Remove Deprecated Unicode Conversion Facets From C++26

Removing an underspecified feature from C++26

Document #:	P2871R0
Date:	2023-05-15
Project:	Programming Language C++
Audience:	Library Evolution Incubator
Revises:	N/A
Reply-to:	Alisdair Meredith <ameredith1@bloomberg.net>

Contents

1 Abstract **1**

2 Revision history. **1**
2.1 R0: Varna 2023 1

3 Introduction **1**

4 History of this feature **2**

5 Current implementations **2**

6 Proposal to remove from C++26 **2**

7 Wording **3**
7.1 Impact of removal 3
7.2 Proposed changes 3

8 Acknowledgements **6**

9 References **6**

1 Abstract

The `<codecvt>` header was first provided by C++11, and then deprecated by C++17 due to its underspecification, notably a lack of error handling. This paper proposes removing that header from the C++ Standard Library.

2 Revision history.

2.1 R0: Varna 2023

Initial draft of the 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, [P2863R0], 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 `<codecvt>` header in D.26 [depr.locale.stdcvt].

4 History of this feature

The `<codecvt>` header was originally proposed for C++11 by paper [N2007], and deprecated for C++17 by paper [P0618R0]. As noted at the time of deprecation, this feature was underspecified and would require more work than we wished to invest to bring it up to the expected standard.

One particularly notable quote from that initial deprecation review was provided by Beman Dawes, who sought expert opinions from the domain, outside of WG21:

"As the Unicode standard and security experts point out, ill-formed UTF can be and has been used as an attack vector. Yet the `<codecvt>` facets don't provide the safe forms of error handling as their defaults, and it is far to hard to use them in a safe way."

Reports from users in the field indicated repeated attempts to use this facility, as a part of the standard library bearing the most useful name suggested this should be our best practice, only to repeatedly fail for subtle reasons in our underspecification, leading to many home-grown solutions instead after much wasted time.

Since then SG16 has been convened and is producing a steady stream of work to bring reliable well-specified Unicode support to C++. However, no replacement facility is yet available as of C++23.

One fix for C++23 was provided by [P2736R2], which removed the last reference to an antiquated Unicode standard that had been retained purely so that this deprecated C++ facility could refer to obsolete-and-removed Unicode encoding, UCS2.

5 Current implementations

The 3 main standard libraries provide these classes since they supported C++11.

libc++ has been issuing deprecation warnings since Clang 15.0, released September 2022. libstdc++ does not yet issue any deprecation warnings for these facets. MSVC does not yet issue any deprecation warnings for these facets.

6 Proposal to remove from C++26

The paper proposes removing the whole of the `<codecvt>` header from C++26, even without a replacement in place. Library vendors' freedom to provide ongoing support will be preserved by adding all relevant names to 16.4.5.3.2 [zombie.names].

As noted above, the deprecated feature has been an active nuisance in the past, if not an outright security risk when dealing with malformed Unicode. There are no plans to rehabilitate these facets, with SG16 working on a better approach to Unicode text processing in general.

If we remove this feature outright, existing user code is unlikely to break as vendors exercise their freedoms under the zombie names clause. However, new code is even less likely to use these poorly specified tools, and detractors will not be able to point to the C++ standard.

When C++26 ships, this feature will have been deprecated for almost a decade, and three years longer than it was a part of the main library specification.

Note that the consensus of the review of this facility for C++23 was that the zombie names clause was sufficient support, and both SG16 and LEWG recommended removal in C++23. This did not occur as the author failed to update the whole Annex D review after [P2139R2].

7 Wording

Wording is relative to the latest working draft, [N4944].

7.1 Impact of removal

This feature was originally added at the Kona 2007 meeting, updating the example in the then recently added **[lib.conversions]** (now D.27.2 [depr.conversions.string]) to use these standard conversion facets rather than unspecified user-provided classes to illustrate use. Therefore, as we remove this library unilaterally, we must also amend that example to clearly rely on user-provided facets again.

The author has audited the current standard for any remaining usage of the UCS2 encoding removed below, and not found any.

7.2 Proposed changes

16.4.5.3.2 [zombie.names] Zombie names

1 In namespace `std`, the following names are reserved for previous standardization:

— `auto_ptr`,
— `auto_ptr_ref`,
— `binary_function`,
— `binary_negate`,
— `bind1st`,
— `bind2nd`,
— `binder1st`,
— `binder2nd`,
— `codecvt_mode,`
— `codecvt_utf16,`
— `codecvt_utf8,`
— `codecvt_utf8_utf16,`
— `const_mem_fun1_ref_t`,
— `const_mem_fun1_t`,
— `const_mem_fun_ref_t`,
— `const_mem_fun_t`,
— `consume_header,`
— `declare_no_pointers`,
— `declare_reachable`,
— `generate_header,`
— `get_pointer_safety`,
— `get_temporary_buffer`,
— `get_unexpected`,
— `gets`,
— `is_literal_type`,
— `is_literal_type_v`,
— `little_endian,`
— `mem_fun1_ref_t`,
— `mem_fun1_t`,
— `mem_fun_ref_t`,
— `mem_fun_ref`,

— `mem_fun_t`,
— `mem_fun`,
— `not1`,
— `not2`,
— `pointer_safety`,
— `pointer_to_binary_function`,
— `pointer_to_unary_function`,
— `ptr_fun`,
— `random_shuffle`,
— `raw_storage_iterator`,
— `result_of`,
— `result_of_t`,
— `return_temporary_buffer`,
— `set_unexpected`,
— `unary_function`,
— `unary_negate`,
— `uncaught_exception`,
— `undeclare_no_pointers`,
— `undeclare_reachable`, and
— `unexpected_handler`.

2 The following names are reserved as members for previous standardization, and may not be used as a name for object-like macros in portable code:

— `argument_type`,
— `first_argument_type`,
— `io_state`,
— `open_mode`,
— `preferred`,
— `second_argument_type`,
— `seek_dir`, and.
— `strict`.

3 The name `stossc` is reserved as a member function for previous standardization, and may not be used as a name for function-like macros in portable code.

4 The header names `<ccomplex>`, `<ciso646>`, `<codecvt>`, `<cstdalign>`, `<cstdbool>`, and `<ctgmath>` are reserved for previous standardization.

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

Change: Remove header `<codecvt>` and all its contents.

Rationale: The header was underspecified, and deprecated for over a decade. Ongoing support is at implementer's discretion, exercising freedoms granted by 16.4.5.3.2 [zombie.names].

Effect on original feature: A valid C++ 2023 program `#include`-ing the header may fail to compile. Code that uses any of the following names by importing the standard library modules may fail to compile:

— `codecvt_mode`
— `codecvt_utf16`
— `codecvt_utf8`
— `codecvt_utf8_utf16`
— `consume_header`
— `generate_header`
— `little_endian`

D.26 [depr.locale.stdcvt] Deprecated standard code conversion facets

D.26.1 [depr.locale.stdcvt.general] General

1 The header `<codecvt>` provides code conversion facets for various character encodings.

D.26.2 [depr.codecvt.syn] Header <codecvt> synopsis

```
namespace std {
  enum codecvt_mode {
      consume_header = 4,
      generate_header = 2,
      little_endian = 1
  };

  template<class Elem, unsigned long Maxcode = 0x10ffff, codecvt_mode Mode = (codecvt_mode)0>
    class codecvt_utf8 : public codecvt<Elem, char, mbstate_t> {
    public:
      explicit codecvt_utf8(size_t refs = 0);
      ~codecvt_utf8();
    };

  template<class Elem, unsigned long Maxcode = 0x10ffff, codecvt_mode Mode = (codecvt_mode)0>
    class codecvt_utf16 : public codecvt<Elem, char, mbstate_t> {
    public:
      explicit codecvt_utf16(size_t refs = 0);
      ~codecvt_utf16();
    };

  template<class Elem, unsigned long Maxcode = 0x10ffff, codecvt_mode Mode = (codecvt_mode)0>
    class codecvt_utf8_utf16 : public codecvt<Elem, char, mbstate_t> {
    public:
      explicit codecvt_utf8_utf16(size_t refs = 0);
      ~codecvt_utf8_utf16();
    };
}
```

D.26.3 [depr.locale.stdcvt.req] Requirements

1 For each of the three code conversion facets `codecvt_utf8`, `codecvt_utf16`, and `codecvt_utf8_utf16`:

- `Elem` is the wide-character type, such as `wchar_t`, `char16_t`, or `char32_t`.
- `Maxcode` is the largest wide-character code that the facet will read or write without reporting a conversion error.
- If `(Mode & consume_header)`, the facet shall consume an initial header sequence, if present, when reading a multibyte sequence to determine the endianness of the subsequent multibyte sequence to be read.
- If `(Mode & generate_header)`, the facet shall generate an initial header sequence when writing a multibyte sequence to advertise the endianness of the subsequent multibyte sequence to be written.
- If `(Mode & little_endian)`, the facet shall generate a multibyte sequence in little-endian order, as opposed to the default big-endian order.
- UCS-2 is the same encoding as UTF-16, except that it encodes scalar values in the range u+0000–u+ffff (Basic Multilingual Plane) only.

2 For the facet `codecvt_utf8`:

- The facet shall convert between UTF-8 multibyte sequences and UCS-2 or UTF-32 (depending on the size of Elem).
- Endianness shall not affect how multibyte sequences are read or written.
- The multibyte sequences may be written as either a text or a binary file.

3 For the facet `codecvt_utf16`:

— The facet shall convert between UTF-16 multibyte sequences and UCS-2 or UTF-32 (depending on the size of Elem).
— Multibyte sequences shall be read or written according to the `Mode` flag, as set out above.
— The multibyte sequences may be written only as a binary file. Attempting to write to a text file produces undefined behavior.

4 For the facet `codecvt_utf8_utf16`:

— The facet shall convert between UTF-8 multibyte sequences and UTF-16 (one or two 16-bit codes) within the program.
— Endianness shall not affect how multibyte sequences are read or written.
— The multibyte sequences may be written as either a text or a binary file.

D.27 [depr.conversions] Deprecated convenience conversion interfaces

D.27.2 [depr.conversions.string] Class template `wstring_convert`

1 Class template `wstring_convert` performs conversions between a wide string and a byte string. It lets you specify a code conversion facet (like class template `codecvt` (30.4.2.5.1 [locale.codecvt.general])) to perform the conversions, without affecting any streams or locales.

[*Example 1:* If you want to use ~~the~~a code conversion facet, `codecvt_for_utf8`, to output to `cout` a UTF-8 multibyte sequence corresponding to a wide string, but you don't want to alter the locale for `cout`, you can write something like:

```
  wstring_convert<~~std::~~codecvt_for_utf8<wchar_t>> myconv;
  std::string mbstring = myconv.to_bytes(L"Hello\n");
  std::cout << mbstring;
```

—*end example*]

```
namespace std {
  template<class Codecvt, class Elem = wchar_t,
           class WideAlloc = allocator<Elem>,
           class ByteAlloc = allocator<char>>
    class wstring_convert {
    public:
      // details elided
    };
}
```

2 The class template describes an object that …

8 Acknowledgements

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

Thanks again to Matt Godbolt for maintaining Compiler Explorer, the best public resource for C++ compiler and library archaeology, especially when researching the history of deprecation warnings!

9 References

[N2007] P.J. Plauger. 2006-04-15. Proposed Library Additions for Code Conversion.
https://wg21.link/n2007

[N4944] Thomas Köppe. 2023-03-22. Working Draft, Standard for Programming Language C++.
https://wg21.link/n4944

[P0618R0] Alisdair Meredith. 2017-03-02. Deprecating <codecvt>.
https://wg21.link/p0618r0

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

[P2736R2] Corentin Jabot. 2023-02-09. Referencing the Unicode Standard.
https://wg21.link/p2736r2