Remove Deprecated `std::allocator` Typedef From C++26

Document #:	P2868R1
Date:	2023-08-15
Project:	Programming Language C++
Audience:	Library Evolution

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

Contents

1 Abstract

The Standard Library `allocator` class contains a deprecated `typedef` member than can cause problems for derived classes. This paper proposes removing that member from the C++ Standard Library.

2 Revision History

2.1 R1: 2023 August (midterm mailing)

Updates following LEWG online review, before submitting to electronic poll:

- Fix broken markup, correcting some links
- Moved from LEWGI to LEWG
- Added a new section to address concerns raised during LEWG review
- Confirmed that no changes are expected for 16.4.5.3.2 [zombie.names]
- Provided Annex C wording

2.2 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 the paper was not completed.

For the C++26 cycle, a much shorter paper, [P2863R1], will track the overall analysis, but for features that the author wants to actively progress, a distinct paper will decouple progress from the larger paper so that the delays on a single feature do not hold up progress on all.

This paper takes up the deprecated `is_always_equal` member typedef in `std::allocator`, D.17 [depr.default.allocator].

4 Proposal

Remove the deprecated typedef `std::allocator<T>::is_always_equal` from C++26.

5 Analysis

The Standard Library `allocator` class has a typedef member that could be synthesized from the primary `allocator_traits` template and was deprecated in C++20 by [LWG3170]. When `std::allocator` provides this member directly, any classes that derive from it will not synthesize this type name correctly, but use the `true_type` value provided directly by `std::allocator`. If the derived allocator type is not empty, its value for this trait will not match the expected default behavior, forcing the allocator author (if they are aware) to add their own typedef that should not be needed to restore the default behavior; typically, this process leads to subtle bugs.

While this case is a small corner for misuse, the concern is embarrassing for the Committee to explain, and the Standard Library allocator is a common example folks will follow when trying to write their first allocators. Hence, this paper recommends the removal of this deprecated typedef for C++26.

6 Concerns

During LEWG reflector review before electronic polling, Pablo Halpern observed that our assumption for backward compatibility seems to implicitly reply on `std::allocator` always being an empty type, which is not guaranteed by the Standard.

However, Corentin Jabot highlighted that 20.2.10.1 [default.allocator.general]p2 places a guarantee on the trait that it yields `true_type` for all instantiations of `std::allocator`, for both Standard Library vendors and users specializing the `std::allocator` template for their own types.

If a Standard Library vendor maintains a `std::allocator` implementation that is not empty such that the default algorithm in `allocator_traits` would produce the wrong result, that vendor has the freedom to specialize that trait as an implementation detail of the Standard Library itself. This freedom is unavailable to end users since C++23; see [P2652R2] for more details.

An end user can achieve this guarantee for their own specializations of `std::allocator` only by implementing as an empty type, unless the Standard Library vendor *has* provided the partial specialization for `std::allocator_traits<allocator<T>>` to make this happen.

While the current wording seems sufficient, we might benefit from exploring a mandate that `std::allocator` is always an empty type or that the Standard Library provides a partial specialization for `allocator_traits` so that `std::allocator_traits<allocator<T>>::is_always_equal::value` is always `true`.

We might also reasonably argue that 20.2.10.1 [default.allocator.general]p2 already places sufficient constraints on a Standard Library vendor that it must satisfy this requirement, by either making `std::allocator` empty or providing the partial specialization described above, and which technique is used is unspecified. Following this interpretation, users' specializations of `std::allocator` must always be empty types in portable code.

7 Wording

All wording is relative to [N4950], the latest working draft at the time of writing.

7.1 Zombie names

The only name being removing is a member type alias that is part of a protocol that relies on that name being present or not; the Standard Library's requirement for this name does not change simply by removing one use of it. Hence, nothing need be added to 16.4.5.3.2 [zombie.names].

7.2 Proposed changes

C.1.X Annex D: compatibility features [diff.cpp23.depr]

1 **Change:** Remove the type alias `std::allocator<T>::is_always_equal`.

Rationale: The behavior that follows from defining this type is the default behavior supplied by the `allocator_traits` template, so the type is not needed. However, the redundant declaration is then present in derived classes where it will replace the deduced behavior of the `allocator_traits`, potentially causing wrong behavior.

Effect on original feature: A valid C++ 2023 program that derives an allocator class from `std::allocator` and does not provide its own alias named `is_always_equal` to override its own behavior will now get the deduced behavior from `allocator_traits`, potentially aliasing `false_type` instead of `true_type` for correct behavior.

D.17 [depr.default.allocator] The default allocator

1 The following member is defined in addition to those specified in 20.2.10 [default.allocator]:

```
namespace std {
  template<class T> class allocator {
  public:
    using is_always_equal = true_type;
  };
}
```

8 Acknowledgements

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

Thanks to Pablo Halpern and Corentin Jabot for pointing out the importance of 20.2.10.1 [default.allocator.general]p2 to confirm that the behavior after removal is always guaranteed to be the same for `std::allocator`.

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

9 References

[LWG3170] Billy O'Neal III. is_always_equal added to std::allocator makes the standard library treat derived types as always equal.
https://wg21.link/lwg3170

[N4950] Thomas Köppe. 2023-05-10. Working Draft, Standard for Programming Language C++.
https://wg21.link/n4950

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

[P2652R2] Pablo Halpern. 2023-02-09. Disallow user specialization of allocator_traits.
https://wg21.link/p2652r2

[P2863R1] Alisdair Meredith. 2023-08-15. Review Annex D for C++26.
https://wg21.link/d2863r1