**Doc No:** P0339r1
**Date:** 2016-10-15
**Audience:** LEWG
**Authors:** Pablo Halpern, Intel Corp. <phalpern@halpernwightsoftware.com>
Dietmar Kühl <dkuhl@bloomberg.net>

# polymorphic_allocator<> as a vocabulary type

## Contents

1 Abstract ........ 1
2 Changes since R0 ........ 2
3 Motivation ........ 2
4 Proposal Overview ........ 4
5 Alternatives Considered ........ 6
6 Future directions ........ 6
7 Formal Wording ........ 6
  7.1 Document Conventions ........ 6
  7.2 Undo changes to uses-allocator construction ........ 7
  7.3 Remove `erased_type` from the TS ........ 7
  7.4 Changes to `std::experimental::function` ........ 7
  7.5 Changes to type-erased allocator ........ 9
  7.6 Definition of `polymorphic_allocator<>` ........ 10
  7.7 Changes to class template `promise` ........ 12
  7.8 Changes to class template `packaged_task` ........ 12
8 References ........ 13

## 1 Abstract

The `pmr::memory_resource` type, recently added to the C++17 working draft, provides a way to control the memory allocation for an object without affecting its compile-time type – all that is needed is for the object's constructor to accept a pointer to `pmr::memory_resource`. The `pmr::polymorphic_allocator<T>` adaptor class allows memory resources to be used in all places where allocators are used in the standard: uses-allocator construction, scoped allocators, type-erased allocators, etc.. For many classes, however, the `T` parameter does not make sense.

In this paper, we propose an explicit specialization of `pmr::polymrophic_allocator` for use as a vocabulary type. This type meets the requirements of an allocator in the standard, but is easier to use in contexts where it is not necessary or desirable to fix the allocator type at compile time. The use of `pmr::polymorphic_allocator<>` also simplifies the definition of *uses-allocator construction* in the TS and situations where allocator type-erasure would otherwise be used, including in `std::function`.

This proposal is targeted for the next release of the Library Fundamentals technical specification.

## 2 Changes since R0

The original version of this proposal was to use `polymorphic_allocator<void>` as a vocabulary type, instead of `polymorphic_allocator<>`. LEWG discussion in Oulu uncovered two related problems with the original proposal:

1. `void` is not a valid `value_type` for an allocator, so `polymorphic_allocator<void>` does not meet the allocator requirements.

2. Even if `void` were valid, its use here might conflict with the proposal to make `void` a regular type, P0146.

To correct these problems, we made the following changes:

- Instead of `polymorphic_allocator<void>`, use `polymorphic_allocator<>`, which is a shorthand for `polymorphic_allocator<char>`.

- Instead of hijacking `allocate` and `deallocate` for byte allocation, add new member functions, `allocate_bytes` and `deallocate_bytes`. This change also removed the need for creating an explicit specialization of `polymorphic_allocator`, as the `allocate_bytes` function can usefully be a member of all instantiations.

In addition, this proposal folds in the changes from P0335, which was applied to the C++17 WP in June, but was not applied to the LFTS.

## 3 Motivation

Consider the following class that works like `vector<int>`, but with a fixed maximum size determined at construction:

```
class IntVec {
    std::size_t m_size;
    std::size_t m_capacity;
    int *       m_data;
  public:
    IntVec(std::size_t capacity);
      : m_size(0), m_capacity(capacity), m_data(new int[capacity]) { }
    …
};
```

Suppose we want to add the ability to choose an allocator. One way would be to make the allocator type be a compile-time parameter:

```
template <class Alloc = std::allocator<int>> class IntVec …
```

But that has changed our simple class into a class template, and introduced all of the complexities of writing classes with allocators, including the use of `allocator_traits`. The constructor for this class template looks like this:

```
IntVec(std::size_t capacity, Alloc alloc = {} )
  : m_size(0), m_capacity(capacity), m_alloc(alloc)
  , m_data(std::allocator_traits<Alloc>::allocate(m_alloc, capacity)) { }
```

Our next attempt removes the templatization by using `pmr::memory_resource` to choose the allocation mechanism at run time instead of at compile time, thus avoiding the complexities of templates and ensuring that all `IntVec` objects are of the same type:

```
IntVec(std::size_t capacity,
       std::pmr::memory_resource *memrsrc = std::pmr::get_default_resource())
  : m_size(0), m_capacity(capacity), m_memrsrc(memrsrc)
  , m_data(memrsrc->allocate(capacity*sizeof(int), alignof(int)) { }
```

This solution works very well in isolation, but suffers from a number of drawbacks:

1. **Does not conform to the Allocator concept**

   The pointer type, `std::pmr::memory_resource*`, does not meet the requirements of an allocator, and so does not fit into the facilities within the standard designed for allocators, such as *uses-allocator construction* (section 20.9.7.2 in the standard working draft, N4582).

   The original proposal for `memory_resource`, N3916, included modifications to the definition of *uses-allocator* construction in order to address this deficiency. Those changes were not added to the C++17 working draft with the rest of the Fundamentals TS version 1

2. **Lack of reasonable value-initialization**

   The result of default-initialization of a pointer is indeterminate, and the result of value initialization is a null pointer, neither of which is a useful value for storing in the class. The programmer must explicitly call `std::pmr::get_default_resource()`, as shown above. It is easily forgotten and is verbose.

3. **Danger of null pointers**

   Any time you pass a pointer to a function, you must contend with the possibility of a null pointer. Either you forbid it (ideally with a precondition check or assert), or you handle it some special way (i.e., by substituting some default). Either way, there is a chance of error.

4. **Inadvertent reseating of the memory resource**

   Idiomatically, neither move assignment nor copy assignment of an object using an allocator or memory resource should move or copy the allocator or memory resource. With rare exceptions, the memory resource used to construct an object should be the one used for its entire lifetime. Changing the resource can result in a mismatch between lifetime of the resource and the lifetime of the object that uses it. Also, assigning to an element of a container would result in breaking the homogenous use of a single allocator for all elements of that container, which is crucial to safely and efficiently applying algorithms like sort that swap elements within the container. Raw pointers encourage blind moving or copying of member variables during assignment, which can be dangerous.

Issues 2, 3, and 4 were addressed by another paper, P0148, which proposed a new type that provided a default constructor, and which was not assignable, `memory_resource_ptr`. That proposal, however, was withdrawn in Jacksonville when we (the authors of that paper as well as the current one) discovered that there was a simpler and more complete solution possible without introducing a completely new type: by using `polymorphic_allocator`. That discovery was the genesis of this paper.

## 4 Proposal Overview

We observed that a `polymorphic_allocator` object, which is basically a wrapper around a `memory_resource` pointer, can be used just about anywhere that a raw `memory_resource` pointer can be used, but does not suffer from the drawbacks listed above. Consider a minor rewrite of the `IntVec` class (above):

```
class IntVec {
  public:
    typedef std::pmr::polymorphic_allocator<int> allocator_type;

  private:
    std::size_t    m_size;
    std::size_t    m_capacity;
    allocator_type m_alloc;
    int *          m_data;
  public:
    IntVec(std::size_t capacity, allocator_type alloc = {} );
      : m_size(0), m_capacity(capacity), m_alloc(alloc)
      , m_data(alloc.allocate(capacity)) { }
    …
};
```

Let's consider the deficiencies of using a raw `memory_resource` pointer, one by one, to see how this new approach compares to the previous one:

1. The definition of the `allocator_type` nested type and the constructor taking a trailing allocator argument allows `IntVec` to play in the world of *uses-allocator construction*, including being passed an allocator when inserted into a container that uses a `scoped_allocator_adaptor`.

2. Value-initializing the allocator causes the default memory resource to be used, simplifying the default allocator argument and reducing the chance of error. If `IntVec` had a default constructor, the allocator would, again, use the default memory resource, with no effort on the part of the programmer.

3. A `polymorphic_allocator` is not a pointer and cannot be null. Attempting to construct a `polymorphic_allocator` with a null pointer violates the preconditions of the `polymorphic_allocator` constructor. This contract can be enforced by a single contract assertion in the `polymorphic_allocator` constructor, rather than in every client.

4. P0335 proposes that the assignment operators for `polymorphic_allocator` should be deleted. If this recommendation is accepted, then the problem of accidentally reseating the allocator would no longer exist for `polymorphic_allocator`. The deleted assignment operators would prevent the incorrect assignment operations from being generated automatically, forcing the programmer to define them, hopefully with the correct semantics. See P0335 for more details.

The above list shows that `polymorphic_allocator` can be used idiomatically to good effect, but suffers from some usability issues. To begin, `polymorphic_allocator` is a template, when what is desired is a non-template vocabulary type. Also, in order to allocate objects of different types, it is necessary to rebind the allocator, a step backwards from direct use of `memory_resource`, which does not require rebinding. This paper proposes a default parameter for `polymorphic_allocator` so that `polymorphic_allocator<>` can be used as a ubiquitous type. It also adds certain features to conveniently expose the capabilities of the underlying `memory_resource` pointer.

In addition to normal allocator functions, the `polymorphic_allocator<>` proposed here provides the following features:

- Being completely specialized, `polymorphic_allocator<>` does not behave like a template, but like a class. This fact can prevent inadvertent template bloat in client types.

- It can allocate objects of any type without needing to use `rebind`. Allocating types other than `value_type` is common for node-based and other non-vector-like containers.

- It can allocate objects on any desired alignment boundary. For example, `VecInt` might choose to align its data array on a SIMD data boundary.

- It provides member functions to allocate and construct objects in one step.

- It provides a good alternative to type erasure for types that don't have an allocator template argument. Note that an important part of this proposal is to simplify `std::function` to avoid the problematic two-dimensional type erasure that has caused problems in the C++11 and C++14 standards. (The C++17 CD removes allocators from `std::function`, making it easier to add them back more simply in the future.)

In addition to the definition of `polymorphic_allocator<>` itself, we propose the following significant simplifications to the memory section of the Library Fundamentals TS:

- Remove changes to the definition of *uses-allocator construction* and the `uses_allocator` trait. (Section 2 of the TS is completely removed.)

- Rewrite the **Type-erased allocator** section in terms of `polymorphic_allocator<>` instead of `memory_resource*`. Eliminate the `erased_type` struct.

- Eliminate type-erased allocators from the `function` class template, replacing it with `polymorphic_allocator<>`. (Note that the type-erased allocator was not implemented by any major standard-library supplier.)

- Update `promise` and `packaged_task` to use the new type-erased allocator idiom.

## 5 Alternatives Considered

P0148 proposed a new type, `memory_resource_ptr`, which provided many of the benefits described for `polymorphic_allocator<>`. The `memory_resource_ptr` type did not, however, conform to *allocator requirements* and did less to smooth the integration of `memory_resource` into the allocator ecosystem than does `polymorphic_allocator<>`. P0148 was withdrawn in favor of this proposal.

It has been suggested that we create a new class instead of using `polymorphic_allocator<>`. However, such a type would need to behave like a `polymorphic_allocator` in every way, so the only benefit we saw was, perhaps, a shorter name. We'll leave it up to the user to create their own shortened `typedef`s, as desired.

Instead of using `char` as the default template parameter for `polymorphic_allocator<T>`, we could have used a unique tag type. This might have been a useful direction if we had created an explicit specialization for `polymorphic_allocator<`*tag_type*`>`, but experiments showed that it only complicated the standard language and implementation, with no significant benefit over the current proposal.

## 6 Future directions

We should consider using `polymorphic_allocator` in the interface to `std::experimental::any`.

## 7 Formal Wording

### 7.1 Document Conventions

All section names and numbers are relative to the **July 2016 draft of the Library Fundamentals TS, N4600**. Note that major sections of the TS have been moved into the C++17 WD. Section numbers are, therefore, subject to significant change.

> Existing working paper text is indented and shown in dark blue. Edits to the working paper are shown with ~~red strikeouts for deleted text~~ and green underlining for inserted text within the indented blue original text.

Comments and rationale mixed in with the proposed wording appears as shaded text.

Requests for LWG opinions and guidance appear with light (yellow) shading. It is expected that changes resulting from such guidance will be minor and will not delay acceptance of this proposal in the same meeting at which it is presented.

## 7.2 Undo changes to uses-allocator construction

Remove section 2 from the TS, which would have made changes to sections 20.7.7.1, [allocator.uses.trait] and 20.7.7.2 [allocator.uses.construction] of the standard.

## 7.3 Remove `erased_type` from the TS

Remove section 3.1 [utility] from the TS, which defines `struct erased_type`. The changes to type-erased allocators, below, make this `struct` no longer necessary.

## 7.4 Changes to `std::experimental::function`

In section 4.1 [header.functional.synop] of the TS, remove the specialization of `uses_allocator` from the end of the `<functional>` synopsis:

```
~~template<class R, class... ArgTypes, class Alloc>~~
~~struct uses_allocator<experimental::function<R(ArgTypes...)>, Alloc>;~~
```

In section 4.2 [func.wrap.func] of the TS, modify `allocator_type` and all of the constructors that take an allocator in `std::experimental::function`:

```
template<class R, class... ArgTypes>
class function<R(ArgTypes...)> {
public:
  typedef R result_type;
  typedef T1 argument_type;
  typedef T1 first_argument_type;
  typedef T2 second_argument_type;

  typedef ~~erased_type~~pmr::polymorphic_allocator<> allocator_type;

  function() noexcept;
  function(nullptr_t) noexcept;
  function(const function&);
  function(function&&);
  template<class F> function(F);
  ~~template<class A> ~~function(allocator_arg_t,
                             const ~~A~~allocator_type&) noexcept;
  ~~template<class A> ~~function(allocator_arg_t,
                             const ~~A~~allocator_type&, nullptr_t) noexcept;
  ~~template<class A> ~~function(allocator_arg_t,
                             const ~~A~~allocator_type&, const function&);
  ~~template<class A> ~~function(allocator_arg_t,
                             const ~~A~~allocator_type&, function&&);
  template<class F~~, class A~~> function(allocator_arg_t,
                                      const ~~A~~ allocator_type&, F);
```

replace `get_memory_resource()` with `get_allocator()`:

```
  ~~pmr::memory_resource* get_memory_resource();~~
  allocator_type get_allocator() const noexcept;
};
```

and remove the definition of `uses_allocator`:

~~`template<class R, class... ArgTypes, class Alloc>`~~
~~`struct uses_allocator<experimental::function<R(ArgTypes...)>, Alloc>`~~
~~`    : true_type { };`~~

In sections 4.2.1 [func.wrap.func.con] and 4.2.2 [func.wrap.func.mod], eliminate all references to type erasure and memory resources:

### 4.2.1 function construct/copy/destroy [func.wrap.func.con]

When a function constructor that takes a first argument of type `allocator_arg_t` is invoked, the second argument ~~is treated as a *type-erased allocator* (8.3)~~ shall be a polymorphic allocator (C++17 §20.12.3 [memory.polymorphic.allocator.class] or LFTS §8.6 [memory.polymorphic.allocator.class]). A copy of the allocator argument is used to allocate memory, if necessary, for the internal data structures of the constructed `function` object, otherwise `pmr::polymorphic_allocator<>{}` is used. If the constructor moves or makes a copy of a `function` object (C++14 §20.9), including an instance of the `experimental::function` class template, then that move or copy is performed by *using-allocator construction* with allocator ~~`get_memory_resource()`~~`get_allocator()`.

~~In the following descriptions, let *`ALLOCATOR_OF(f)`* be the allocator specified in the construction of `function f`, or `allocator<char>()` if no allocator was specified.~~

`function& operator=(const function& f);`

*Effects*: `function(allocator_arg,` ~~*`ALLOCATOR_OF(*this)`*~~`get_allocator(),`
`f).swap(*this);`

*Returns*: `*this`.

`function& operator=(function&& f);`

*Effects*: `function(allocator_arg,` ~~*`ALLOCATOR_OF(*this)`*~~`get_allocator(),`
`std::move(f)).swap(*this);`

*Returns*: `*this`.

`function& operator=(nullptr_t) noexcept;`

*Effects*: If `*this != nullptr`, destroys the target of this.

*Postconditions*: `!(*this)`. The ~~memory resource~~allocator returned by ~~`get_memory_resource()`~~`get_allocator()` after the assignment is equivalent to the ~~memory resource~~allocator before the assignment. [ *Note*: the address returned by get_memory_resource() might change — *end note* ]

*Returns*: `*this`.

`template<class F> function& operator=(F&& f);`

*Effects* `function(allocator_arg,` ~~*`ALLOCATOR_OF(*this)`*~~`get_allocator(),`
`std::forward<F>(f)).swap(*this);`

*Returns*: `*this`.

*Remarks*: This assignment operator shall not participate in overload resolution unless `declval<decay_t<F>&>()` is Callable (C++14 §20.9.11.2) for argument types `ArgTypes...` and return type `R`.

`template<class F> function& operator=(reference_wrapper<F> f);`

*Effects*: `function(allocator_arg,` ~~`ALLOCATOR_OF(*this)`~~`get_allocator(),` `f).swap(*this);`

*Returns*: `*this`.

### 4.2.2 function modifiers [func.wrap.func.mod]

`void swap(function& other);`

*Requires*: ~~`*this->get_memory_resource() == *other.get_memory_resource()`~~ `this->get_allocator() == other.get_allocator()`.

*Effects*: Interchanges the targets of `*this` and `other`.

*Remarks*: The allocators of `*this` and `other` are not interchanged.

Add a new section describing the `get_allocator()` function:

`allocator_type get_allocator() const noexcept;`

Returns: A copy of the allocator specified at construction, if any; otherwise a copy of `allocator_type{}` evaluated at the time of construction of this object.

## 7.5 Changes to type-erased allocator

Make the following changes to section 8.3 Type-erased allocator [memory.type.erased.allocator]:

### 8.3 Type-erased allocator [memory.type.erased.allocator]

A type-erased allocator is an allocator or memory resource, `alloc`, used to allocate internal data structures for an object `X` of type `C`, but where `C` is not dependent on the type of `alloc`. Once `alloc` has been supplied to `X` (typically as a constructor argument), a copy of `alloc` can be retrieved from `X` only ~~as a pointer `rptr` of static type `std::experimental::pmr::memory_resource*` (8.5)~~ via an object named (for exposition) `pmr_alloc` of type `pmr::polymorphic_allocator<>` (C++17 §20.12.3 [memory.polymorphic.allocator.class] or LFTS §8.6 [memory.polymorphic.allocator.class]). The process by which ~~`rptr`~~`pmr_alloc` is ~~computed~~initialized from `alloc` depends on the type of `alloc` as described in Table 13:

Table 13 — Initialization of type-erased allocator

| **If the type of `alloc` is** | **then ~~the value of `rptr` is~~** |
|---|---|
| non-existent — no `alloc` specified | ~~The value of `experimental::pmr::get_default_resource()` at the time of construction~~ `pmr_alloc` is value initialized. |
| `nullptr_t` | ~~The value of `experimental::pmr::get_default_resource()` at the time of construction~~ `pmr_alloc` is value initialized. |
| a pointer type convertible to `pmr::memory_resource*` | ~~`static_cast<experimental::pmr::memory_resource*>(alloc)`~~`pmr_alloc` is initialized with `alloc` |
| `pmr::polymorphic_allocator<U>` | `pmr_alloc` is initialized with `alloc`~~`.resource()`~~ |
| any other type meeting the ~~Allocator requirements (C++14 §17.6.3.5)~~ requirements for the Allocator parameter to `pmr::resource_adaptor` [memory.resource.adaptor.overview] | `pmr_alloc` is initialized with a pointer to a value of type ~~`experimental::`~~`pmr::resource_adaptor<A>` where `A` is the type of `alloc`. ~~`rptr`~~`pmr_alloc` remains valid only for the lifetime of `X`. |
| None of the above | The program is ill-formed. |

Additionally, class C shall meet the following requirements:

— `C::allocator_type` shall be ~~identical to~~ a specialization of `std::`~~`experimental::erased_type`~~`pmr::polymorphic_allocator`.

— ~~`X.get_memory_resource()`~~`X.get_allocator()` returns ~~`rptr`~~`pmr_alloc`.

## 7.6 Definition of `polymorphic_allocator<>`

In section 8.6.1 [memory.polymorphic.allocator.overview], modify the general definition of `polymorphic_allocator<Tp>` as follows. Note that this diverges from the C++17 CD but remains compatible with it:

```
template <class Tp = char>
class polymorphic_allocator {
  memory_resource* m_resource; // For exposition only

public:
  typedef Tp value_type;

  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& rhs) = ~~default~~delete;
```

This is a drive-by fix. P0335 has been applied to the C++17 WP, but should also have been applied to the LFTS.

```
  Tp* allocate(size_t n);
  void deallocate(Tp* p, size_t n);

  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>
    T* allocate_object(size_t n = 1);
  template <class T>
    void deallocate_object(T* p, size_t n = 1);

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

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

  // Specializations for pair using piecewise construction
  template <class T1, class T2, class... Args1, class... Args2>
    void construct(pair<T1,T2>* p, piecewise_construct_t,
                   tuple<Args1...> x, tuple<Args2...> y);
  template <class T1, class T2>
    void construct(pair<T1,T2>* p);
  template <class T1, class T2, class U, class V>
    void construct(pair<T1,T2>* p, U&& x, V&& y);
  template <class T1, class T2, class U, class V>
    void construct(pair<T1,T2>* p, const std::pair<U, V>& pr);
  template <class T1, class T2, class U, class V>
    void construct(pair<T1,T2>* p, pair<U, V>&& pr);

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

  // Return a default-constructed allocator (no allocator propagation)
  polymorphic_allocator select_on_container_copy_construction() const;

  memory_resource* resource() const;
};
```

Add descriptions for the new member functions in section 8.6.3 [memory.polymorphic.allocator.mem] (underline highlighting omitted for ease of reading):

```
void* allocate_bytes(size_t nbytes, size_t alignment = alignof(max_align_t));
```

*Returns*: `m_resource->allocate(nbytes, alignment)`.

```
void deallocate_bytes(void* p, size_t nbytes,
                      size_t alignment= alignof(max_align_t));
```

*Effects*: Equivalent to `m_resource->deallocate(p, nbytes, alignment)`.

*Throws*: Nothing.

```
template <class T>
  T* allocate_object(size_t n = 1);
```

*Effects:* Allocates memory suitable for holding an array of `n` objects of type `T`.

*Returns*: `static_cast<T*>(allocate_bytes(n*sizeof(T), alignof(T)))`.

```
template <class T>
  void deallocate_object(T* p, size_t n = 1);
```

*Effects*: Equivalent to `deallocate_bytes(p, n*sizeof(T), alignof(T))`.

```
template <class T, class CtorArgs...>
  T* new_object(CtorArgs&&... ctor_args);
```

*Effects*: Allocates and constructs an object of type `T` as if by

```
void* p = allocate_object<T>();
try {
    new (p) T(std::forward<CtorArgs>(ctor_args)...);
} catch (...) {
    m_resource->deallocate(p, sizeof(T), alignof(T));
    throw;
}
```

*Returns*: The address of the newly constructed object (i.e., `p`).

```
template <class T>
  void delete_object(T* p);
```

*Effects*: Equivalent to `p->~T(); deallocate(p, sizeof(T), alignof(T))`.

## 7.7 Changes to class template `promise`

Make the following changes to the class definition of `promise` in section 11.2 [futures.promise] of the TS, consistent with the change in type-erased allocators:

```
template <class R>
class promise {
public:
  typedef ~~erased_type~~polymorphic_allocator<> allocator_type;
…
  ~~pmr::memory_resource* get_memory_resource();~~
  pmr::polymorphic_allocator<> get_allocator() const noexecpt;
};
```

## 7.8 Changes to class template `packaged_task`

Make the following changes to the class definition of `packaged_task` in section 11.3 [futures.task], consistent with the change in type-erased allocators:

```
    template <class R, class... ArgTypes>
    class packaged_task<R(ArgTypes...)> {
    public:
      typedef ~~erased_type~~polymorphic_allocator<> allocator_type;
  …
      ~~pmr::memory_resource* get_memory_resource();~~
      pmr::polymorphic_allocator<> get_allocator() const noexecpt;
    };
```

## 8 References

N4600 *Working Draft, C++ Extensions for Library Fundamentals, Version 2*, Geoffrey Romer, editor, 2016-07-01.

N3916 *Polymorphic Memory Resources - r2*, Pablo Halpern, 2014-02-14.

P0148 *`memory_resource_ptr`: A Limited Smart Pointer for `memory_resource` Correctness*, Pablo Halpern and Dietmar Kühl, 2015-10-14.

P0335 *Delete `operator=` for `polymorphic_allocator`*, Pablo Halpern, 2016-05.