`char_traits`: Stop the bleeding!

Document #:	P3681R0
Date:	2025-05-19
Programming Language C++	
Audience:	SG-16

Reply-to: Corentin Jabot <corentin.jabot@gmail.com>

Abstract

We propose to deprecate the use of user-defined types for the Traits template parameter of `std::basic_string`, `std::basic_string_view` and iostream-related types. More importantly, we argue that `std::zstring_view` should not be encumbered by char traits.

Motivation

Classes such as `basic_string`, `basic_string_view`, and the stream types can be customized with a user-provided *CharTraits* type. This customization has limited use cases, a non-negligible impact on compile times, symbol sizes, and diagnostics messages.

It is also not designed to properly handle Unicode and Unicode character types.

While getting rid of `char_traits` or the `Traits` template parameters is unrealistic, we should consider that this customization is not worth its cost, and unburden in-flight and future proposals such as `zstring_view` P3655R0 [1].

Problems

`CharTraits` does not handle multi-bytes encodings

Every *CharTraits* `requirement` is specified to operate on single values, *even* for member functions operating on sequences such as `compare` and `copy`. Consequently, using it to transform multibyte strings will lead to incorrect results. Comparisons might also be incorrect for shift-state encodings and other non-UTF multibyte encodings.

CharTraits provides functionalities unrelated to code units

It is hard to imagine use cases for a non-default implementation of `assign`, `length`, `move`, `copy`. `not_eof`, `eof` and other functions are only useful for `iostream`-related facilities and conflate byte reading and encoding concerns. It is also hard to see how they would be implemented differently from `std::char_traits`. Presumably, a user-provided `Traits` type could circumvent LWG2959 [3]. However, that this issue has not been fixed (which would be an ABI break), illustrates the lack of interest in specializing the stream types for `charN_t`.

`CharTraits` encourages bloat

Consider:

```
void f(std::unordered_map<std::zstring_view, std::zstring_view>);
```

The presence of `char_traits` would make the mangled name 25% larger using the MSVC mangling scheme. Which, for widely used types, adds up. This would be justifiable if `char_traits` provided any value whatsoever… but it does not.

Note that this is less of a problem for the Itanium ABI, which compresses repeated type names.

CharTraits worsen teachability of text-related notions

By being incompatible with modern encodings and with its design conflating text encoding, *CharTraits CharTraits* is too often used to encourage incorrect text handling. This is also inconsistent with the rest of `std::string` and `string`-like types, which expose an interface that is otherwise only designed to manipulate sequences of values, without text semantics or encoding considerations.

A plan of action

1. No `Traits` parameters in new types

`zstring_view` and any other such future types or interface should forgo the `Traits` customization. The consistency argument is not a very strong one. Never acknowledging mistakes is not useful for the teachability of the language. We have an opportunity for `zstring_view` to produce diagnostics and be faster to compile.

There is, however, the question of the conversion from `std::basic_string` to `std::basic_zstring_view` (for example). For which, we have two adequate solutions:

- Silently drop the user-defined trait type, recognizing that case-insensitivity is a property of a transformation and not an intrinsic property of the string (and any other use of user-provided `Traits` would be equally assumed to not affect the invariant of the string).
- Make such conversion ill-formed (forcing users to manually construct the string_view with, for example, `zstring_view(str.data(), str.size() + 1)`.

2. `char_traits` should not be necessary

The one intrinsic value of `char_traits` is to provide the size of a null-terminated sequence. `char_traits<T>::length(str)` is not the most intuitive way to spell `std::strlen`, so we should add `constexpr` overloads of `strlen` for all character types. Note that the abandoned P1944R0 [2] proposed to make `strlen` constexpr, but that paper was abandoned, and it does not solve the lack of `strlen` overloads for `char8_t`, `char16_t`, `char32_t`. To be clear, we are not proposing to deprecate `std::char_traits` in this paper, but we also want to ensure its use is unnecessary.

3. Replacing off-label *CharTraits* use cases (Casing)

The one use case we often see purported for user-defined CharTraits types is to let string classes do case-insensitive comparisons. Which, of course, only works for non-multibyte encoding. This is partly because there is no easy way to do that in C++. Because case-insensitive comparisons are a very reasonable thing to want to do, we should facilitate them. [P3688R0] proposes facilities to compare ranges of ASCII characters. We could also add function objects to ease use with associative containers. Independently, we should pursue casing and folding Unicode views in the C++29 time frame (upper casing or lower casing are not correct ways to compare Unicode sequences).

4. Deprecating user-provided `Traits`

We should deprecate specialization of any standard type that has a template parameter defaulted to a specialization of `char_traits`, when the corresponding template argument differs from the default value.

In other words, `basic_string<char, MyTrait<char>>` should be deprecated. This is something that implementation can diagnose with depreciation warnings (at least as long as the specialization of `basic_string` is complete).

5. Longer-term prospects

We could, in a future version of C++, make `basic_string<char, MyTrait<char>>` ill-formed. This would allow an implementation not to use Traits in its implementations.

Because the `Traits` parameter is often not the last parameter in standard types that use it, it would be a serious breaking change to remove it. And because of ABI concerns, it's unlikely to ever be removed from implementations.

However, if it is neither used nor user-provided, we solve most problems created by char traits, except that:

- Providing an allocator to string will remain a bit cumbersome (which we could solve by adding an alias template)
- Symbol size will not improve.

However, if an implementation decided to change its ABI one day, or provide a way to opt-out of ABI stability, knowing the `Trait` parameter need not be mangled could lead to improvements.

References

[1] Peter Bindels, Hana Dusikova, and Jeremy Rifkin. P3655R0: zstring_view. https://wg21.link/p3655r0, 3 2025.

[2] Daniil Goncharov and Antony Polukhin. P1944R0: Add constexpr modifiers to functions in cstring and cwchar headers. https://wg21.link/p1944r0, 12 2019.

[3] Jonathan Wakely. LWG2959: char_traits<char16_t>::eof is a valid utf-16 code unit. https://wg21.link/lwg2959.

[P3688R0] Jan Schultke, Corentin Jabot *ASCII character utilities* https://wg21.link/P3688R0

[N5008] Thomas Köppe *Working Draft, Standard for Programming Language C++* https://wg21.link/N5008