**Doc No:** P0339r6

**Date:** 2019-02-22

**Audience:** LWG

**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 Change History 1

2.1 Changes since R5 1

2.2 Changes since R4 2

2.3 Changes since R3 2

2.4 Changes since R2 2

2.5 Changes since R1 2

2.6 Changes since R0 2

3 Motivation 3

4 Proposal Overview 4

5 Before and After 6

6 Alternatives Considered 8

7 Formal Wording 8

7.1 Document Conventions 8

7.2 Definition of polymorphic\_allocator<> 9

8 References 10

# Abstract

The pmr::memory\_resource type 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.

This proposal is targeted for the C++ working paper.

# Change History

## Changes since R5

LWG has forwarded R6 for straw polls for inclusion in C++20 (Feb 21, 2019).

Rebased to the Jan 2019 working draft, N4800.

The allocate\_object function now tests for length error.

Other minor rewordings as per LWG review in Kona, Feb 2019.

## Changes since R4

Fixed some incorrect references to P0978 that should have been P0987.

## Changes since R3

The changes to pmr::polymorphic\_allocator have been retargeted to the C++20 working paper. The other changes (to function, promise, and packaged\_task) have been split into a separate paper ([P098](https://wg21.link/P0987)7), which is targeted at the next Library TS.

Forwarded from LEWG to LWG in Jacksonville, March 2018.

## 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 [P0337](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0337r0.html), which was applied to the C++17 WP in June 2016, 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 would have been 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 in 2016 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. The assignment operators for polymorphic\_allocator are deleted. Thus, the problem of accidentally reseating the allocator does not 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](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. See [P0148](http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2015/p0148r0.pdf) for examples of avoiding allocator type-erasure in std::function, std::promise, and std::packaged\_task.

# Before and After

The following example shows the part implementation and use of a simple list-of-string class. The code on the left (before), shows the use of the fully-general allocator model. The code on the right (after) shows the use of (hard-coded) pmr::polymorphic\_allocator<>. In both cases, exception-safety code in push\_front is omitted for simplicity. Although the code on the left is more general and closer to standard library code, the code on the right is sufficient for probably 80% of programmers who wish to add the benefits of allocators to their classes. As you can see, it is much simpler and less error-prone. Of particular note:

* The list class on the right is not a template
* There is no use of std::allocator\_traits.
* There is no need to do any rebinding
* Large chunks of boiler-plate code is unnecessary.

|  |  |
| --- | --- |
| **Before** | **After** |
| template <class Alloc = std::allocator<std::string>>  class StringList1  {  **using alloc\_traits =**  **std::allocator\_traits<Alloc>;**  public:  using allocator\_type = Alloc;  using value\_type =  std::basic\_string<char,  std::char\_traits<char>,  **typename alloc\_traits::**  **template rebind\_alloc<char>**>;  // It is easy to get the allocator's  // value\_type type wrong! Check it!  static\_assert(std::is\_same<  typename Alloc::value\_type,  value\_type>::value,  "Alloc::value\_type is incorrect");  private:  struct node {  node\* m\_next = nullptr;  union {  // Non-initialized member  value\_type m\_value;  };  };  **using node\_alloc =**  **typename alloc\_traits::**  **template rebind\_alloc<node>;**  node\_alloc m\_alloc;  node \*m\_head = nullptr;  node \*m\_tail = nullptr;  public:  StringList1(const allocator\_type& a =  {})  : m\_alloc(a)  , m\_head(nullptr) { }  void push\_front(const value\_type& v) {  **using alloc\_node\_traits =**  **typename alloc\_traits::**  **template rebind\_traits<node>;**  node \*n = **alloc\_node\_traits::**  allocate(m\_alloc, 1);  // NOTE: Exception safety elided  **alloc\_node\_traits::**  construct(m\_alloc, &n->m\_value,v);  n->m\_next = m\_head;  m\_head = n;  if (! m\_tail)  m\_tail = n;  }  // ...  }; | // List of strings using  // polymorphic\_allocator<>  class StringList2  {  public:  using allocator\_type =  std::pmr::polymorphic\_allocator<>;  using value\_type =  std::pmr::string;  private:  struct node {  node\* m\_next = nullptr;  union {  // Non-initialized member  value\_type m\_value;  };  };  allocator\_type m\_alloc;  node \*m\_head = nullptr;  node \*m\_tail = nullptr;  public:  StringList2(const allocator\_type& a =  {})  : m\_alloc(a)  , m\_head(nullptr) { }  void push\_front(const value\_type& v) {  node \*n =  m\_alloc.allocate\_object<node>();  // NOTE: Exception safety elided  m\_alloc.construct(&n->m\_value, v);  n->m\_next = m\_head;  m\_head = n;  if (! m\_tail)  m\_tail = n;  }  // ...  }; |
| int main()  {  using SaString =  std::basic\_string<char,  std::char\_traits<char>,  SimpleAlloc<char>>;  SimpleAlloc<SaString> sa;  StringList1<**SimpleAlloc<SaString>**>  slst1(sa);  slst1.push\_front("hello");  } | int main()  {  SimpleResource sr;  StringList2 slst2(&sr);  slst2.push\_front("goodbye");  } |

# Alternatives Considered

In Jacksonville, LEWG considered changing some or all of the proposed new member functions for polymorphic\_allocator to free functions, instead. The allocate/deallocate\_object and new/delete\_object functions, in particular, could be implemented for any allocator type, not just polymorphic\_allocator. There was, however, insufficient consensus for this change.

[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.

# Formal Wording

## Document Conventions

All section names and numbers are relative to the January 2019 C++ Working Paper, N4800.

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.

## Definition of polymorphic\_allocator<>

In section 19.12.3 [mem.poly.allocator.class], modify the general definition of polymorphic\_allocator<Tp> as follows. Note that this diverges from the WP but remains compatible with it:

template <class Tp = byte>

class polymorphic\_allocator {

memory\_resource\* memory\_rsrc; // *exposition only*

public:

using value\_type = Tp;

// *19.12.3.1, constructors*

polymorphic\_allocator() noexcept;

polymorphic\_allocator(memory\_resource\* r);

polymorphic\_allocator(const polymorphic\_allocator& other) = default;

template <class U>

polymorphic\_allocator(const polymorphic\_allocator<U>& other) noexcept;

polymorphic\_allocator&

operator=(const polymorphic\_allocator& rhs) = delete;

// *19.12.3.2, member functions*

[[nodiscard]] 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... CtorArgs>

T\* new\_object(CtorArgs&&... ctor\_args);

template <class T>

void delete\_object(T\* p);

template <class T, class... Args>

void construct(T\* p, Args&&... args);

template <class T>

void destroy(T\* p);

polymorphic\_allocator select\_on\_container\_copy\_construction() const;

memory\_resource\* resource() const;

};

Add descriptions for the new member functions in section 19.12.3.2 [mem.poly.allocator.mem] (underline highlighting omitted for ease of reading):

void\* allocate\_bytes(size\_t nbytes, size\_t alignment = alignof(max\_align\_t));

*Effects*: Equivalent to: return memory\_rsrc->allocate(nbytes, alignment);

*Note:* The return type is void\* (rather than, e.g., byte\*) to support conversion to an arbitrary pointer type U\* by static\_cast<U\*>, thus facilitating construction of a U object in the allocated memory.

void deallocate\_bytes(void\* p, size\_t nbytes,  
 size\_t alignment = alignof(max\_align\_t));

*Effects*: Equivalent to memory\_rsrc->deallocate(p, nbytes, alignment).

template <class T>  
  T\* allocate\_object(size\_t n = 1);

*Effects:* Allocates memory suitable for holding an array of n objects of type T, as follows:

* if SIZE\_MAX / sizeof(T) < n, throws length\_error,
* otherwise equivalent to: return static\_cast<T\*>(allocate\_bytes(n\*sizeof(T), alignof(T)));

*Note*: T is not deduced and must therefore be provided as a template argument.

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 follows: equivalent to:

T\* p = allocate\_object<T>();

try {

construct(p, std::forward<CtorArgs>(ctor\_args)...);  
} catch (...) {  
 deallocate\_object(p);

throw;

}

return p;

*Note*: T is not deduced and must therefore be provided as a template argument.

template <class T>

void delete\_object(T\* p);

*Effects*: Equivalent to:

destroy(p);  
deallocate\_object(p);

# References

[P0987](https://wg21.link/P0987) *polymorphic\_allocator<byte> instead of type-erasure*, Pablo Halpern, 2018-04-01.

[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.