**Doc No:** P0987r0

**Date:** 2018-04-02

**Audience:** LWG

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

polymorphic\_allocator<byte> instead of type-erasure

**Contents**

1 Abstract 1

2 History 1

3 Motivation 2

4 Proposal Overview 2

5 Future directions 3

6 Formal Wording 3

6.1 Document Conventions 3

6.2 Feature test macros 3

6.3 Undo changes to uses-allocator construction 3

6.4 Remove erased\_type from the TS 4

6.5 Changes to std::experimental::function 4

6.6 Changes to type-erased allocator 6

6.7 Changes to class template promise 7

6.8 Changes to class template packaged\_task 7

7 References 7

# 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-dimentional 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](https://wg21.link/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.

# 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:

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<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:

polymorphic\_allocator<byte> get\_allocator() const noexcept;

# 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.

# Future directions

We should consider using polymorphic\_allocator<byte> in the interface to std::experimental::any.

# Formal Wording

## Document Conventions

All section names and numbers are relative to the November 2016 draft of the Library Fundamentals TS, [N4617](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/n4617.pdf). 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.

## Feature test macros

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

Table 2 — Significant features in this technical specification

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Doc. No.** | **Title** | **Primary Section** | **Macro Name Suffix** | **Value** | **Header** |
| N3916 P0987R0 | Type-erased Polymorphic allocator for function | 4.2 | function\_erasedpolymorphic\_allocator | 201406 201804 | <experimental/functional> |
| N3916 P0987R0 | Type-erased Polymorphic allocator for promise | 11.2 | promise\_erasedpolymorphic\_allocator | 201406 201804 | <experimental/future> |
| N3916 P0987R0 | Type-erased Polymorphic allocator for packaged\_task | 11.3 | packaged\_task\_erasedpolymorphic\_allocator | 201406 201804 | <experimental/future> |

## 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.

## 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.

## 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\_typepmr::polymorphic\_allocator<byte>;

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 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()get\_allocator().

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.

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). The memory resourceallocator returned by get\_memory\_resource()get\_allocator() after the assignment is equivalent to the memory resourceallocator before the assignment. [ *Note*: the address returned by get\_memory\_get\_allocator().resource() might change — *end note* ]

*Returns*: \*this.

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... and return type R.

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()  
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.

## 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 rptrpmr\_alloc is computedinitialized from alloc depends on the type of alloc as described in Table 13:

Table 13 — Initialization of type-erased allocator

|  |  |
| --- | --- |
| **If the type of alloc is** | **then the value of rptr is** |
| 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 |
| pmr::polymorphic\_allocator<U> | pmr\_alloc is initialized with alloc.resource() |
| any other type meeting the Allocator requirements (C++14 §17.6.3.5) 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. rptrpmr\_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\_typepmr::polymorphic\_allocator.
* X.get\_memory\_resource()X.get\_allocator() returns rptrpmr\_alloc.

## 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:

template <class R>

class promise {

public:

using allocator\_type = erased\_typepolymorphic\_allocator<byte>;

...

pmr::memory\_resource\* get\_memory\_resource();

allocator\_type get\_allocator() const noexecpt;

};

## 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:

template <class R, class... ArgTypes>

class packaged\_task<R(ArgTypes...)> {

public:

using allocator\_type = erased\_typepolymorphic\_allocator<byte>;

...

pmr::memory\_resource\* get\_memory\_resource();

allocator\_type get\_allocator() const noexecpt;

};

# References

[P0039r4](http://wg21.link/P0339) *polymorphic\_allocator<> as a vocabulary type*, Pablo Halpern & Dietmar Kühl, 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.