polymorphic_allocator<byte> instead of type-erasure

Abstract

Type-erased allocators have been proposed in the Library Fundamentals Technical Specification working draft as a way to add allocator customization to types such as std::function that do not have allocators as part of their type (i.e., we specify the allocator type on construction, not when instantiating the type). Type erasure of allocators is somewhat complex and inefficient for implementers, especially when combined with erasure of other types in the constructor (2-dimensional type erasure), as would be the case for std::function. This paper proposes replacing type-erased allocators in the LFTS WP with the use of std::pmr::polymorphic_allocator<byte>, consistent with the proposed use of polymorphic_allocator as a vocabulary type, proposed in P0339.

This paper is split off from P0339r3, which proposes polymorphic_allocator<byte> as a vocabulary type. While P0339r4 contains those portions of P0339r3 targeted for the C++ working draft, this proposal contains those portions of P0339r3 that are targeted for the next release of the Library Fundamentals technical specification.

History

This paper was formerly part of P0339, which proposed extensions to polymorphic_allocator so that it can more easily be used as a vocabulary type. At the March 2018 Jacksonville meeting, LEWG voted to split P0339r3 into two parts: one part to be...
targeted to C++20 ([P0339r4]), and the other part to be targeted to the next LFTS (this paper). LEWG also voted to advance both papers to LWG without further LEWG review.

3 Motivation

The current definition of `std::function` in the C++17 standard does not allow the user to supply an allocator to control memory allocation despite the fact that it sometimes allocates memory and that the C++14 standard had a (broken and never implemented) interface for supplying an allocator. The LFTS defines a version of `function` that does take an allocator argument at construction and uses type erasure to hold that allocator. The main constructor, as it appears currently in the LFTS looks like this:

```cpp
template<class F, class A>
function(allocation_arg_t, const A&, F);
```

Note that both F and A are template parameters to the constructor that do not appear in the class type. This means that the implementation of `function` needs to do two-dimensional type erasure, which is both complicated and can be inefficient. The LFTS specification for type-erased allocators is also somewhat complicated by the desire to have type-erased objects place nicely in the realm of other objects that take allocator parameters.

The proposed revision of the above constructor looks like this:

```cpp
template<class F>
function(allocation_arg_t, const polymorphic_allocator<byte>&, F);
```

Note that the allocator is no longer a template argument, which simplifies specification and copying of the allocator, and provides the ability to return the allocator to the client using a straight-forward interface consistent with other allocator-savvy types:

```cpp
polymorphic_allocator<byte> get_allocator() const noexcept;
```

4 Proposal Overview

Consistent with the use of `polymorphic_allocator<>` as a vocabulary type in P0339, this paper proposes the following significant simplifications to the memory section of the Library Fundamentals TS:

- **Because** `polymorphic_allocator<byte>` *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<byte>` 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<byte>`. *(Note that the type-erased allocator for function was not implemented by any major standard-library supplier.)*

- **Update** `promise` and `packaged_task` to use the new type-erased allocator idiom.
5 Future directions

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

6 Formal Wording

6.1 Document Conventions

All section names and numbers are relative to the **November 2016 draft of the Library Fundamentals TS, N4617**. Note that major sections of the TS have been moved into C++17. 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.

6.2 Feature test macros

Modify selected rows from Table 2 in section 1.6 [general.feature.test] as follows:

<table>
<thead>
<tr>
<th>Doc. No.</th>
<th>Title</th>
<th>Primary Section</th>
<th>Macro Name Suffix</th>
<th>Value</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>N3916 P0987R0</td>
<td>Type-erased Polymorphic allocator for function</td>
<td>4.2</td>
<td>function_<code>erased</code>polymorphic_allocator</td>
<td>201406 201804</td>
<td><code>&lt;experimental/functional&gt;</code></td>
</tr>
<tr>
<td>N3916 P0987R0</td>
<td>Type-erased Polymorphic allocator for promise</td>
<td>11.2</td>
<td>promise_<code>erased</code>polymorphic_allocator</td>
<td>201406 201804</td>
<td><code>&lt;experimental/future&gt;</code></td>
</tr>
<tr>
<td>N3916 P0987R0</td>
<td>Type-erased Polymorphic allocator for packaged_task</td>
<td>11.3</td>
<td><code>package_task_</code>erased<code>polymorphic_allocator</code></td>
<td>201406 201804</td>
<td><code>&lt;experimental/future&gt;</code></td>
</tr>
</tbody>
</table>

6.3 Undo changes to uses-allocator construction

Remove section 2.1 [mods.allocator.uses] 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. Note that this change, applied to N4617 would make section 2 [mods] empty, so that section can be completely removed unless some other material is added before adoption of this paper.
6.4 Remove erased_type from the TS

Remove section section 3.1.1 [utility.synop], which introduces an `<experimental/utility>` header, and section 3.1.2 [utility.erased.type], which defines struct erased_type, from the TS draft. The changes to type-erased allocators, below, make this struct no longer necessary. Note that removing these two sections from N4617 would make section 3.1 [utility] empty, and thus it, too, can be removed.

6.5 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_type<pmr::polymorphic_allocator<byte>>;  

function() noexcept;  
function(nullptr_t) noexcept;  
function(const function&);  
function(function&&);  
template<class F> function(F);  
template<class A> function(allocation_arg_t, 
const Allocator_type& ) noexcept;  
template<class A> function(allocation_arg_t, 
const Allocator_type& , nullptr_t ) noexcept;  
template<class A> function(allocation_arg_t, 
const Allocator_type&, const function&);  
template<class A> function(allocation_arg_t, 
const Allocator_type&, function&&);  
template<class F, class A> function(allocation_arg_t, 
const 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 and a second argument of type polymorphic_allocator<byte> is invoked, the second argument is treated as a type-erased allocator (8.3) 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<byte>{} 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().

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.

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

**Effects:**

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

**Returns:** *this.

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

**Effects:**

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

**Returns:** *this.

```cpp
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().resource() might change — end note ]

**Returns:** *this.

```cpp
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...

```cpp
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() and
            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.

6.6 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<byte>(C++17 §23.12.3
 [memory.polymorphic_allocator_class]). The process by which rptr = pmr_alloc is computed is described in Table 13:

Table 13 — Initialization of type-erased allocator

<table>
<thead>
<tr>
<th>If the type of alloc is</th>
<th>then the value of rptr is</th>
</tr>
</thead>
</table>
| 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 |
polymorphic_allocator<> instead of type-erasure

pmr::polymorphic_allocator<U>

pmr_alloc is initialized with alloc_resource()

any other type meeting the Allocator requirements (C++14 §17.6.3.3)
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. *ptr_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 *ptr_pmr_alloc.

6.7 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:

```cpp
template <class R>
class promise {
public:
    using allocator_type = erased_type pmr::polymorphic_allocator<bryte>;
    ...
    pmr::memory_resource* get_memory_resource();
    allocator_type get_allocator() const noexecpt;
};
```

6.8 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:

```cpp
template <class R, class... ArgTypes>
class packaged_task<R(ArgTypes...)> { 
public:
    using allocator_type = erased_type pmr::polymorphic_allocator<bryte>;
    ...
    pmr::memory_resource* get_memory_resource();
    allocator_type get_allocator() const noexecpt;
};
```

7 References

P0039r4 polymorphic_allocator<> as a vocabulary type, Pablo Halpern & Dietmar Kühl, 2018-04-01.

polymorphic_allocator<byte> instead of type-erasure