this complexity by never propagating. An open question is whether assignment should be disabled by default (just as for a type with a reference member) or just ignore any scoped objects when synthesizing a defaulted assignment operator. The latter seems more appealing for simplicity, but is also more novel.

4.3 Scoped Allocators and Aggregates

As aggregates are not allowed to define constructors, there is no way to force consistency of allocators, nor a way for containers and other types to pass an allocator at initialization. Consequently, any attempt to force `uses_allocator_v<Aggregate, Allocator>` to be `true` will produce a type that behaves badly in a container.

A determined user might still be able to force correct behavior by carefully enforcing consistent allocators at initialization, and using member functions that rely specifically on moving to insert into a container. Careful usage will behave correctly, until the first single accidental insertion by copy inserts an element using the system default memory resource, breaking the single allocator for all elements guarantee.

For example:

Modern C++	Scoped Object Support
`struct Aggregate {`	`struct Aggregate {`
`   std::pmr::string data1;`	`   std2::string data1;`
`   std::pmr::string data2;`	`   std2::string data2;`
`   std::pmr::string data3;`	`   std2::string data3;`
`};`	`};`
`std::pmr::test_resource tr;`	`std::pmr::test_resource tr;`
`std::pmr::polymorphic_allocator a1(&tr);`	
`Aggregate ag  = {{"Hello", a1}, {"World", a1},`	`Aggregate ag using tr = {"Hello", "World", "!"};`
`std::pmr::vector<Aggregate> va(a1);`	`std::pmr::vector<Aggregate> va using tr;`
`va.emplace_back(std::move(ag));   // Correct al`	`va.emplace_back(std::move(ag));   // Correct allocator re`
`va.emplace_back(ag);  // Error, copied lvalue h`	`va.emplace_back(ag);  // Scoped allocator applied to copie`
`va.resize(5);         // Error, new values have`	`va.resize(5);         // All elements use scoped allocato`
`va.resize(1);         // OK, remove all objects`	`va.resize(1);         // OK`

That this example works at all is due to aggregates move/copy being simple memberwise move and copy, so the individual members enforce consistency if initialized correctly. There is a another worked example further down the paper that looks into mitigations for this, and how our proposal leads to a much simpler user experience.

Note that we are relying on `Aggregate` having a `noexcept` move constructor. What happens if we try to use `std::pmr::list`?

```
std::pmr::polymorphic_allocator a2;
Aggregate agl  = {{"Hello", a2}, {"World", a2}, {"!", a2}};

std::pmr::list<Aggregate> lsa1(a2);
lsa1.emplace_back(std::move(ag));   // Correct allocator retained by moves
std::pmr::list<Aggregate> lsa2 = std::move(lsa1);  // QoI: may throw


std::pmr::vector<std::pmr::list<Aggregate>> vla(a2);
vla.emplace_back(std::move(lsa1));  // Correct allocator retained by move
vla.reserve(5);                     // QoI on list whether correct allocator retained
```

The issue here is that it is QoI whether `std::list` has a non-throwing move constructor, to allow for implementations that need to allocate a sentinel node after moving. Depending on whether the implementation of `list` has a non-throwing move constructor, `vector` will either move or copy the existing elements into the new buffer allocated by `reserve`.

4.4 Scoped Allocators and Lambda Expressions

Lambda expressions carry the same problem as aggregates, that users cannot specify an allocator for the capture lists, producing a type that is not allocator-aware, and similarly cannot specialize the `uses_allocator` trait.

4.5 Allocator-aware Move Construction

One principle underlying the scoped allocator model is that all initialization, when not explicitly supplied an allocator, uses the system supplied default memory resource. However, this breaks down for the move constructor, that clearly cannot achieve the expected efficiency for a move if the new object does not have the same allocator as the moved-from object. Hence, there is a specific exception to the rule so that the move constructor should take its allocator from the moving argument.

Now let us consider the following simple example:

```
struct MyType {

   MyType(MyType const & other, std::pmr::polymorphic_allocator<> a = {});
   MyType(MyType      && other) = default;
   MyType(MyType      && other, std::pmr::polymorphic_allocator<> a);

private:
   std::pmr::vector<int>  d_data;
   std::pmr::vector<int*> d_index;
};
```

This class maintains an invariant that the pointers in `d_index` always refer to elements in the vector `d_data`. This invariant is preserved in the definition of the copy constructor. It is preserved by the defaulted move constructor, as both vectors move and that preserves the invariant. The question arises as to what is the preferred semantic of the move-with-allocator constructor? A naive implementation might perform a memberwise move-with-allocator construction for each data member, however that would break the invariant. A better implementation would be to use a delegating constructor to simply forward to the copy constructor, that enforces the invariant. However, this implementation loses the optimized move when the supplied allocator argument matches that of the `other` argument. The preferred solution is therefore to test if the allocators are equal, and then delegate to either the move constructor, or the copy-with-allocator constructor. This is our recommended default best practice, and one possible implementation is part of the worked example below.

Issues surrounding move-only scoped allocator types remain an open question

4.6 Allocator-aware `swap`

The generic `swap` function that forms the basis of the swappable requirements and concepts has a wide contract that permits exceptions. There is a lot of code that expects ”best practice” where the `swap` function cannot throw, and has constant complexity. While this is not guaranteed for the general case, the standard provides this guarantee where it is able, such as for the standard containers.

When two containers have different allocators that do not propagate, it is not possible to implement a non-throwing, constant complexity, `swap` function. These performance guarantees are available only at run time, when the allocators compare equal.

Hence, implementations of scoped types must choose between narrowing the wide contract on `swap` with a precondition that allocators compare equal (noting the lack of a standard requirement to provide a `get_allocator` function to check, although it is frequently added as best practice), or allowing the function to throw and make copies knowing that copies are often not constant time constructs.

The standard has gone with narrowing the contract, although this is thought by many to be a mistake, and there remains an open issue on this question, [LWG2153]. Note that while it is reasonable to later specify specific behavior to replace UB, as any assumptions on such code are already broken, it is not simple to declare existing behavior undefined.

4.7 `noexcept` and Differing Allocators

There is a general concern for functions that adopt or exchange resources that, in order to avoid allocation, there is a precondition that the objects have the same allocator at runtime; otherwise allocation may occur, and so the operation may throw. Hence, many operations that we would like to indicate do not throw cannot lean on C++ language features such as `noexcept` that enforce the constraint at compile time. For example, the move-assignment operator will typically be `noexcept(false)`, and the standard library containers put a precondition on non-member `swap`.

This concern is relatively minor for non-template code, as the specific types with their known contracts allow testing whether the allocators are the same before selecting an implementation algorithm. This concern is much harder to work around in generic code though, where simply knowing whether move assignment or swapping may throw is the trigger for choosing different implementation algorithms.

4.8 Awkward Standardese

Quoting Jonathan Wakely (with permission) from a reflector discussion on [LWG3758]:

> Of course this all ignores the problem that the container has no real way to know whether move insertion is potentially throwing, because it doesn't know which constructor move insertion will use, that's decided by the allocator. So the container checks `is_nothrow_move_constructible` to decide whether to move insert or copy insert, but the allocator might not use the same constructor that `is_nothrow_move_constructible` considers. Yay.

The issue here is that container elements are initialized by `allocator_traits::construct` that, by default, is implemented with *uses-allocator construction* where the actual constructor chosen is determined by a formula looking to see if an allocator is supported. If an allocator supporting constructor is chosen, it is expected to have the same `noexcept` guarantees as the same constructor that does not accept an allocator, but there are no guarantees in the standard that this must be the case. Worse, the optional `construct` member function of an allocator is a deliberate extension point to allow the allocator to use other initialization forms that might not call the `noexcept` constructor that the container requirements are specified in terms of. In practice, we write code to support the intent, but there are no such guarantees present in the standard itself.

5 Propose a Solution

The following proposes a partial solution, with the details we have worked through to a reasonable level of confidence. It is expected some minor details may change as we address the Open Questions deeper in this paper, but these are the foundations that a complete solution will build on.

For simplicity of our current proposal, we are focusing exclusively on scoped allocators of a single type, but see the discussion of Scoped Types Beyond Allocators towards end the of this document.

For the purpose of illustration, all examples of new library code will be written in namespace `std2`. We are not proposing any library extensions at this point, so this should not be seen as proposing such. However, it is a conveniently short namespace for our examples, and distinct from namespace `std` itself. Where code uses `std` rather than `std2` it is intentional, e.g., to illustrate that `std::tuple` needs no changes to support the new facilities.

Most of our examples in `std2` show types that would be equivalent to the corresponding type in `std::pmr`, for familiar examples with significantly less boilerplate standardese being needed.

5.1 Language Support for Scoped Object Types

The essential problem that needs resolving is the complexity inherent in the scoped model, and how it has different expectations to the defaults inherent in the C++ language. This leads to requiring support to pass allocators to every constructor/initializer, and simply requiring this property violates the rule of zero, putting aggregates such as arrays beyond our ability to support. Hence, the place to start is with language support to make the scoped model as well supported as today's defaults, re-enabling the rule of zero.

5.1.1 Pass via `using`

The essential idea is that a scoped property, in this case an allocator, must be supported in every initializer. That means that every initialization syntax must allow an optionally supplied allocator, and have a good default behavior for the (common) case that no allocator is supplied.

To avoid redundancy in constructors, and to reach types where there is no ability to write constructors, we need another mechanism to pass the scoped values to object initialization. We propose to do this with a new *using-expression* on variable initialization. We pick on the `using` keyword as the best grammatical English token that is already reserved, and sufficiently underloaded within initialization to serve our purposes. A simple example of this syntax might be:

Modern C++	Scoped Object Support
`int main() {`	`int main() {`
`   std::pmr::test_resource tr;`	`   std::pmr::test_resource tr;`
`   std::pmr::vector<int>   vi({1, 2, 3}, &tr);`	`   std2::vector vi using tr = {1, 2, 3};`
`}`	`}`

Our initial idea was that `using` feels more intuitive at the end of any expression (or subexpression), as that feels like it would expand into many more use cases. However, that turned out to be why we opted against that syntax, at least for the initial proposal, as it leads to much more complexity, parsing ambiguities that would need to be resolved, and distracts from our primary focus, at least for now. See the Open Questions for further discussion.

```
int main() {
   std::pmr::test_resource tr;
   std2::vector vi = {1, 2, 3} ~~using tr~~;  // Error! `using` does not go here
}
```

5.1.2 Implicit Propagation

The essential feature of the scoped allocator model is that all objects that participate in the same data structure, meaning base classes, non-static data members, and dynamically allocated data, must use the same allocator. There is a lot of boiler-plate code to get correct in constructors to ensure that this property holds, so we propose a feature where *allocator-enabled* types are implicitly supplied their allocator in the base and member initializers from their derived/containing class, and that there is no other syntax to accidentally override this.

An allocator-enabled class is defined recursively, as a class that comprises any allocator-enabled base classes or non-static data members. At the root of this hierarchy will be a new fundamental type, tentatively named `_Resource` prior to a formal naming discussion, that will have further special properties detailed below.

Here we show how regular language arrays, which have no constructors, can now support elements that use allocators via the implicit propagation property of scoped element types.

```
int main() {
   std2::test_resource tr;
   std2::string s2[] using tr = { "Hello", "world" };
}
```

Similarly, we should expect the standard types to implicitly support our scoped behavior, where they directly store an object of template parameter type, as all the constructors implicitly pick up a scoped allocator argument.

```
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 };

   static_assert(is_same_v< decltype(t4)
                          , tuple< pmr::string
                                 , pmr::string
                                 , pmr::string
                                 , pmr::string
```

```
                              >
                          >);

   assert(get_allocator(p2) == &tr);
   assert(get_allocator(t4) == &tr);

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

   assert(get_allocator(get<0>(t4)) == &tr);
   assert(get_allocator(get<1>(t4)) == &tr);
   assert(get_allocator(get<2>(t4)) == &tr);
   assert(get_allocator(get<3>(t4)) == &tr);
}
```

If we were to attempt the same test with modern C++, it would look something like:

```
#include <memory_resource>
#include <string>
#include <tuple>
#include <utility>hv

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")
              };

   static_assert(is_same_v< decltype(t4)
                          , tuple< pmr::string
                                 , pmr::string
                                 , pmr::string
                                 , pmr::string
                                 >
                          >);

~~// assert(get_allocator(p2) == &tr);// No equivalent~~
~~// assert(get_allocator(t4) == &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);
}
```

5.1.3 Implicit and Defaulted Operators

Objects that support the scoped model share a couple of features. The scoped member cannot be changed, as it does not propagate. This would make all assignment operators ill-formed (and implicitly deleted) without a user-provided definition that ignored the scoped object. Similarly, the scoped object is typically *not* a salient data member for comparison, and comparing the scoped values would produce false-negative results. For example, scoped allocators are never salient.

The ideal default behavior for implementation-provided assignment and comparison operators for types that comprise scoped objects (such as allocators) is to simply ignore scoped members, leaving them unchanged.

5.1.4 Move Operations

One of the tricky problems for allocator enabled types is the notion of move operations. One fundamental property is that the allocator never propagates, and we can easily build support for that into our proposal. However, the *pmr* model comes with the additional constraint that move operations are potentially throwing if the memory resources are not the same. The simplest consequence is that the move-assignment operator will always be `noexcept(false)`. The deeper question is whether we can get move construction right?

The basic issue is that if we do not supply an allocator via `using` to initialize an object, then the system default resource will be used. This turns out to be the intended behavior for the copy constructor, but not for the move constructor. To properly effect a move, we need the newly constructed object to have the same allocator as the moving object. Hence, we propose a special rule when initializing from an rvalue.

Should we restrict to rvalue of the same type, or any rvalue?

The move constructor will always receive the allocator of the rvalue when there is no supplied `using` argument. If a `using` allocator is supplied, then, at the call point, the compiler will generate code to compare the supplied allocator with that of the rvalue. If the allocators are the same, then the move constructor is called. If the allocators are different then the *copy* constructor is called with the supplied allocator; if the copy constructors is not available, then the program is ill-formed.

We may wish to allow customization of this behavior by overloading the move constructor with original and allocator-enabled forms, but that is left as an open question for now, as we do not (yet) propose a syntax to pass allocators to arbitrary functions for overloading.

Also considered was making it undefined behavior or throwing a `bad_allocator` exception if the copy constructors is not available. We would like to see stronger motivation before injecting more UB or another exception from the language itself. This remains an open question.

5.2 Implicit Factory Functions

A *factory function* is any function that returns an object by value. For the purposes of this proposal, an *implicit factory function* is any factory function that returns an allocator-aware type, and is shorthand for ”implicitly allocator-enabled factory function”.

5.2.1 Passed an Allocator

When a factory function returns an allocator-enabled object by value, clearly the caller would like to be able to supply an allocator that the returned value would use. In such cases, the factory function will implicitly accept an allocator via the `using` mechanism.

Note that for the initial proposal, the only place a `using` clause is valid is on variable declarations, so at this point the only way to supply the allocator to a factory function is to declare a variable, and supply the allocator to the declared variable. That same allocator will be supplied to the implicit `using` clause for the factory function.

5.2.2 Implicit `using` on Return Expressions

Every `return` expression in an implicit factory function comes with an implicit `using` of the allocator supplied to the function call. This `using` will inhibit RVO unless the compiler can prove that the returned object was constructed with the same allocator. Note that this will trivially be the case where RVO has been mandated since C++17/20.

5.2.3 Potential Return Variables

A potential return variable is any value that, once constructed, will be subject to a return expression. A function may have multiple potential return variables, such as when an `if/else` clause contains two different object and return paths. More detailed examples may be found in the separate proposal: [P2025R2] Guaranteed Copy Elision for Return Variables, although a simpler formulation may suffice for this paper, and RVO may remain optional rather than guaranteed.

In general, potential return variables, as well as any variable currently subject to potential NRVO, should not include the case of a variable with an explicit `using` clause, as that using clause may conflict with the allocator supplied to the factory function.

Where the compiler can identify a declared variable as a *potential return variable*, initialization of that variable will have an implicit `using` clause to use the allocator supplied to the factory function. This will result in either constructing the variable in-place with the requested allocator or move-constructing it with an already matching allocator upon `return`.

5.3 Eliminate Dependency on `std::allocator_traits`

The standard allocator-aware containers have a dependency on `std::allocator_traits` for any use of an allocator. This is what supports such versatility as allocators returning *fancy* pointers, having a variety of different allocator propagation traits (POCMA etc.), and the ability to customize allocation of elements, e.g., using the container's allocator for allocators supporting the scoped allocator model.

The `std::allocator_traits` model also comes with assumptions that allocators are passed as regular function arguments, which would no longer hold under our new language-supported model. Note that `constexpr` allocation is also built on top of `std::allocator_traits`, which may be a problem we must tackle later — see open question.

This proposal deliberately removes much of the flexibility supported by `allocator_traits`, baking in such assumptions as never propagating allocators, and `allocate` returning a plain pointer. This will allow us to simplify our assumptions, providing the appropriate default behavior in each case, so indirection through `allocator_traits` should be unnecessary. While it is certainly possible to build an allocator on top of these new facilities that would plug into `allocator_traits` the experience might be less than ideal, as the `allocator_traits` interface is designed for types that take actual allocator arguments, rather than following the general purpose model we propose where allocators must necessarily be passed independently from the initializer arguments.

5.3.1 Build on `std::pmr`

The intent of this paper is to propose flexible support for allocators, while also taking the allocator parameter out of class templates. This is the basis of the `std::pmr` polymorphic memory resource facility added to C++17, so our proposal will simply adopt this protocol rather than re-invent that particular wheel. We refer to the material of that time for reference to why this is an important and good fit for providing allocators at runtime.

5.3.2 Hidden Friend `allocator_of`

An important part of the polymorphic allocator protocol is to determine the allocator used by any given allocator-enabled object. We propose adding a "hidden friend" function, `allocator_of`, that can answer this question. Furthermore, `allocator_of` will be implicitly defined for any allocator-enabled type to return `allocator_of` one of the bases or members that granted that status to the class. There will be some extra core wording to transform `allocator_of` an array to the `allocator_of` one of its elements.

6 Related Proposals

6.1 Importance of Trivial Relocatability

One of the big optimizations of C++11 was the move optimization when growing a `vector`. If the element type has a non-throwing move constructor, then we can move all of the elements from the current `vector` buffer into the new, larger, buffer just allocated for that `vector`, potentially saving many re-allocations by each element. However, if we cannot prove that such moves are non-throwing (such as by querying the `noexcept` specification on the move constructor) then, in order to preserve the strong exception safety guarantees provided by this container, we must fall back on the whole `copy`/`swap` idiom, doing the work in a side buffer until we are ready to commit, and making a full copy of each element in the process.

While it is relatively easy to provide a non-throwing (and `noexcept`) move constructor for allocator-enabled types, `vector` will actually call the move-with-allocator constructor, that will always be `noexcept(false)`. The type system cannot see the guarantee that the scoped model allocator being supplied at runtime will guarantee, *at runtime* that no exceptions will propagate.

Many such types would be *trivially relocatable* though, and optimizations built around this property may become more important. For more information on trivial relocation see the papers [P2814R0], [P1144R8], and [P2786R1].

6.2 Runtime `noexcept`

As noted above, in many cases allocator-enabled types can report at runtime, rather than compile-time, whether there is a risk of propagating an exception. If we could query for such a property, then library code could optimize with a regular `if` statement at runtime, rather than forcing that same optimization choice be exclusively at compile time.

A facility to expose such a runtime query from a type would complete the picture, but as we have noted it is of lower importance in the presence of trivial relocatability.

7 Examples

We present several examples that illustrate how code would be simpler and more consistent when expressed using our proposed features.

7.1 Simple Aggregates

Two of the initial motivators that lead to this paper was the awkwardness of aggregates with allocators, and the amount of boilerplate associated with writing a proper allocator-aware class. This example illustrates both, as the amount of boilerplate when emulating a simple aggregate emphasizes the work involved, that is often just careful attention to details that must otherwise be addressed anyway for a non-aggregate class.

7.1.1 A Basic Aggregate

Let us consider a simple aggregate using a scoped allocator type in C++23. For simplicity we will use `std::pmr::string` as our scoped allocator aware data member. To complete the example, we also provide the key comparison operators, which we can simply default. To simplify code comparison, all the subsequent example types will have an identifier of exactly the same length.

```
struct BasicAggregate {
   std::pmr::string data1;
   std::pmr::string data2;
   std::pmr::string data3;
   friend auto operator ==(BasicAggregate const &, BasicAggregate const &) -> bool = default;
   friend auto operator<=>(BasicAggregate const &, BasicAggregate const &) -> std::strong_ordering = defa
};
```

`BasicAggregate` does not work well as the type stored in a `std::pmr::vector` as it does not advertise that it uses `pmr` allocators — which is a good thing, as it does not! There is no easy way to supply an allocator to the members, although a determined user will find a way:

```
std::pmr::polymorphic_allocator a1;
BasicAggregate bd;
BasicAggregate bd1 = {"Hello"};
BasicAggregate bd2 = {"Hello", "World"};
BasicAggregate bd3 = {"Hello", "World", "!"};

BasicAggregate b   = {{{}, a1}, {{}, a1}, {{}, a1}};
BasicAggregate b1  = {{"Hello", a1}, {{}, a1}, {{}, a1}};
BasicAggregate b2  = {{"Hello", a1}, {"World", a1}, {{}, a1}};
BasicAggregate b3  = {{"Hello", a1}, {"World", a1}, {"!", a1}};
```

Observe that in order to enforce a consistent allocator, all aggregate members must be aggregate initialized, with the same allocator passed to initialization in each case. Once initialized this way though, the correct allocator will be used for copy and move construction, as those are performed memberwise for an aggregate, and `std::pmr::polymorphic_allocator` has the correct behavior.

7.1.2 `std::array` as an aggregate

Some of you may have noticed that `std::array` also provides a homogeneous aggregate, and also supplied the comparison operator, so how would this code look using `std::array`?

```
using ArrayAggregate = std::array<std::pmr::string, 3>;
ArrayAggregate aps   = {{{{}, a1}, {{}, a1}, {{}, a1}}};
ArrayAggregate aps1  = {{{"Hello", a1}, {{}, a1}, {{}, a1}}};
ArrayAggregate aps2  = {{{"Hello", a1}, {"World", a1}, {{}, a1}}};
ArrayAggregate aps3  = {{{"Hello", a1}, {"World", a1}, {"!", a1}}};

// Accept the default allocator

ArrayAggregate apsd;
ArrayAggregate apsd1 = {"Hello"};
ArrayAggregate apsd2 = {"Hello", "World"};
ArrayAggregate apsd3 = {"Hello", "World", "!"};
```

Here we observe that *yet another* pair of braces is needed to surround the object initialization, so that the aggregate nested within `std::array` takes the whole supplied initialization, rather than trying to initialize the internal aggregate with just the first braced initializer. The simplicity when using just the defaults is repeated for comparison.

7.1.3 Emulating an Aggregate

No matter the effort we run to to force a consistent set of scoped allocators across all of our elements though, aggregate types can never serve as proper scoped allocator types element types for standard `pmr>` containers as:

— They do not have an `allocator_type` member, nor do they specialize `std::uses_allocator` to inform the container that they desire an allocator.
— They do not have an initialization that accepts a single allocator that is supplied to all (relevant) members.

Suppose we wished to write a properly allocator-aware class that is a drop-in substitute for our `BasicAggregate` in almost every way, but that correctly supports `pmr` allocators? That might looks something like the following. Note that modern best practice suggests we use `std::pmr::polymorphic_allocator<>` as the common vocabulary type for `pmr` allocators, rather than trying to force an allocator for a single type. See [P0339R6] for more details.

```
struct AproxAggregate {
   std::pmr::string                  data1;
   std::pmr::string                  data2;
   std::pmr::string                  data3;
   std::pmr::polymorphic_allocator<> alloc;

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

   // Default, move, and copy constructors

   AproxAggregate() = default;
   AproxAggregate(AproxAggregate&&) = default;
   AproxAggregate(AproxAggregate&& other, std::pmr::polymorphic_allocator<> a); // see below

   AproxAggregate(AproxAggregate const& other, std::pmr::polymorphic_allocator<> a = {})
   : AproxAggregate(other.data1, other.data2, other.data3, a)
   {}

   // Value constructors

   explicit AproxAggregate(std::pmr::polymorphic_allocator<> a)
   : AproxAggregate(std::pmr::string{}, std::pmr::string{}, std::pmr::string{}, a)
   {}

   template <typename T>
      requires std::constructible_from<std::pmr::string, T>
   explicit AproxAggregate( T&& s
                          , std::pmr::polymorphic_allocator<> a = {}
                          )
   : AproxAggregate(std::forward<T>(s), std::pmr::string{}, std::pmr::string{}, a)
   {}

   template <typename T1, typename T2>
      requires std::constructible_from<std::pmr::string, T1>
           and std::constructible_from<std::pmr::string, T2>
   AproxAggregate( T1&& s1
                 , T2&& s2
                 , std::pmr::polymorphic_allocator<> a = {}
                 )
   : AproxAggregate(std::forward<T1>(s1), std::forward<T2>(s2), std::pmr::string{}, a)
   {}

   template <typename T1, typename T2, typename T3>
      requires std::constructible_from<std::pmr::string, T1>
           and std::constructible_from<std::pmr::string, T2>
           and std::constructible_from<std::pmr::string, T3>
   AproxAggregate( T1&& s1
                 , T2&& s2
                 , T3&& s3
                 , std::pmr::polymorphic_allocator<> a = {}
                 )
   : data1(std::forward<T1>(s1), a)
   , data2(std::forward<T2>(s2), a)
   , data3(std::forward<T3>(s3), a)
   , alloc(a)
   {}

   // Allocator access

   auto get_allocator() noexcept
     -> std::pmr::polymorphic_allocator<> {
      return alloc;
   }

   // Assignment operators

   auto operator=(AproxAggregate&& other)
     -> AproxAggregate & {
      data1 = std::move(other.data1);
      data2 = std::move(other.data2);
      data3 = std::move(other.data3);
      // do not propagate the allocator

      return *this;
   }

   auto operator=(AproxAggregate const& other)
     -> AproxAggregate & {
      data1 = other.data1;
      data2 = other.data2;
      data3 = other.data3;
      // do not propagate the allocator

      return *this;
   }

   // Comparison operators

   friend
   auto operator==(AproxAggregate const &lhs, AproxAggregate const &rhs) noexcept
     -> bool
   {
       return lhs.data1 == rhs.data1
          and lhs.data2 == rhs.data2
          and lhs.data3 == rhs.data3;
       // and ignore the allocator
   }

   friend
   auto operator<=>(AproxAggregate const & lhs, AproxAggregate const & rhs) noexcept
     -> std::strong_ordering {
       return lhs.data1 < rhs.data1 ? std::strong_ordering::less
```

```
            : lhs.data1 > rhs.data1 ? std::strong_ordering::greater
            : lhs.data2 < rhs.data2 ? std::strong_ordering::less
            : lhs.data2 > rhs.data2 ? std::strong_ordering::greater
            : lhs.data3 < rhs.data3 ? std::strong_ordering::less
            : lhs.data3 > rhs.data3 ? std::strong_ordering::greater
            :                         std::strong_ordering::equal;
        // and ignore the allocator
    }

};
```

If that seems like a lot of work, then you understand the motivation for this paper!

First we add a data member to hold the allocator for this object, that will be consistently supplied to all data members, and the `get_allocator` member function so that clients can query for the allocator after construction. This is important information to ensure that moves can happen efficiently, for example. We also add the `typedef` member for the `uses_allocator` type trait to detect, advertising that this class is allocator-aware.

Then we add a set of constructor overloads that allow initialization like an aggregate, where we can supply an initializer for each of the first 0–3 elements, followed by an optional trailing allocator that defaults to value initialized. We constrain each argument on constructability to ensure our type returns the correct answer for the `is_constructible` type trait and for the `constructible_form` concept itself. We then use delegating constructors to simplify concerns and ensure consistency is maintained, perfectly forwarding each argument, and value initializing the ”default” arguments (which cannot be deduced) to ensure behavior consistent with aggregate initialization. Note that a value-initialized polymorphic allocator will pick up the system supplied default memory resource.

Next we add the default, copy, and move constructors, as we can no longer rely on implicit declaration. In order to get the correct `explicit` behavior for the default constructor, we have had to separate the single-argument allocator constructor into the value constructors above, rather than relying on default arguments. The default constructor using the default allocator has the correct implementation supplied behavior, as does the move constructor, so those members can be defaulted. However, for the copy constructor we want to allow the user to supply an allocator, that can be defaulted, so we implement this like the value constructors above, by using a delegating constructor to the common implementation. Note that this copy constructor also serves as the move-with-allocator constructor; we will revisit that topic after the next example, in order to complete both.

Then we address the assignment operators, that are implicitly deleted, as the assignment operators of `std::pmr::polymorphic_allocator` are deleted. These operators are deleted to ensure that we do not accidentally propagate the allocator, breaking the scoped model. Thus, we are forced to always write the assignment operators, taking care to not try to assign the allocator member, `alloc`. The implementation is tedious but straight forward, and vulnerable to falling out of date if the class is modified in the future, like any other used supplied assignment operator. We also take care to remember to move each member when implementing the move assignment operator, a simple opportunity to introduce an error that is equally easily detected by reasonable test driver coverage.

Finally we add the comparison operators to better describe a value semantic type. Again, the presence of the allocator member `alloc` means we cannot simply rely on the default implementation. We note that the spaceship operator, while tedious, is exactly the kind of code where errors are easily introduced by simple typos or ordering constraints, and not something we would want to routinely writing for any types.

What has all this work brought us? We can now write a much simpler usage example!

```
std::pmr::polymorphic_allocator a2;
AproxAggregate x { a2 };
AproxAggregate x1{"Hello", a2};
AproxAggregate x2{"Hello", "World", a2};
AproxAggregate x3{"Hello", "World", "!", a2};
```

Note that this simplified usage example show how code is cleaned up with all typical usage of this class, where allocators matter, highlighting the real problem with aggregates at the point of use.

However, the key takeaway is that this class will behave properly as the element type for any standard container.

7.1.4 Proposed Simplification

So how would this type be expressed using our new language features. Here, we will assume `std2::string` is a new type, exactly like `std::string` other than relying on our new language support for its allocator. Remember, this is for illustration purposes only, we are not actively recommending `std2` or other library changes at this time.

```
struct ScopeAggregate {
   std2::string data1;
   std2::string data2;
   std2::string data3;

   friend auto operator ==(ScopeAggregate const &, ScopeAggregate const &) -> bool = default;
   friend auto operator<=>(ScopeAggregate const &, ScopeAggregate const &) -> std::strong_ordering = defa
};
```

That looks remarkably like the original simple aggregate that started this example, other than the switch of string types. However, this aggregate will correctly work with containers (implemented using the same language support) and can report the object's allocator through the implicitly declared and defined friend function `allocator_of`. Observe that we can finally rely on the classic *rule of zero* when implementing allocator-aware types.

How does our usage example change?

```
std::pmr::polymorphic_allocator a4;
ScopeAggregate s  using a4;
ScopeAggregate s1 using a4 {"Hello"};
ScopeAggregate s2 using a4 {"Hello", "World"};
ScopeAggregate s3 using a4 {"Hello", "World", "!"};
```

The usage example looks more like the simplified usage examples of the emulated types, but the position of the allocator argument has moved, using our proposed syntax. We can omit the `using` expressions to just accept the system supplied default memory resource.

Finally, we can look at the usage example for `std::array` with the proposed facilities. Note that we really do mean *`std`* array, and not some funky new `std2` array type.

```
using StdArrayString = std::array<std2::string, 3>;

std::pmr::polymorphic_allocator a5;
StdArrayString as2  using a5;
StdArrayString as21 using a5 {"Hello"};
StdArrayString as22 using a5 {"Hello", "World"};
StdArrayString as23 using a5 {"Hello", "World", "!"};

// Accept the default allocator

StdArrayString as2d;
StdArrayString as2d1 {"Hello"};
StdArrayString as2d2 {"Hello", "World"};
StdArrayString as2d3 {"Hello", "World", "!"};
```

This is a demonstration that there are generally fewer gotchas with this proposal, such as requiring unexpected additional braces. Observe that the initialization with or without allocators look simple and consistent with each other.

7.2 `std::vector`

For a familiar example, let us take a look at how the current `std::vector` would be revised using these new language features. Note that we do not propose making such a change to the library as part of this proposal, due to obvious ABI (and API) compatibility concerns; this is an illustration of how existing user code might be updated once these features are available, or alternatively, where the simplicity of an implementation using our proposed language support lies. We are using namespace `std2` as per our convention to distinguish new code from true `std` specification.

```
namespace std2 {  // For illustrative purposes only, not a proposal

  // 24.3.11, class template vector
  template<class T~~, class Allocator = allocator<T>~~> class vector;
  template<class T~~, class Allocator~~>
    constexpr bool operator==(const vector<T~~, Allocator~~>& x, const vector<T~~, Allocator~~>& y);
  template<class T~~, class Allocator~~>
    constexpr synth-three-way-result<T> operator<=>(const vector<T~~, Allocator~~>& x,
                                                    const vector<T~~, Allocator~~>& y);
  template<class T~~, class Allocator~~>
    constexpr void swap(vector<T~~, Allocator~~>& x, vector<T~~, Allocator~~>& y);
      ~~noexcept(noexcept(x.swap(y)));~~

  // 24.3.11.6, erasure
  template<class T, ~~class Allocator,~~ class U>
    ~~constexpr typename vector<T, Allocator>::size_type~~
    constexpr size_t erase(vector<T~~, Allocator~~>& c, const U& value);
  template<class T, ~~class Allocator,~~ class Predicate>
    ~~constexpr typename vector<T, Allocator>::size_type~~
    constexpr size_t erase_if(vector<~~T, Allocator~~>& c, Predicate pred);

  ~~namespace pmr {~~
    ~~template<class T>~~
      ~~using vector = std::vector<T, polymorphic_allocator<T>>;~~
  ~~}~~

  // 24.3.12, specialization of vector for bool
  // 24.3.12.1, partial class template specialization vector<bool~~, Allocator~~>
  template<~~class Allocator~~>
  class vector<bool~~, Allocator~~>;

  // hash support
  template<class T> struct hash;
  template<~~class Allocator~~> struct hash<vector<bool~~, Allocator~~>>;

  ~~// 24.3.12.2, formatter specialization for vector<bool>~~
  ~~template<class T>~~
  [`inline constexpr bool_is-vector-bool-reference_=_see below_; // exposition only`]{.rm}

  template<~~class T,~~ class charT> ~~requires is-vector-bool-reference<T>~~
    struct formatter<~~T~~vector<bool>::reference, charT>;
}
```

In addition to simplifying vector by eliminating one of the type parameters, we see a few side-effects as a result:

— The allocator is fully determined, so there is no more type dependency when providing certain types, such as the `size_type` needed for the consistent erasure API.
— `vector<bool>::reference` is no longer type-dependent on the allocator, so can simply be named, entirely eliminating an exposition-only concept.
— The `hash` template specialization is now an explicit template specialization rather than a partial template specialization. This opens up more freedom for where the definitions may be placed in source.
— The free-function `swap` is simplified by losing its `noexcept` specification; this might be seen as losing a feature, but it is already the case for any scoped-type model, such as `std::pmr::vector` today, and it is easier to see. This lack of `noexcept` could be ameliorated with the introduction of a runtime exception specification, but we are not highlighting that extension here, nor `operator swap`.
— Finally, the aliases in namespace `pmr` serve no purpose, so are eliminated.

```
namespace std2 {  // For illustrative purposes only, not a proposal
template<class T~~, class Allocator = allocator<T>~~>
  class vector {
  public:
    // types
    using value_type             = T;
    ~~using allocator_type         = Allocator;~~
    using pointer                = ~~typename allocator_traits<Allocator>::pointer~~value_type*;
    using const_pointer          = ~~typename allocator_traits<Allocator>::const_pointer~~const value_type*;
    using reference              = value_type&;
    using const_reference        = const value_type&;
    using size_type              = ~~implementation-defined~~size_t; // see 24.2
    using difference_type        = ~~implementation-defined~~ptrdifft_;// see 24.2
    using iterator               = implementation-defined; // see 24.2
    using const_iterator         = implementation-defined; // see 24.2
    using reverse_iterator       = std::reverse_iterator<iterator>;
    using const_reverse_iterator = std::reverse_iterator<const_iterator>;

    // 24.3.11.2, construct/copy/destroy
    constexpr vector() noexcept~~(noexcept(Allocator()))~~ : vector(Allocator()) { };
    ~~constexpr explicit vector(const Allocator&) noexcept;~~
    constexpr explicit vector(size_type n~~, const Allocator& = Allocator()~~);
    constexpr vector(size_type n, const T& value~~, const Allocator& = Allocator()~~);
    template<class InputIterator>
      constexpr vector(InputIterator first, InputIterator last~~, const Allocator& = Allocator()~~);
    template<container-compatible-range<T> R>
      constexpr vector(from_range_t, R&& rg~~, const Allocator& = Allocator()~~);
    constexpr vector(const vector& x);
    constexpr vector(vector&&) noexcept;
    ~~constexpr vector(const vector&, const type_identity_t<Allocator>&);~~
    ~~constexpr vector(vector&&, const type_identity_t<Allocator>&);~~
    constexpr vector(initializer_list<T>~~, const Allocator& = Allocator()~~);
    constexpr ~vector();

    constexpr vector& operator=(const vector& x);
    constexpr vector& operator=(vector&& x);
      ~~noexcept(allocator_traits<Allocator>::propagate_on_container_move_assignment::value ||~~
               ~~allocator_traits<Allocator>::is_always_equal::value);~~
    constexpr vector& operator=(initializer_list<T>);
    template<class InputIterator>
    constexpr void assign(InputIterator first, InputIterator last);
```

```
template<container-compatible-range<T> R>
  constexpr void assign_range(R&& rg);
constexpr void assign(size_type n, const T& u);
constexpr void assign(initializer_list<T>);
~~constexpr allocator_type get_allocator() const noexcept;~~

// iterators
constexpr iterator               begin() noexcept;
constexpr const_iterator         begin() const noexcept;
constexpr iterator               end() noexcept;
constexpr const_iterator         end() const noexcept;
constexpr reverse_iterator       rbegin() noexcept;
constexpr const_reverse_iterator rbegin() const noexcept;
constexpr reverse_iterator       rend() noexcept;
constexpr const_reverse_iterator rend() const noexcept;

constexpr const_iterator         cbegin() const noexcept;
constexpr const_iterator         cend() const noexcept;
constexpr const_reverse_iterator crbegin() const noexcept;
constexpr const_reverse_iterator crend() const noexcept;

// 24.3.11.3, capacity
[[nodiscard]] constexpr bool empty() const noexcept;
constexpr size_type size() const noexcept;
constexpr size_type max_size() const noexcept;
constexpr size_type capacity() const noexcept;
constexpr void      resize(size_type sz);
constexpr void      resize(size_type sz, const T& c);
constexpr void      reserve(size_type n);
constexpr void      shrink_to_fit();

// element access
constexpr reference
constexpr const_reference operator[](size_type n) const;
constexpr const_reference at(size_type n) const;
constexpr reference       at(size_type n);
constexpr reference       front();
constexpr const_reference front() const;
constexpr reference       back();
constexpr const_reference back() const;

// 24.3.11.4, data access
constexpr T* data() noexcept;
constexpr const T* data() const noexcept;

// 24.3.11.5, modifiers
template<class... Args>
  constexpr reference emplace_back(Args&&... args);
constexpr void push_back(const T& x);
constexpr void push_back(T&& x);
template<container-compatible-range<T> R>
  constexpr void append_range(R&& rg);
constexpr void pop_back();
```

```
    template<class... Args>
      constexpr iterator emplace(const_iterator position, Args&&... args);
    constexpr iterator insert(const_iterator position, const T& x);
    constexpr iterator insert(const_iterator position, T&& x);
    constexpr iterator insert(const_iterator position, size_type n, const T& x);
    template<class InputIterator>
      constexpr iterator insert(const_iterator position,
                                InputIterator first, InputIterator last);
    template<container-compatible-range<T> R>
      constexpr iterator insert_range(const_iterator position, R&& rg);
    constexpr iterator insert(const_iterator position, initializer_list<T> il);
    constexpr iterator erase(const_iterator position);
    constexpr iterator erase(const_iterator first, const_iterator last);
    constexpr void     swap(vector&) ;
      ~~noexcept(allocator_traits<Allocator>::propagate_on_container_swap::value ||~~
               ~~allocator_traits<Allocator>::is_always_equal::value);~~
    constexpr void     clear() noexcept;

  private:
    _Resource mem-res;  // exposition only
  };

  template<class InputIterator~~, class Allocator = allocator<iter-value-type<InputIterator>>~~>
    vector(InputIterator, InputIterator~~, Allocator = Allocator()~~)
      -> vector<iter-value-type<InputIterator>~~, Allocator~~>;

  template<ranges::input_range R~~, class Allocator = allocator<ranges::range_value_t<R>>~~>
    vector(from_range_t, R&&~~, Allocator = Allocator()~~)
      -> vector<ranges::range_value_t<R>~~, Allocator~~>;
}
```

When we look at the class template definition, it should be no surprise that the main impact is on the constructors. As the container is itself responsible for allocating its members (and propagating the allocator), there are no allocator arguments to any other function member — the allocator of the container is implicit in each call, via `this`, and has been since C++98. Note that the copy and move constructors simplify, no longer needing a separate allocator-aware overload, and so avoiding template tricks to support deduction guides. Similarly, there is no longer an `explicit` constructor taking a single allocator argument as the allocator is not passed through the constructor argument list, and so the default constructor is unconditionally `noexcept`.

There are a couple of simplifications taking the type dependency out of type aliases, and the `get_allocator` function member is replaced by the (implicit) friend function `allocator_of`. The function member `swap` loses its exception specification, just like the free-function.

Finally, the deduction guides simplify as there is no optional allocator argument to the constructors they deduce from.

8 Open Questions

Several other approaches were considered and rejected by the authors. Here we record those ideas and the reasoning by which they were set aside.

8.1 Support for `union`s and `std::variant`

We would like to efficiently represent `union`s and `variant`s that comprise allocator-enabled types. This is relatively simple to solve for the case that all the alternatives are allocator-enabled, but in the case that some of the alternatives are not so enabled, we must store a copy of the allocator somewhere. In the ideal world, we would not store a redundant copy of the allocator when the active element is allocator-enabled though. It is expected that there will be some kind of extension facility for users to customize their allocator storage, and then overload the `allocator_of` friend function to return the right allocator in all circumstances.

8.2 Support for `std::optional`

The case for `optional` is the same as for `variant`, if we consider that `optional<T>` is fundamentally equivalent to `std::variant<T, std::monostate>`, a variant with one potentially allocator-enabled alternative and one non-allocator-enabled alternative.

8.3 Explicit Factory Functions

Sometimes there will be a desire to pass an allocator to be used within a factory function that does not return an allocator-enabled object, for example, a factory function returning a smart pointer. We expect there will be a need to supply an allocator-enabled overload to such an *explicit factory function*. This question is deliberately deferred until after the basic idea is proven successful though, as integrating with the world of overload resolution is anything but a trivial task.

8.4 `using` Beyond Initialization Expressions

There is clearly a desire to support `using` clauses more widely than just variable declarations, especially once explicit factory functions are available. However, there are a variety of concerns around consistency and ambiguity that easily derail a discussion, so this topic is deferred until the basic model is proven.

Control over temporaries created by subexpressions, parameters passed to functions, and allocators used within entire subexpressions all seem like interesting use cases that should be addressed eventually.

8.5 Move-only Scoped Types

The default implementation of initialization from an rvalue with a supplied allocator requires a type to be copy constructible. This is not always the case, for example:

```
struct MyPair {
   std2::vector<int>    first;
   std::unique_ptr<int> second;
};
```

Here we would like move-with-allocator construction to either move or make a copy of the `first` vector, per our current proposal. However, the `second` member is not allocator enabled, and cannot be copied, so we would like to move this member instead.

This problem brings us back to the issue of having overloads for a function, in this case the move constructor, both with and without the hidden `using` allocator, so is beyond the reach of the initial simplified model of this paper.

8.6 `constexpr` Allocation

C++20 added support for dynamic memory allocation within constant evaluation of functions by [P0784R7]. The customization point was specified in terms of `std::allocator_traits<A>::allocate`, rather than simply requiring use of `std::allocator<T>`, to allow for user supplied allocator types that might provide memory (during constant evaluation) that comes from a non-dynamic source, such as a static array of `std::byte`.

C++20 added the ability to determine whether a function was called at runtime or during constant evaluation by [P0595R2], allowing an allocator to dispatch dynamic allocation during constant evaluation, rather than at runtime, back to `std::allocator_traits<A>::allocate` if a dynamic allocation cannot be avoided.

C++20 added support for dynamic dispatch of virtual functions during constant evaluation by [P1064R0], which might appear to be the last hurdle to supporting polymorphic memory resources in constant evaluation. However, one last issue remains: 7.7 [expr.const]p6 requires that if the pointer supplied to `std::allocator_traits::construct_at` of type `T*` is to a region of dynamic storage, it must have been supplied by a call to `std::allocator<T>::allocate`. Therefore, we can support `constexpr` allocation with `pmr` allocators only if the stored objects are always of type `std::byte`, corresponding to the vocabulary pointer type of the `memory_resource` interface.

We would seek permission for `constexpr` allocation to support static casting from `std::byte *` to a region of storage suitable for storing objects of type `T` for the purposes of calling `construct_at` and `destroy_at`.

8.7 Scoped Types Beyond a Single Allocator Type

We have extensive experience applying the scoped model to allocators, and we have often seen the benefits of its use for reasoning about object lifetime and leveraging local memory arenas.

Support for more similarly scoped properties, along with the questions that would arise of how they might interoperate, seem like an inevitable requirement for this proposal.

The first such additional scoped property would likely be other allocator types. We're aware of libraries that use non-`pmr` polymorphic memory resource handles to manage, for example, GPU memory or similar pools that `pmr` does not lend itself to. Allowing this facility to support both types of allocators side by side would be beneficial.

In addition, we would expect a similar facility to benefit potential use for telemetry, logging, and execution resources. Tying objects to particular threads or cores seems like a particular case that might match the same model we use to tie objects to specific regions in memory. We hope to gather such additional use cases to have that guide the further evolution of our design.

8.8 Controlled Breaking of the Scoped Model

Allowing types to assume the guarantees of the scoped allocator model is our primary goal. Experience has taught us, however, that there are exceptions to every rule, and a facility by which the implicit rules can be subverted, especially for specific bases and members, seems like it will be necessary to cover all potential use cases.

9 Acknowledgements

Thanks to Michael Parks for the pandoc-based framework used to transform this document's source from Markdown.

Special thanks to Sean Baxter for providing extensive feedback and an early implementation, which made testing the code samples possible. Further thanks to all our reviewers, especially Mungo Gill, Pablo Halpern, and John Lakos who greatly improved the clarity of this paper, along with fixing many simple typos and grammatical mistakes.

10 References

[LWG2153] Robert Shearer. Narrowing of the non-member swap contract.
https://wg21.link/lwg2153

[LWG3758] Jiang An. Element-relocating operations of std::vector and std::deque should conditionally require Cpp17CopyInsertable in their preconditions.
https://wg21.link/lwg3758

[P0339R6] Pablo Halpern, Dietmar Kühl. 2019-02-22. polymorphic_allocator<> as a vocabulary type.
https://wg21.link/p0339r6

[P0595R2] Richard Smith, Andrew Sutton, Daveed Vandevoorde. 2018-11-09. std::is_constant_evaluated.
https://wg21.link/p0595r2

[P0784R7] Daveed Vandevoorde, Peter Dimov,Louis Dionne, Nina Ranns, Richard Smith, Daveed Vandevoorde. 2019-07-22. More constexpr containers.
https://wg21.link/p0784r7

[P1064R0] Peter Dimov, Vassil Vassilev. 2018-05-04. Allowing Virtual Function Calls in Constant Expressions.
https://wg21.link/p1064r0

[P1144R8] Arthur O'dwyer. 2023-05-15. Object relocation in terms of move plus destroy.
https://wg21.link/p1144r8

[P2025R2] Anton Zhilin. 2021-03-15. Guaranteed copy elision for return variables.
https://wg21.link/p2025r2

[P2786R1] Mungo Gill, Alisdair Meredith. 2023-05-15. Trivial Relocatability Options.
https://wg21.link/p2786r1

[P2814R0] Mungo Gill, Alisdair Meredith, Arthur O'dwyer. 2023-05-15. Trivial Relocatability — Comparing P1144 with P2786.
https://wg21.link/p2814r0