Doc No: P0339r5
Date: 2018-10-07
Audience: LWG
Authors: Pablo Halpern, Intel Corp. <phalpern@halpernwightsoftware.com>
Dietmar Kühl <dkuhl@bloomberg.net>

polymorphic_allocator<> as a vocabulary type

Contents

1 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.

2 Change History

2.1 Changes since R4

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

2.2 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 (P0987), which is targeted at the next Library TS.

2.3 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.

2.4 Changes since R1

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

2.5 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<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, which was applied to the C++17 WP in June 2016, 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 23.10.7.2 [allocator.uses.construction] in the C++17 DIS, N4660).

   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 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](), 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.

4 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] 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 for examples of avoiding allocator type-erasure in `std::function`, `std::promise`, and `std::packaged_task`.

5 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

```
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;
    };
```

After

```
// 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;
    };
  };
```

<table>
<tr>
<td>

```
  };

  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;
  }

  // ...
};
```

</td>
<td>

```





  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;
  }

  // ...
};
```

</td>
</tr>
<tr>
<td>

```
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");
}
```

</td>
<td>

```
int main()
{




 SimpleResource sr;

 StringList2 slst2(&sr);
   slst2.push_front("goodbye");
}
```

</td>
</tr>
</table>

6 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 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.

7 Formal Wording

7.1 Document Conventions

All section names and numbers are relative to the **February 2018 C++ Working Paper, N4727.**

> 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 <ins>green underlining for inserted text</ins> 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 Definition of `polymorphic_allocator<>`

In section 23.12.3 [mem.poly.allocator.class], 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; // exposition only

public:
  using value_type = Tp;

  // 23.12.3.1, 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;

  // 23.12.3.2, 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);

  // 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);

  polymorphic_allocator select_on_container_copy_construction() const;

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

Add descriptions for the new member functions in section 23.12.3.2 [mem.poly.allocator.mem] (underline highlighting omitted for ease of reading):

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

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)))`.

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 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`).

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)`.

8 References

[P0987](#) *polymorphic_allocator<byte> instead of type-erasure*, Pablo Halpern, 2018-04-01.

[N4617](#) *Draft Technical Specification, C++ Extensions for Library Fundamentals, Version 2*, Geoffrey Romer, editor, 2016-11-28.

[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.