Doc No: P0987r2
Date: 2022-10-12
Audience: LWG
Authors: Pablo Halpern <phalpern@halpernwightsoftware.com>

polymorphic_allocator<> instead of type-erasure

Contents

1 Abstract 1
2 Related issues 1
3 History 2
4 Motivation 2
5 Proposal Overview 3
6 Future directions 3
7 Formal Wording 3
7.1 Document Conventions 3
7.2 Feature test macros 4
7.3 Undo changes to uses-allocator construction 4
7.4 Remove all mention of *type-erased allocator* from the TS 4
7.5 Changes to `std::experimental::function` 4
8 References 7

1 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 removing type-erased allocators from the LFTS WP and, for `experimental::function`, replacing them with the use of `std::pmr::polymorphic_allocator<>`, consistent with the use of `polymorphic_allocator` as a vocabulary type (see P0339, which was adopted into C++20).

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.

2 Related issues

Adoption of this paper would resolve LWG issue 2564 by removing type erasure and, thus, allowing the `noexcept` constructors to remain `noexcept`.

Adoption of this paper would resolve a small part of LWG issue 3411 by removing a few sections touched by the proposed resolution of issue 3411.

3 History

Changes from R0 to R1

- Rebased section numbers, etc., onto the latest LFTS and C++20.
- Use `std::pmr::polymorphic_allocator<>` instead of `std::pmr::polymorphic_allocator<byte>` in most cases.
- Added wording to remove all mention of type-erased allocators from the TS and removed wording that attempted (but failed) to give type-erased allocators a uniform pmr interface. This change should avoid an NB comment that would otherwise be guaranteed.
- Corrected wording for `experimental::function`, especially wrt selection of allocator on construction. Also updated the language to use *preconditions* and *constraints* instead of *requires.*
- Removed wording that tweaked the existing (incorrect) interfaces to `experimental::promise` and `experimental::packaged_task` from the TS. With these changes, all futures-related extensions are removed from the TS.

Prior to R0

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. P0339r6 was ultimately accepted into the C++20 standard.

4 Motivation

The current definition of `std::function` in the C++20 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:

```
template<class F, class A>
  function(allocator_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:

```
template<class F>
  function(allocator_arg_t, const polymorphic_allocator<>&, 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:

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

5 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<>` *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.)

- Eliminate the **Type-erased allocator** section from the TS. A type using type-erased allocators according to the existing LFTS would be forced to create a `resource_adaptor` on the heap, and providing an interface by which it could be accessed. Unfortunately, once made available to clients, the lifetime of the `resource_adaptor` cannot be specified in such a way as to make it safely usable.

- Eliminate the type-erased allocator from the `function` class template, replacing it with `polymorphic_allocator<>`. (Note that the type-erased allocator for `function` was not implemented by any major standard-library supplier.)

- Remove `expermental::promise` and `expermental::packaged_task`, which existed solely to use the new, but ill-conceived ability to make the type-erased allocator visible to clients.

6 Future directions

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

7 Formal Wording

7.1 Document Conventions

All section names and numbers are relative to the **August 2022 draft of the Library Fundamentals TS, N4920** and the **C++20 standard** (DIS at N4860).

> 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 <u>green underlining for inserted text</u> 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 Feature test macros

Modify selected rows from Table 2 in section 1.5 [general.feature.test] as follows:

Table 2 — Significant features in this technical specification

Doc. No.	Title	Primary Section	Macro Name Suffix	Value	Header
~~N3916~~ P0987R1	~~Type-erased~~ Polymorphic allocator for `std::function`	4.2	`function_`~~`erased`~~`polymorphic_allocator`	~~201406~~ 202211	`<experimental/functional>` ~~`<experimental/utility>`~~
~~N3916~~	~~Type-erased allocator for `std::promise`~~	~~7.2~~	~~`promise_erased_allocator`~~	~~201406~~	~~`<experimental/future>` `<experimental/utility>`~~
~~N3916~~	~~Type-erased allocator for `std::packaged_task`~~	~~7.3~~	~~`packaged_task_erased_allocator`~~	~~201406~~	~~`<experimental/future>` `<experimental/utility>`~~

7.3 Undo changes to uses-allocator construction

Remove section 2.2 [mods.allocator.uses] from the TS, which would have made changes to sections 20.10.8.1, [allocator.uses.trait] and 20.10.8.2 [allocator.uses.construction] of the standard.

7.4 Remove all mention of *type-erased allocator* from the TS

Remove section section 3.1 [utility], which introduces header `<experimental/utility>`, defining `struct erased_type`, from the TS draft.

Remove section 5.3 [memory.type.erased.allocator], which defines *type-erased allocator* and describes its properties, from the TS draft.

Remove all of section 7 [futures], which attempted to apply the ill-conceived principles of section 5.3 to `promise` and `packaged_task`, from the TS draft.

7.5 Changes to `std::experimental::function`

In section 4.1 [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_typestd::pmr::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 (5.3). If the constructor moves or makes a copy of a function object (C++20 §20.14), 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`, `experimental::pmr::get_default_resource()` at the time of the construction of `f` if no allocator was specified.~~

A `function` object stores an allocator object of type of `std::pmr::polymorphic_allocator<>`, which it uses to allocate memory for its internal data structures. In the `function` constructors, the allocator is initialized (before the target object, if any) as follows:

— For the move constructor, the allocator is initialized from `f.get_allocator()`, where `f` is the parameter of the constructor.

— For constructors having a first parameter of type `allocator_arg_t`, the allocator is initialized from the second parameter.

— For all other constructors, the allocator is value initialized.

In all cases, the allocator of a parameter having type `function&&` is unchanged. If the constructor creates a target object, that target object is initialized by uses-allocator construction with the allocator and other target-object constructor arguments. [*Note*: If a constructor parameter of type `experimental::function&&` has an allocator equal to that of the object being constructed, the implementation can often transfer ownership of the target rather than constructing a new one. – *end note*]

`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)`. [*Note:* the stored allocator is unchanged – *end note*] ~~The memory resource returned by `get_memory_resource()` after the assignment is equivalent to the memory resource before the assignment. [*Note*: the address returned by `get_memory_resource()` might change – *end note*]~~

Returns: `*this`.

`template<class F> function& operator=(F&& f);`

Constraints: `declval<decay_t<F>&>()` is *Lvalue-Callable* (C++20 §20.14.16.2) for argument types `ArgTypes...` and return type `R`.

Effects: `function(allocator_arg, ~~ALLOCATOR_OF(*this)~~get_allocator(), std::forward<F>(f)).swap(*this);`

Returns: `*this`.

`template<class F> function& operator=(reference_wrapper<F> f) noexcept;`

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);
```

Preconditions: ~~`*this->get_memory_resource() == *other.get_memory_resource()`~~ `this->get_allocator() == other.get_allocator()`.

Effects: Interchanges the targets of `*this` and `other`.

Throws: nothing.

Remarks: The allocators of `*this` and `other` are not interchanged.

NOTE: The omission of `noexcept` for `swap` is deliberate. When `noexcept` was added to `swap` in C++20, `swap` had a wide interface (no preconditions). The addition of allocators gives `swap` a narrow interface, so `noexcept` would violate the Lakos rule. Nevertheless, if LWG wants to add it back, I would not have a serious objection.

Add a new section describing the `get_allocator()` function:

```
allocator_type get_allocator() const noexcept;
```

Returns: A copy of the allocator initialized during construction (4.2.1) of this object.

8 References

P0039r6 `polymorphic_allocator<>` *as a vocabulary type*, Pablo Halpern & Dietmar Kühl, 2019-02-22.

N4920 *Working Draft, C++ Extensions for Library Fundamentals, Version 3*, Thomas Köppe, editor, 2022-08-15.

N3916 *Polymorphic Memory Resources - r2*, Pablo Halpern, 2014-02-14.