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

Document #:	P3365R0
Date:	2024-07-23
Project:	Programming Language C++
Audience:	Library Evolution
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

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 Historical Reviews

5.1 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.

5.2 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.

6 Design Principles

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

7 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.

8 C++26 Review History

8.1 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.

9 Wording

Make the following changes to the C++ Working Draft. All wording is relative to [N4986], the latest draft at the time of writing.

9.1 Add new identifiers to 16.4.5.3.2 [zombie.names]

16.4.5.3.2 [zombie.names] Zombie names

1 In namespace `std`, the names shown in Table 38 are reserved for previous standardization:

Table 1: Table 38 — Zombie names in namespace std [tab:zombie.names.std]

`auto_ptr`		`pointer_to_binary_function`
	~~`generate_header`~~	
`auto_ptr_ref`	`get_pointer_safety`	`pointer_to_unary_function`
`binary_function`	`get_temporary_buffer`	`ptr_fun`
`binary_negate`	`get_unexpected`	`random_shuffle`
`bind1st`	`gets`	`raw_storage_iterator`
`bind2nd`	`is_literal_type`	`result_of`
`binder1st`	`is_literal_type_v`	`result_of_t`
`binder2nd`	`istrstream`	`return_temporary_buffer`
`codecvt_mode`		`set_unexpected`
	`iterator`	
`codecvt_utf16`	`little_endian`	`strstream`
`codecvt_utf8`	`mem_fun1_ref_t`	`strstreambuf`
`codecvt_utf8_utf16`	`mem_fun1_t`	`unary_function`
`const_mem_fun1_ref_t`	`mem_fun_ref_t`	`unary_negate`
`const_mem_fun1_t`	`mem_fun_ref`	`uncaught_exception`
`const_mem_fun_ref_t`	`mem_fun_t`	`undeclare_no_pointers`
`const_mem_fun_t`	`mem_fun`	`undeclare_reachable`
`consume_header`	`not1`	`unexpected_handler`
`declare_no_pointers`	`not2`	`wbuffer_convert`
`declare_reachable`	`ostrstream`	`wstring_convert`
	`pointer_safety`	
`generate_header`		

9.2 Update Annex C

S T I L L T O P R O V I D E W O R D S F O R A N N E X C

9.3 Strike from Annex D

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

1 The header `<iterator>` (25.2 [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*]

9.4 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.conversions *removed*
depr.conversions.buffer *removed*
depr.conversions.general *removed*
depr.conversions.string *removed*
depr.default.allocator *removed*
depr.istrstream *removed*
depr.istrstream.cons *removed*
depr.istrstream.general *removed*
depr.istrstream.members *removed*
depr.iterator *removed*
depr.locale.stdcvt *removed*
depr.locale.stdcvt.general *removed*
depr.locale.stdcvt.req *removed*
depr.mem.poly.allocator.mem *see*
mem.poly.allocator.mem
depr.ostrstream *removed*
depr.ostrstream.cons *removed*
depr.ostrstream.general *removed*
depr.ostrstream.members *removed*
depr.res.on.required *removed*
depr.string.capacity *removed*
depr.str.strstreams *removed*
depr.strstream *removed*
depr.strstream.cons *removed*
depr.strstream.dest *removed*

10 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.

11 References

[N4986] Thomas Köppe. 2024-07-16. Working Draft, Programming Languages — C++.
https://wg21.link/n4986

[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