

Remove `wstring_convert` From C++26

Document #: P2872R1
Date: 2023-06-07
Project: Programming Language C++
Audience: Library Evolution
Reply-to: Alisdair Meredith
<ameredith1@bloomberg.net>

Contents

1 Abstract	1
2 Revision history	2
2.1 R1: June 2023 (SG16 telecon)	2
2.2 R0: May 2023 (pre-Varna)	2
3 Introduction	2
4 History	2
5 Deployment Experience	2
6 C++23 Feedback	3
6.1 Initial LEWGI Review: telecon 2020/07/13	3
6.2 SG16 Review: telecon 2020/07/22	3
6.3 LEWGI Consensus for C++23:	3
7 Recommendation for C++26	3
8 C++26 Feedback	3
8.1 SG16 (Unicode) review	3
8.2 LEWG initial review	4
9 Wording	4
10 Acknowledgements	9
11 References	9

1 Abstract

The `wstring_convert` library component has been deprecated since C++17. As noted at that time, the feature is underspecified and would require more work than we wish to invest to bring it up to standard. This paper proposes removing the deprecated convenience conversion interfaces, `wstring_buffer` and `wbuffer_convert`, from the C++ Standard Library.

2 Revision history

2.1 R1: June 2023 (SG16 telecon)

- fixed copy/paste where common text was clearly taken from another paper
- assigned SG16 as reviewer of first resort
- Provided full library wording against current draft, [N4950]
- Recorded when (or if) popular library implementations first warn of deprecation
- Thanked Matt Godbolt for Compiler Explorer
- Completed SG16 review

2.2 R0: May 2023 (pre-Varna)

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 convenience conversion interfaces, `wstring_buffer` and `wbuffer_convert`.

4 History

This feature was originally proposed for C++11 by paper [N2401] and deprecated for C++17 by paper [P0618R0]. As noted at the time, the feature was underspecified and would require more work than we wished to invest to bring it up to standard. Since then SG16 has been convened and is producing a steady stream of work to bring reliable well-specified Unicode support to C++.

There are currently 4 open LWG issues relating to this clause; there would be more, but we would rather see this feature removed than keep adding issues to deprecated library features.

- [LWG2478] Unclear how `wstring_convert` uses `cvtstate`
- [LWG2479] Unclear how `wbuffer_convert` uses `cvtstate`
- [LWG2480] Error handling of `wbuffer_convert` unclear
- [LWG2481] `wstring_convert` should be more precise regarding “byte-error string” etc.

5 Deployment Experience

The following program, based on an example in the Standard, was tested with Godbolt Compiler Explorer to determine when (or if) libraries started warning about the deprecation.

```
#include <codecvt>
#include <iostream>
#include <locale>
#include <string>

int main() {
    std::wstring_convert<std::codecvt_utf8<wchar_t>> myconv;
```

```
std::string mbstring = myconv.to_bytes(L"Hello\n");
std::cout << mbstring;
}
```

- libc++: First warns in Clang 15 (2022-09-06)
- libstdc++: Does not warn in latest release
- MSVC: Does not warn in latest release

6 C++23 Feedback

6.1 Initial LEWGI Review: telecon 2020/07/13

Discussion was broadly in favor of removal from the C++23 specification, and relying on library vendors to maintain source compatibility as long as needed. However, LEWGI explicitly requested to confer with SG16 in case that study group is aware of any reason to hold back on removal, before proceeding with the recommendation.

6.2 SG16 Review: telecon 2020/07/22

SG16 raises concerns that the original paper deprecating this feature lacked a strong motivation, as it was simply recording a recommendation from LWG review when deprecating the `<codecvt>` header for D.20. There is general concern that `codecvt` is not fit for purpose, notably due to poorly specified error handling capabilities while transcoding, but this deprecation does not address that, it is merely a convenience API for using that underspecified library component. While removing the `<codecvt>` header might mean there would be fewer standard `codecvt` facets to use with this API, it remains just as usable with user provided `codecvt` facets as before, as well as those in the `<locale>` header. While we would like to see a replacement facility, there is no such proposal at this time.

Polling showed no consensus to recommend the removal for C++23, but no objection to that removal either.

6.3 LEWGI Consensus for C++23:

Confirmed SG16 has no objection; remove this feature from C++23

7 Recommendation for C++26

Given vendors propensity to provide ongoing support for these names under the zombie name reservations, and following the LEWGI consensus for C++23, this paper proposed the removal of these interfaces from the C++26 Standard. Then close LWG issues [[LWG2478](#)], [[LWG2479](#)], [[LWG2480](#)], and [[LWG2481](#)] as Resolved by removal of the feature by this paper.

A weaker recommendation, if the library is retained, is to do the minimal work to clean up the wording to use the more precise terms that replaced Requires clauses ([\[P2874R0\]](#)), and wait until SG16 (or some other entity) produces a clean replacement for this facility for users to migrate to before removal.

8 C++26 Feedback

8.1 SG16 (Unicode) review

SG16 held a telecon on 2023/06/07, and reviewed this paper. The motivation given in the proposed Annex C wording was accepted, although LWG will likely want to make some updates in the wording review.

The main review comments were that one attendee observed that they had 16 uses in their code base, and all were an error should be replaced (and will be shortly)! Another attendee performed a Github code search, and found just 5 hits in the whole of Github.

The paper is forwarded to LEWG by unanimous consent.

8.2 LEWG initial review

Pending.

9 Wording

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

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`,
- `const_mem_fun1_ref_t`,
- `const_mem_fun1_t`,
- `const_mem_fun_ref_t`,
- `const_mem_fun_t`,
- `declare_no_pointers`,
- `declare_reachable`,
- `get_pointer_safety`,
- `get_temporary_buffer`,
- `get_unexpected`,
- `gets`,
- `is_literal_type`,
- `is_literal_type_v`,
- `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`,
- `wbuffer_convert`, and
- `wstring_convert`.

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

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

- `converted`,
- `from_bytes`,
- `stoss`, and
- `to_bytes`.

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

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

Change: Remove convenience interfaces `wstring_buffer` and `wbuffer_convert`.

Rationale: These features were underspecified with no clear error reporting mechanism, and deprecated for the last 3 editions of this standard. Ongoing support remains at the implementers' discretion, exercising freedoms granted by 16.4.5.3.2 [zombie.names].

Effect on original feature: A valid C++ 2023 program using these interfaces will not compile.

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

D.27.1 [depr.conversions.general] General

¹ The header `<locale>` (30.2) has the following additions:

```
namespace std {
    template<class Codecvt, class Elem = wchar_t,
            class WideAlloc = allocator<Elem>,
            class ByteAlloc = allocator<char>>
        class wstring_convert;
    template<class Codecvt, class Elem = wchar_t,
            class Tr = char_traits<Elem>>
        class wbuffer_convert;
}
```

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`) to perform the conversions, without affecting any

streams or locales.

[*Example 1:* If you want to use the code conversion facet `codecvt_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_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:
        using byte_string = basic_string<char, char_traits<char>, ByteAlloc>;
        using wide_string = basic_string<Elem, char_traits<Elem>, WideAlloc>;
        using state_type = typename Codecvt::state_type;
        using int_type = typename wide_string::traits_type::int_type;

        wstring_convert() : wstring_convert(new Codecvt) {}
        explicit wstring_convert(Codecvt* pcvt);
        wstring_convert(Codecvt* pcvt, state_type state);
        explicit wstring_convert(const byte_string& byte_err,
                                const wide_string& wide_err = wide_string());
        ~wstring_convert();

        wstring_convert(const wstring_convert&) = delete;
        wstring_convert& operator=(const wstring_convert&) = delete;
        wide_string from_bytes(char byte);
        wide_string from_bytes(const char* ptr);
        wide_string from_bytes(const byte_string& str);
        wide_string from_bytes(const char* first, const char* last);

        byte_string to_bytes(Elem wchar);
        byte_string to_bytes(const Elem* wptr);
        byte_string to_bytes(const wide_string& wstr);
        byte_string to_bytes(const Elem* first, const Elem* last);

        size_t converted() const noexcept;
        state_type state() const;

    private:
        byte_string byte_err_string; //exposition only
        wide_string wide_err_string; //exposition only
        Codecvt* cvtptr; //exposition only
        state_type cvtstate; //exposition only
        size_t cvtcount; //exposition only
    };
}
```

² The class template describes an object that controls conversions between wide string objects of class `basic_string<Elem, char_traits<Elem>, WideAlloc>` and byte string objects of class

`basic_string<char, char_traits<char>, ByteAlloc>`. The class template defines the types `wide_string` and `byte_string` as synonyms for these two types. Conversion between a sequence of `Elem` values (stored in a `wide_string` object) and multibyte sequences (stored in a `byte_string` object) is performed by an object of class `Codecvt`, which meets the requirements of the standard code-conversion facet `codecvt<Elem, char, mbstate_t>`.

³ An object of this class template stores:

- `byte_err_string` — a byte string to display on errors
- `wide_err_string` — a wide string to display on errors
- `cvtptr` — a pointer to the allocated conversion object (which is freed when the `wstring_convert` object is destroyed)
- `cvtstate` — a conversion state object
- `cvtcount` — a conversion count

```
size_t converted() const noexcept;
```

⁴ *Returns:* `cvtcount`.

```
wide_string from_bytes(char byte);
wide_string from_bytes(const char* ptr);
wide_string from_bytes(const byte_string& str);
wide_string from_bytes(const char* first, const char* last);
```

⁵ *Effects:* The first member function shall convert the single-element sequence `byte` to a wide string. The second member function shall convert the null-terminated sequence beginning at `ptr` to a wide string. The third member function shall convert the sequence stored in `str` to a wide string. The fourth member function shall convert the sequence defined by the range `[first, last)` to a wide string.

⁶ In all cases:

- If the `cvtstate` object was not constructed with an explicit value, it shall be set to its default value (the initial conversion state) before the conversion begins. Otherwise it shall be left unchanged.
- The number of input elements successfully converted shall be stored in `cvtcount`.

⁷ *Returns:* If no conversion error occurs, the member function shall return the converted wide string. Otherwise, if the object was constructed with a wide-error string, the member function shall return the wide-error string. Otherwise, the member function throws an object of class `range_error`.

```
state_type state() const;
```

⁸ *Returns:* `cvtstate`.

```
byte_string to_bytes(Elem wchar);
byte_string to_bytes(const Elem* wptr);
byte_string to_bytes(const wide_string& wstr);
byte_string to_bytes(const Elem* first, const Elem* last);
```

⁹ *Effects:* The first member function shall convert the single-element sequence `wchar` to a byte string. The second member function shall convert the null-terminated sequence beginning at `wptr` to a byte string. The third member function shall convert the sequence stored in `wstr` to a byte string. The fourth member function shall convert the sequence defined by the range `[first, last)` to a byte string.

¹⁰ In all cases:

- If the `cvtstate` object was not constructed with an explicit value, it shall be set to its default value (the initial conversion state) before the conversion begins. Otherwise it shall be left unchanged.
- The number of input elements successfully converted shall be stored in `cvtcount`.

¹¹ *Returns:* If no conversion error occurs, the member function shall return the converted byte string. Otherwise, if the object was constructed with a byte-error string, the member function shall return the byte-error string.

Otherwise, the member function shall throw an object of class `range_error`.

```
explicit wstring_convert(Codecvt* pcvt);
wstring_convert(Codecvt* pcvt, state_type state);
explicit wstring_convert(const byte_string& byte_err,
    const wide_string& wide_err = wide_string());
```

12 *Requires:* For the first and second constructors, `pcvt != nullptr`.

13 *Effects:* The first constructor shall store `pcvt` in `cvtptr` and default values in `cvtstate`, `byte_err_string`, and `wide_err_string`. The second constructor shall store `pcvt` in `cvtptr`, `state` in `cvtstate`, and default values in `byte_err_string` and `wide_err_string`; moreover the stored state shall be retained between calls to `from_bytes` and `to_bytes`. The third constructor shall store new `Codecvt` in `cvtptr`, `state_type()` in `cvtstate`, `byte_err` in `byte_err_string`, and `wide_err` in `wide_err_string`.

```
~wstring_convert();
```

14 *Effects:* The destructor shall delete `cvtptr`.

D.27.3 [depr.conversions.buffer] Class template `wbuffer_convert`

1 Class template `wbuffer_convert` looks like a wide stream buffer, but performs all its I/O through an underlying byte stream buffer that you specify when you construct it. Like class template `wstring_convert`, it lets you specify a code conversion facet to perform the conversions, without affecting any streams or locales.

```
namespace std {
    template<class Codecvt, class Elem = wchar_t, class Tr = char_traits<Elem>>
        class wbuffer_convert : public basic_streambuf<Elem, Tr> {
        public:
            using state_type = typename Codecvt::state_type;
            wbuffer_convert() : wbuffer_convert(nullptr) {}
            explicit wbuffer_convert(streambuf* bytebuf,
                Codecvt* pcvt = new Codecvt,
                state_type state = state_type());
            ~wbuffer_convert();

            wbuffer_convert(const wbuffer_convert&) = delete;
            wbuffer_convert& operator=(const wbuffer_convert&) = delete;

            streambuf* rdbuf() const;
            streambuf* rdbuf(streambuf* bytebuf);

            state_type state() const;
        private:
            streambuf* bufptr;           //exposition only
            Codecvt* cvtptr;           //exposition only
            state_type cvtstate;       //exposition only
        };
}
```

2 The class template describes a stream buffer that controls the transmission of elements of type `Elem`, whose character traits are described by the class `Tr`, to and from a byte stream buffer of type `streambuf`. Conversion between a sequence of `Elem` values and multibyte sequences is performed by an object of class `Codecvt`, which shall meet the requirements of the standard code-conversion facet `codecvt<Elem, char, mbstate_t>`.

3 An object of this class template stores:

— `bufptr` — a pointer to its underlying byte stream buffer

- `cvtptr` — a pointer to the allocated conversion object (which is freed when the `wbuffer_convert` object is destroyed)
- `cvtstate` — a conversion state object

```
state_type state() const;
```

⁴ *Returns:* `cvtstate`.

```
streambuf* rdbuf() const;
```

⁵ *Returns:* `bufptr`.

```
streambuf* rdbuf(streambuf* bytebuf);
```

⁶ *Effects:* Stores `bytebuf` in `bufptr`.

⁷ *Returns:* The previous value of `bufptr`.

```
explicit wbuffer_convert(
    streambuf* bytebuf,
    Codecvt* pcvt = new Codecvt,
    state_type state = state_type());
```

⁸ *Requires:* `pcvt != nullptr`.

⁹ *Effects:* The constructor constructs a stream buffer object, initializes `bufptr` to `bytebuf`, initializes `cvtptr` to `pcvt`, and initializes `cvtstate` to `state`.

```
~wbuffer_convert();
```

¹⁰ *Effects:* The destructor shall delete `cvtptr`.

10 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!

11 References

[LWG2478] Jonathan Wakely. Unclear how `wstring_convert` uses `cvtstate`.

<https://wg21.link/lwg2478>

[LWG2479] Jonathan Wakely. Unclear how `wbuffer_convert` uses `cvtstate`.

<https://wg21.link/lwg2479>

[LWG2480] Jonathan Wakely. Error handling of `wbuffer_convert` unclear.

<https://wg21.link/lwg2480>

[LWG2481] Jonathan Wakely. `wstring_convert` should be more precise regarding “byte-error string” etc.

<https://wg21.link/lwg2481>

[N2401] P.J. Plauger. 2007-09-03. Code Conversion Facets for the Standard C++ Library.

<https://wg21.link/n2401>

- [N4950] Thomas Köppe. 2023-05-10. Working Draft, Standard for Programming Language C++.
<https://wg21.link/n4950>
- [P0618R0] Alisdair Meredith. 2017-03-02. Deprecating <codecv>.
<https://wg21.link/p0618r0>
- [P2139R2] Alisdair Meredith. 2020-07-15. Reviewing Deprecated Facilities of C++20 for C++23.
<https://wg21.link/p2139r2>
- [P2863R0] Alisdair Meredith. 2023-05-19. Review Annex D for C++26.
<https://wg21.link/p2863r0>
- [P2874R0] Alisdair Meredith. 2023-05-19. Mandating Annex D.
<https://wg21.link/p2874r0>