Undeprecate `polymorphic_allocator::destroy` for C++26

Document #:	P2875R3
Date:	2024-01-15
Project:	Programming Language C++
Audience:	Library Evolution

Reply-to:	Alisdair Meredith <ameredith1@bloomberg.net>

Contents

1 Abstract

The member function `polymorphic_allocator::destroy` was deprecated by C++23 as it defines the same semantics that would be synthesized automatically by `std::allocator_traits`. However, some common use cases for `std::pmr::polymorphic_allocator` do not involve generic code and thus do not necessarily use `std::allocator_traits` to call on the services of such allocators. This paper recommends undeprecating that function and restoring its wording to the main Standard clause.

2 Revision History

R3: Febuary 2024 (pre-Tokyo mailing)

— Applied an editorial review, fixing grammar and typos
— Then totally rewrote the analysis, reflecting a subtle change of semantics
— Confirmed wording against latest working draft, [N4971]
— Record results of LEWG telecon, January 23, 2024
— Add requested code examples of what migration would look like

R2: September 2023 (midterm mailing)

— Removed revision history's redundant subsection numbering
— Added comparison with effects of removing a typedef member instead
— Wording updates
  — Confirm wording against latest working draft, N4958
  — Updated stable label cross-reference to C++23
— Applied numerous editorial corrections

R1: August 2023 (midterm mailing)

— Confirmed wording for latest working draft, N4950
— Removed syntax highlighting from standardese to avoid markup conflicts
— Removed use of `allocator_traits` in `delete_object`
— Improved rationale following initial reflector review — thanks, Pablo!

R0: May 2023 (pre-Varna mailing)

— Initial draft of this paper.

3 Introduction

At the start of the C++23 cycle, [P2139R2] tried to review each deprecated feature of C++ to see which we would benefit from actively removing and which might now be better undeprecated. Consolidating all this analysis into one place was intended to ease the (L)EWG review process but in return gave the author so much feedback that the next revision of that paper was not completed.

For the C++26 cycle, a concise paper will track the overall review process, [P2863], but all changes to the Standard will be pursued through specific papers, decoupling progress from the larger paper so that delays on a single feature do not hold up progress on all.

This paper takes up the deprecated member function `std::polymorphic_allocator::destroy`, D.15 [depr.mem.poly.allocator.mem].

4 Issue History

This feature was deprecated by [LWG3036].

4.1 LWG Poll, 2019 Kona meeting

Q: Are we in favor of deprecation, pending on paper [P0339R6]?

```
|  F  |  N  |  A  |
|  5  |  3  |  2  |
```

4.2 2020-10-11 Reflector poll

Moved to Tentatively Ready after seven votes in favour.

4.3 November 2020 Virtual Plenary

Adopted for C++23 by omnibus issues paper [P2236R0].

5 Analysis

When the original LWG issue was opened on November 15, 2017, the issue claimed that the code for `destroy` was identical to that which would be synthesized automatically by `allocator_traits` if the member function were missing. That part was accurate. It further claimed that this member was therefore redundant and should be removed. We claim that part was misunderstood and taken at face value, as there is no record of controversy in the issue. However, just because two functions have an identical implementation at one point in time does not mean they have an identical contract, or rather, that as designs and contracts evolve they would remain synchronized unless that synchronization itself were an explicit part of the contract. In this case, the `destroy` member is specified to be the appropriate way to destroy an object created by the corresponding `construct` call, where the `allocator_traits` member is a default pattern to use in the absence of a specific `destroy` for a given allocator type. There is no guarantee that those two functions with an identical implementation in 2017 would retain an identical definition into the future. Indeed, the adoption of [P0784R7] at the July 2019 meeting in Cologne changed the specification of the `allocator_traits::destroy` formula to call the `std::destroy_at` free function, rather than call the destructor directly. That function has a different behavior for array types than non-array types. There is no acknowledgement of this functionality divergence in the LWG issue when it is voted into Ready status. `std::destroy_at` is seen as the natural undo for `std::construct_at`, which is also incorporated as part of the default formula for `allocator_traits::construct`, and that is *not* the formula used by `std::pmr::polymorphic_allocator<T>::construct`. All of this extra complexity does not provide any benefit to `polymorphic_allocator` as it is largely to support `constexpr` allocation, that is not supportable by `polymorphic_allocator` without further language extensions in the domain of compile-time dynamic objects that have not yet been proposed; if `polymorphic_allocator` were to be extended in that way in the future, it is likely that the `destoy` member function would be specified very differently to calling `std::destroy_at`. I say this as feedback from experiments supporting compile-time allocation through `polymorphic_allocator` in the past, which showed a need for `if consteval` logic in the implementation.

Meanwhile, the motivation of removing a redundancy that is not a redundancy is flawed from the perspective of why the function exists in the first place. The natural undo for an `a.construct` call is an `a.destroy` call. When using a polymorphic allocator as a means of wrapping a `std::pmr::memory_resource`, it is intended to use the allocator directly, rather than channel all allocator functionality through `allocator_traits` when there is no genericity swapping out other allocator types to support; the whole point of memory resources is to move the selection of allocator from compile-time to runtime, and `polymorphic_allocator` is the vocabulary for runtime allocator customization in the same way that `allocator_traits` is the vocabulary for compile-time allocator customization. The intent of the issue (assuming the functionality of `allocator_traits::destroy` had kept in sync) is that there is no loss of functionality, but that users should now write

```
std::allocator_traits<std::pmr::polymorphic_allocator<MyType>>::destroy(a, p);
```

rather than

```
a.destroy(p);
```

for exactly the same functionality. It is purely a loss of expressiveness, for no clear purpose.

Meanwhile, resolving the issue was deferred for C++20 until paper [P0339R6] landed, further promoting use of `polymorphic_allocator<>` as the vocabulary type for runtime customization of allocators. It is not clear how this furthered the case for deprecation. On its surface it appears equivalent to saying "we should deprecate comparison operators on containers, as users can call the standard algorithms directly"

5.1 C++17: Issue filed

The implementation of the destroy functions can be seen to have equivalent code when called through `allocator_traits`.

polymorphic_allocator

```
template <class E>
template <class T>
void polymorphic_allocator<E>::destroy(T* p) {




   p->~T();

}
```

allocator_traits

```
template <class A>
template <class T>
void allocator_traits<A>::destroy(A& a, T* p) {
   if constexpr( __has_delete_member<A>() ) {
      a.destroy(p);
   }
   else {
      p->~T();
   }
}
```

Observe that calling through `allocator_traits` puts more work on the compiler, but both branches of the `if constexpr` produce exactly the same destructor call at the end of the call chain — that is the whole rationale for deprecating, and ultimately removing, the `polymorphic_allocator::destroy` function.

However, the case for the `destroy` member function is different to the case for removing a typedef member, such as in [depr.default.allocator] and approved for removal in [P2868R2]. The formula produced by `allocator_traits` for a missing typedef member is to compute a type based upon other typedef names in `allocator_traits`. When a typedef member from a base class provides the exact same result as the formula would produce for the base class, that typedef member will inhibit `allocator_traits` from computing the correct typedef name for the derived class, forcing the user to explicitly provide that member themselves; this situation is often a bug by omission. In the case of a `destroy` function matching the functionality that would be provided by `allocator_traits`, nothing in that functionality actually depends upon the class itself, so calling that function instead for a derived class would still have identical behavior; there is no risk of introducing a bug by error of omission.

5.2 C++20: Support transient `constexpr` (de)allocation

The contract of `allocator_traits` continued to evolve in C++20 in order to support transient allocation and deallocation during constant evaluation. In particular, the path taken by the `destroy` function is no longer the same as that specified for `polymorphic_allocator::destroy`. Rather than invoking the destructor directly, it is deferred through another level of indirection to `std::destroy_at`, which typically invokes the destructor for the pointed-to object, unless it is an array. From the perspective of the optimizer, it can see that the destructor for `T` (typically) does not throw from its exception specification, allowing elimination of any conservative exception unwinding code; as the `destroy_at` function is *not* a `noexcept` function, the optimizer must do more work through inlining the call to make that same optimization.

polymorphic_allocator

```
template <class E>
template <class T>

void polymorphic_allocator<E>::destroy(T* p) {




   p->~T();    // `noexcept` function call
               // can eliminate overhead
}
```

allocator_traits

```
template <class A>
template <class T>
constexpr
void allocator_traits<A>::destroy(A& a, T* p) {
   if constexpr( requires(A& a, T* p){a.destroy(p);} ) {
      a.destroy(p);
   }
   else {
      std::destroy_at(p); // potentially throwing
   }
}
```

5.3 C++23: Deprecate `polymorphic_allocator::destroy`

For C++23, we finally deprecate the `polymorphic_allocator::destroy` member function, despite the functionality no longer being a precise match for the default formula supplied by `allocator_traits`; however, there is no change of functionality yet, as the `allocator_traits` function must still dispatch to the deprecated `polymorphic_allocator::destroy` function.

5.4 Directing towards a bad user experience

Note, however, that the direct call through a `polymorphic_allocator` object does not need any type names as the interface is deliberately designed to be type agnostic through type deduction on the pointer, whereas the allocator type — including the object type that it allocates for — must be known to invoke the `allocator_traits` functionality.

`polymorphic_allocator`	`allocator_traits`
`a.destroy(p);`	`std::allocator_traits<decltype(a)>::destroy(a, p);`

As the `allocator_traits::destroy` function takes its allocator argument by reference to non-`const` allocator, it is ill-formed to pass a polymorphic allocator bound to a different type — while a temporary of the right type of allocator could be produced by a single conversion sequence, the temporary will not bind to the by-reference parameter, requiring the exact type of allocator to be supplied.

6 Review: C++26

6.1 LEWG telecon: 2024/01/23

Presented R1 of this paper, as R2 in the September mailing was missed — possibly as it does not get shown that https://wg21.link/p2875 unless you explicitly request https://wg21.link/p2875r2. It is thought this is due to R1 being a html document, but R2 was published as a prd, changing the file extension.

Oral argument of much of the analysis above without presenting this paper

Suggestions that users updating their code is not onerous, based on experience with removing functions from `std::allocator`.

Concerns that `std::allocator` is not a primary interface for non-generic code, where `std::pmr::polymorphic_allocator` is designed as vocabulary for custom allocation in non-generic code.

Suggestion that best practice is that even non-generic code should always use `std::allocator_traits` to request allocator services. Disagreement from paper authors, and this specific best-practice is never polled for the room.

Consensus of the meeting was to add an example (5.4) of how code would migrate if the deprecated API were removed, and then forward for electronic polling with the paper's preferred recommendation to undeprecate, per the supplied wording.

7 Proposal

`std::pmr::polymorphic_allocator` is an allocator that will often be used in nongeneric circumstances unlike, for example, `std::allocator`. This member function that could otherwise be synthesized by `std::allocator_traits` should still be part of its pubic interface for direct use.

Hence, this paper recommends undeprecating the `destroy` member function as the natural and expected analog paired with `construct`.

8 Proposed Wording

Make the following changes to the C++ Working Draft. All wording is relative to [N4971], the latest draft at the time of writing.

8.1 Update the library specification

20.4.3.1 [mem.poly.allocator.class.general] General

[2] A specialization of class template `pmr::polymorphic_allocator` meets the allocator completeness requirements (16.4.4.6.2 [allocator.requirements.completeness]) if its template argument is a *cv*-unqualified object type.

```
namespace std::pmr {
  template<class Tp = byte> class polymorphic_allocator {
    memory_resource* memory_rsrc;       // exposition only

  public:
    using value_type = Tp;

    // 20.4.3.2[mem.poly.allocator.ctor], 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&) = delete;

    // 20.4.3.3[mem.poly.allocator.mem], member functions
    [[nodiscard]] Tp* allocate(size_t n);
    void deallocate(Tp* p, size_t n);

    [[nodiscard]] 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> [[nodiscard]] T* allocate_object(size_t n = 1);
    template<class T> void deallocate_object(T* p, size_t n = 1);
    template<class T, class... CtorArgs> [[nodiscard]] 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);

    template< class~T>
      void~destroy(T*~p);

    polymorphic_allocator select_on_container_copy_construction() const;

    memory_resource* resource() const;

    // friends
    friend bool operator==(const polymorphic_allocator& a,
                           const polymorphic_allocator& b) noexcept {
      return *a.resource() == *b.resource();
```

```
    }
  };
}
```

20.4.3.3 [mem.poly.allocator.mem] Member functions

```
template<class T>
  void delete_object(T* p);
```

13 *Effects:* Equivalent to:

```
~~allocator_traits< polymorphic_allocator>::~~destroy(~~*this,~~p);
deallocate_object(p);
```

```
template<class T, class... Args>
  void construct(T* p, Args&&... args);
```

14 *Mandates:* Uses-allocator construction of `T` with allocator `*this` (see 20.2.8.2 [allocator.uses.construction]) and constructor arguments `std::forward<Args>(args)...` is well-formed.

15 *Effects:* Construct a `T` object in the storage whose address is represented by `p` by uses-allocator construction with allocator `*this` and constructor arguments `std::forward<Args>(args)...`.

16 *Throws:* Nothing unless the constructor for `T` throws.

```
template<class T>
  void destroy(T* p);
```

X *Effects:* As if by `p->~T()`.

```
polymorphic_allocator select_on_container_copy_construction() const;
```

17 *Returns:* `polymorphic_allocator()`.

18 [*Note 4:* The memory resource is not propagated. —*end note*]

8.2 Strike Annex D wording

D.15 [depr.mem.poly.allocator.mem] Deprecated `polymorphic_allocator` member function

1 The following member is declared in addition to those members specified in 20.4.3.3 [mem.poly.allocator.mem]:

```
namespace std::pmr {
  template<class Tp = byte>
  class polymorphic_allocator {
  public:
    template <class T>
      void destroy(T* p);
  };
}
```

```
template<class T>
  void destroy(T* p);
```

2 *Effects:* As if by `p->~T()`.

8.3 Update cross-reference for stable labels for C++23

Cross-references from ISO C++ 2023

All clause and subclause labels from ISO C++ 2023 (ISO/IEC 14882:2023, *Programming Languages — C++*) are present in this document, with the exceptions described below.

container.gen.reqmts *see*
container.requirements.general

depr.arith.conv.enum *removed*
depr.codecvt.syn *removed*
depr.default.allocator *removed*
depr.locale.stdcvt *removed*
depr.locale.stdcvt.general *removed*
depr.locale.stdcvt.req *removed*
depr.mem.poly.allocator.mem *removed*
depr.res.on.required *removed*
depr.string.capacity *removed*

mismatch *see* alg.mismatch

9 Acknowledgements

Thanks to Michael Park for the pandoc-based framework used to transform this document's source from Markdown.

Thanks to Pablo Halpern for good reviews and helping to organize the rationale.

Thanks to Lori Hughes for reviewing this paper and providing editorial feedback.

10 References

[LWG3036] Casey Carter. polymorphic_allocator::destroy is extraneous.
https://wg21.link/lwg3036

[N4971] Thomas Köppe. 2023-12-18. Working Draft, Programming Languages — C++.
https://wg21.link/n4971

[P0339R6] Pablo Halpern, Dietmar Kühl. 2019-02-22. polymorphic_allocator<> as a vocabulary type.
https://wg21.link/p0339r6

[P0784R7] Daveed Vandevoorde, Peter Dimov,Louis Dionne, Nina Ranns, Richard Smith, Daveed Vandevoorde. 2019-07-22. More constexpr containers.
https://wg21.link/p0784r7

[P2139R2] Alisdair Meredith. 2020-07-15. Reviewing Deprecated Facilities of C++20 for C++23.
https://wg21.link/p2139r2

[P2236R0] Jonathan Wakely. 2020-10-15. C++ Standard Library Issues to be moved in Virtual Plenary, Nov. 2020.
https://wg21.link/p2236r0

[P2863] Alisdair Meredith. Review Annex D for C++26.
https://wg21.link/p2863

[P2868R2] Alisdair Meredith. 2023-09-14. Remove Deprecated ‘std::allocator‘ Typedef From C++26.
https://wg21.link/p2868r2