**Doc No:**

**Date:** 2017-05-29

**Audience:** LEWG

**Authors:** Pablo Halpern, Intel Corp. <[phalpern@halpernwightsoftware.com](mailto:phalpern@halpernwightsoftware.com)>

Dietmar Kühl <[dkuhl@bloomberg.net](mailto:dkuhl@bloomberg.net)>

polymorphic\_allocator<> as a vocabulary type

**Contents**

1 Abstract 1

2 Changes 2

2.1 Changes since R2 2

2.2 Changes since R1 2

2.3 Changes since R0 2

3 Motivation 2

4 Proposal Overview 4

5 Alternatives Considered 6

6 Future directions 6

7 Formal Wording 7

7.1 Document Conventions 7

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 13

8 References 13

# 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.

# Changes

## Changes since R2

Changed polymorphic\_allocator<char> to polymorphic\_allocator<byte>.

Rebased C++17 references to the C++17 DIS.

Fixed bugs in new\_object() and delete\_object() member functions.

## Changes since R1

Minor changes, mostly taking into related proposals that have been accepted since R0.

## 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](http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0146r1.html).

To correct these problems, we made the following changes:

* Instead of polymorphic\_allocator<void>, use polymorphic\_allocator<>, which is a shorthand for polymorphic\_allocator<byte>.
* 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](http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0335r0.html), which was applied to the C++17 WP in June, but was not applied to the LFTS.

# 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 23.10.7.2 [allocator.uses.construction] in the C++17 DIS, N4660).

The original proposal for memory\_resource, [N3916](http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2014/n3916.pdf), 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.

1. **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.

1. **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.

1. **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 the 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](http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2015/p0148r0.pdf), 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.

# Proposal Overview

We observed that a polymorphic\_allocator object, which is nothing more than 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:

**using allocator\_type = std::pmr::polymorphic\_allocator<int>;**

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](http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0335r0.html), which was accepted in Oulu for C++17, deleted the assignment operators for polymorphic\_allocator. Thus, the problem of accidentally reseating the allocator no longer exists 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](http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0335r0.html) 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:

* Because polymorphic\_allocator<> *is* an allocator, and does not require special handling, we back out changes to the definition of *uses-allocator construction* and the uses\_allocator trait that are present in the current draft of the LFTS. (Section 2 of the TS is completely removed.)
* Rewrite the **Type-erased allocator** section in terms of polymorphic\_allocator<> instead of memory\_resource\* and eliminate the erased\_type struct.
* Eliminate the type-erased allocator 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.

# Alternatives Considered

[P0148](http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2015/p0148r0.pdf) 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 aliases, as desired.

Instead of using byte 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 earlier drafts of this proposal proved to us that it only complicated the standard language and implementation, with no significant benefit over the current proposal.

# Future directions

We should consider using polymorphic\_allocator<> in the interface to std::experimental::any.

# Formal Wording

## Document Conventions

All section names and numbers are relative to the November 2016 draft of the Library Fundamentals TS, [N4617](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/n4617.pdf). Note that major sections of the TS have been moved into the C++17 WD. Section numbers are, therefore, subject to significant change in the future.

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.

## Undo changes to uses-allocator construction

Remove section 2 from the TS, which would have made changes to sections 23.10.7.1, [allocator.uses.trait] and 23.10.7.2 [allocator.uses.construction] of the standard.

## 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.

## 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:

using result\_type = R;

using argument\_type = T1;

using first\_argument\_type = T1;

using second\_argument\_type = T2;

using allocator\_type = erased\_typepmr::polymorphic\_allocator<>;

function() noexcept;

function(nullptr\_t) noexcept;

function(const function&);

function(function&&);

template<class F> function(F);

template<class A> function(allocator\_arg\_t,

const Aallocator\_type&) noexcept;

template<class A> function(allocator\_arg\_t,

const Aallocator\_type&, nullptr\_t) noexcept;

template<class A> function(allocator\_arg\_t,

const Aallocator\_type&, const function&);

template<class A> function(allocator\_arg\_t,

const Aallocator\_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 §23.12.3 [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 resourceallocator returned by get\_memory\_resource()get\_allocator() after the assignment is equivalent to the memory resourceallocator before the assignment. [ *Note*: the address returned by get\_memory\_get\_allocator().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.

## 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 §23.12.3 [memory.polymorphic.allocator.class] or LFTS §8.6 [memory.polymorphic.allocator.class]). The process by which rptrpmr\_alloc is computedinitialized 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. rptrpmr\_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\_typepmr::polymorphic\_allocator.
* X.get\_memory\_resource()X.get\_allocator() returns rptrpmr\_alloc.

## 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 = byte>

class polymorphic\_allocator {

memory\_resource\* m\_resource; // For exposition only

public:

using value\_type = Tp;

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) = defaultdelete;

This is a drive-by fix. [P0335](http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0335r0.html) 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 {

construct(p, 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 destroy(p); deallocate(p, sizeof(T), alignof(T)).

## 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:

using allocator\_type = erased\_typepolymorphic\_allocator<>;

...

pmr::memory\_resource\* get\_memory\_resource();

pmr::polymorphic\_allocator<> get\_allocator() const noexecpt;

};

## 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:

using allocator\_type = erased\_typepolymorphic\_allocator<>;

...

pmr::memory\_resource\* get\_memory\_resource();

pmr::polymorphic\_allocator<> get\_allocator() const noexecpt;

};

# References

[N4617](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/n4617.pdf) *Draft Technical Specification, C++ Extensions for Library Fundamentals, Version 2*, Geoffrey Romer, editor, 2016-11-28.

[N3916](http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2014/n3916.pdf) *Polymorphic Memory Resources - r2*, Pablo Halpern, 2014-02-14.

[P0148](http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2015/p0148r0.pdf) *memory*\_*resource*\_*ptr: A Limited Smart Pointer for memory*\_*resource Correctness*, Pablo Halpern and Dietmar Kühl, 2015-10-14.

[P0335](http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0335r0.html) *Delete operator= for polymorphic\_allocator*, Pablo Halpern, 2016-05.