Remove the Deprecated `iterator` Class Template from C++26

Document #:	P3365R1
Date:	2025-03-11
Project:	Programming Language C++
Audience:	LWG

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

Contents

1 Abstract

This paper proposes removing the deprecated `iterator` class template from the next C++ Standard.

2 Revision History

R1 March 2025 (post-Hagenberg mailing)

- Recorded LEWG telecon review following Hagenberg
- Forwarded to LWG for C++29, or time and resource permitting, for C++26
- Rebased wording onto N5001
  - cleaned up presentation of adding entries to tables and lists
  - added Annex C wording

R0 August 2024 (midterm mailing)

Initial draft of this paper, based on content originally in [P2863]

3 Introduction

The topic of this paper has been extracted from the general deprecation review paper, [P2863], into its own paper so as to better track its progress, since this topic has had a couple of reviews but is not reaching a real conclusion while embedded in the broader paper.

The class template `iterator` was part of the original C++ Standard and deprecated in C++17 by [P0174R2].

4 Analysis

Providing the needed support for iterator typenames through a templated dependent base class, which determines which name maps to which typedef name purely by parameter order, is less clear than simply providing the needed names, and this was the concern that led to deprecation. Furthermore, corner cases in usage where name lookup does not look into a dependent base class make this tool hard to recommend as a simpler way of providing the type names, yet that purpose is the whole reason for this class template to exist.

The remaining use case is to ensure that all needed typedef names were supplied with a default, but subsequent work on iterators and ranges ([P0896R4]) that landed in C++20 means that the primary `iterator_traits` template can provide those defaults, using a better set of deduction rules.

With the upgrade of `iterator_traits` in C++20, this class template is not only strictly redundant, but can be actively harmful by substituting the wrong defaults.

5 Design Principles

Remove deprecated features from the Standard specification at the earliest *practical* opportunity to minimize the accumulation of technical debt.

6 Proposed Solution

Remove the deprecated Standard Library API from C++26 while granting vendors permission to continue supplying it as a conforming extension, for as long as they desire, through the use of zombie names.

7 Review History

7.1 C++20 review

When this facility was reviewed for removal in C++20 there were valid use cases that relied on the default template arguments to deduce at least a few of the needed type names.

The main concern that remained was breaking old code by removing this code from the Standard Libraries. That risk is ameliorated by the zombie names clause in the Standard, allowing vendors to maintain their own support for as long as their customers demand. By the time the next Standard would ship, those customers would already be on six years notice that their code might not be supported in future Standards. However, LEWG noted the repeated use of the name `iterator` as a type within many containers means we might choose to leave this name off the zombie list. We conservatively place it there anyway to ensure that we are covered by the previous standardization terminology to encompass uses other than as a container iterator `typedef` and to preserve its use at namespace and/or global scope.

The recommendation at this time was to take no action until a stronger consensus for removal is achieved.

7.2 C++23 review

The initial (and only) LEWG review is minuted for the telecon on 2020/07/13.

Concerns were raised about the lack of research into how much code is likely to break with the removal of this API. We would like to see more analysis of how frequently this class is used, notably in publicly available code such as across all of GitHub. The better treatment of implicit generation of `iterator_traits` in C++23 and more familiarity with a limited number of code bases that still rely on this facility gave more confidence in moving forward with removal than we had for C++20. LEWG noted that the name may be unfortunate with the chosen form of concept naming adopted for C++20, so its removal might lead to one fewer source of future confusion. Given that implementers are likely to provide an implementation (through zombie names freedom) for some time after removal, LEWG reached consensus to proceed with removal, assuming the requested research does not reveal major concerns before the main LEWG review to follow.

7.3 C++26 reviews

7.3.1 LEWG review: Kona, 2023/11/07

Due to an oversight by the author when presenting the larger paper, [P2863], the review to affirm (no) progress on removing the deprecated `iterator` class template was skipped.

7.3.2 LEWG review: Telecon, 2025/03/04

Review for this paper was deferred from review in Hagenberg to the first following telecon, along with other deprecation-removal papers. The review intent is to poll forwarding these papers to LWG for C++29, and if that poll succeeds by a follow-up poll, if time and LWG resources permit, for C++26.

Concerns were raised about whether sufficient research has been conducted on how much code would be broken by this change. Concerns were assuaged by the demonstrated history of library implementers relying on the Zombie Names clause to maintain support for an extended period with a plan to phase in a reduced support through configuration macros to opt into removal, before requiring a macro to opt out of removal, before finally removing support — if ever!

It was noted that deprecated library features are not intended to sit in limbo in Annex D forever. A question was raised about whether we might ever want to undeprecate this feature instead. If not, it has been deprecated long enough that we should proceed with removal. No-one was aware of current attempts to undeprecate.

This review noted a number of formatting issues and the lack of Annex C wording. The author noted that he usually defers the effort of writing Annex C wording until the design is approved. However, Annex C wording is produced ahead of time if the examples are thought to be helpful in evaluating a given removal.

Two polls were taken.

POLL: Fix P3365R0 formatting as needed (and other minor fixes needed) and forward to LWG for C++29.

```
SF  F  N  A SA
 8  9  1  0  0
```

Strong consensus.

POLL: Fix P3365R0 formatting as needed (and other minor fixes needed) and forward to LWG with recommendation to apply for C++26 (if possible).

```
SF F  N  A SA
5  8  1  1  0
```

Consensus.

8 Wording

Make the following changes to the C++ Working Draft. All wording is relative to [N5001], the latest draft at the time of writing.

16.4.5.3.2 [zombie.names] Zombie names

Add new identifiers to table 38 [tab:zombie.names.std]

— `iterator`

C.1.8 [diff.cpp23.depr] Annex D: compatibility features

x **Change:** Remove the class template `iterator`.

Rationale: The class template was originally designed to support deducing the various typedef names that enable `iterator_traits` to deduce its whole interface. That same deduction is now provided more clearly by providing the two needed type aliases in the user's iterator class. Further, as the model for using the class template is to be used as a dependent base class, there were rare cases where the typedef names in that dependent base would not be found by name lookup.

Effect on original feature: A valid C++ 2023 program using this class template may fail to compile.

[*Example 1:*

```
template <class T>
struct DeprecatedIterator<std::iterator<std::input_iterator_tag, T>> {
   // deprecated in C++23, may become ill-formed in C++26

   // implementation details
}

template <class T>
struct ModernIterator {
   using iterator_category = std::input_iterator_tag;
   using value_type        = T;

   // valid in both C++23 and C++26

   // implementation details
}
```

—*end example*]

D.17 [depr.iterator] Deprecated `iterator` class template

1 The header `<iterator>` (24.2 [iterator.synopsis]) has the following addition:

```
namespace std {
  template<class Category, class T, class Distance = ptrdiff_t,
           class Pointer = T*, class Reference = T&>
    struct iterator {
      using iterator_category = Category;

      using value_type        = T;
      using difference_type   = Distance;
      using pointer           = Pointer;
      using reference         = Reference;
```

```
  };
}
```

2 The `iterator` template may be used as a base class to ease the definition of required types for new iterators.

3 [*Note 1:* If the new iterator type is a class template, then these aliases will not be visible from within the iterator class's template definition, but only to callers of that class. —*end note*]

4 [*Example 1:* If a C++ program wants to define a bidirectional iterator for some data structure containing `double` and such that it works on a large memory model of the implementation, it can do so with:

```
class MyIterator :
    public iterator<bidirectional_iterator_tag, double, long, T*, T&> {
    // code implementing ++, __etc._
};
```

—*end example*]

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

Add the following entries to the list of cross-references for stable labels in previous standards.

depr.iterator *removed*

9 Acknowledgements

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

Thanks to Lori Hughes for reviewing this paper.

10 References

[N5001] Thomas Köppe. 2024-12-17. Working Draft, Programming Languages — C++.
https://wg21.link/n5001

[P0174R2] Alisdair Meredith. 2016-06-23. Deprecating Vestigial Library Parts in C++17.
https://wg21.link/p0174r2

[P0896R4] Eric Niebler, Casey Carter, Christopher Di Bella. 2018-11-09. The One Ranges Proposal.
https://wg21.link/p0896r4

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