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

Document #:	P2875R2
Date:	2023-09-15
Project:	Programming Language C++
Audience:	Library Evolution Incubator

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

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)

— 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, [P2863R1], 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.17 [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 2023 Virtual Plenary

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

5 Analysis

`std::pmr::polymorphic_allocator` is an allocator that will often be used in non-generic 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`.

5.1 Symmetry

`polymorphic_allocator` has `construct`, so logically it should also have `destroy`. If I see a class that overrides `new` but does not override `delete`, I get suspicious at best and disgusted at worst. If I write code that uses `construct`, I will probably also want to call `destroy`, even if I know that the call is a no-op or can be expressed another way.

5.2 Nongeneric use

Code that does not use an allocator template (e.g., `experimental::function` from the LFTS), can use `polymorphic_allocator` to avoid type erasure. Such code would not need to use the `allocator_traits` indirection and would call `allocate`, `construct`, `destroy`, and `deallocate` directly. Yes, it could use `destroy_at` directly, but that breaks abstraction and symmetry (see the "Symmetry" section above). Any such existing code would need to change if `destroy` is removed.

5.3 Inability to use `[[deprecated]]`

If one of the goals is to avoid writing something that is equivalent to the code that `allocator_traits` would already provide if you had not provided it, then `allocator_traits` needs to detect the presence or absence of member-function `destroy`. That detection will invariably cause a deprecation warning if `destroy` is annotated as `[[deprecated]]`. Therefore, when the `destroy` method is eventually removed, unsuspecting code breakage will occur.

Note: Reports have since been made that the deprecation warning can be turned off in a platform-specific way using pragmas within `allocator_traits`. Alternatively, `allocator_traits` can be specialized for `polymorphic_allocator` to avoid calling the deprecated member function.

It is worth noting that the case for the `destroy` member function is different to the case for removing a typedef member, such as in D.16 [depr.default.allocator] and heading towards 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` computing the correct tydedef name for the derived class forcing the user to explicitly provide that member themselves, 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.

5.4 Removal does not serve the Standard

The deprecation and removal of `destroy` has very little benefit to the Standard — certainly not enough to justify breaking code (see the "Nongeneric Use" section above).

6 Proposed Wording

Make the following changes to the C++ Working Draft. All wording is relative to [N4958], the latest draft at the time of writing.

6.1 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*]

6.2 Strike Annex D wording

D.17 [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()`.

6.3 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.mem.poly.allocator.mem *removed*
depr.res.on.required *removed*

7 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.

8 References

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

[N4958] Thomas Köppe. 2023-08-14. Working Draft, Programming Languages -- C++.
https://wg21.link/n4958

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

[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

[P2863R1] Alisdair Meredith. 2023-08-16. Review Annex D for C++26.
https://wg21.link/p2863r1

[P2868R2] Alisdair Meredith. 2023-09-15. Remove Deprecated `std::allocator` Typedef From C++26.
https://wg21.link/p2868r2