**Doc No:**

**Date:** 2016-05-22

**Audience:** LEWG

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

Dietmar Kühl <<mailto:dkuhl@bloomberg.net>>

polymorphic\_allocator<void> as a vocabulary type

**Contents**

1 Abstract 1

2 Change History 1

3 Motivation 1

4 Proposal Overview 1

5 Alternatives Considered 1

6 Formal Wording 1

6.1 Document Conventions 1

7 References 2

8 Acknowledgements 2

# 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 in its constructor. 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 a compile-time allocator is not necessary or desirable. The use of pmr::polymorphic\_allocator<void> also simplifies the definition of *uses-allocator construction* in the TS and also simplifies 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.

# 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 adhere 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\_resoure, [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 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 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 proposal, [P0148](http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2015/p0148r0.pdf), which proposed a new type, memory\_resource\_ptr, that provided a default constructor and which was not assignable. However, this proposal was withdrawn in Jacksonville when we (the authors of that paper) discovered that there was a simpler and more complete solution possible without introducing a completely new type: the use of polymorphic\_allocator. That discovery was the genesis of this paper.

# 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 approach compares:

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) proposes that the assignment operators for polymorphic\_allocator should be deleted. If it is accepted, then the problem of accidentally reseating the allocator would not exist for polymorphic\_allocator. The IntVec class above would, by default, have deleted copy and move assignment operators. This would prevent the incorrect assignment operations from being generated automatically, forcing the programmer to define them, hopefully with the correct semantics. See P0035 for more details.

The above list shows that polymorphic\_allocator can be used idiomatically to good effect. The novel feature of this paper is not this idiomatic use, therefore, but a new specialization for polymorphic\_allocator<void>. Unlike std::allocator<void>, which does not actually meet the requirements of an allocator, polymorphic\_allocator<void> is designed to be a complete allocator type. It is similar to polymorphic\_allocator<char>, but has certain features to conveniently expose the features of the underlying memory\_resource pointer.

In addition to normal allocator functions, polymorphic\_allocator<void> provides the following features:

* Being completely specialized, polymorphic\_allocator<void> 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 that 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 can be used as an argument to placement new 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. Part of this proposal is to simplify std::function to avoid the problematic two-dimensional type erasure that has caused problems since C++11.

In addition to the definition of polymorphic\_allocator<void> itself, we propose a few simplifications to the memory section of the Library Fundamentals TS:

* Remove changes to the definition of *uses-allocator construction* (but keep the changes to the uses\_allocator trait).
* Rewrite the **Type-erased allocator** section in terms of polymorphic\_allocator<void> instead of memory\_resource\*.
* Eliminate type-erased allocators from the function class template, replacing it with polymorphic\_allocator<void>. (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<void>. However, the memory\_resource\_ptr type did not

# Formal Wording

## Document Conventions

All section names and numbers are relative to the <<source document here>>.

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.

# References

[N4584](http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2016/n4584.html) Working draft, Library Fundamentals v2

[N3916](http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2014/n3916.pdf) polymorphic memory resources

[P0148](http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2015/p0148r0.pdf) memory\_resource\_ptr

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

# Acknowledgements