Clarifying the interaction of the literal and execution encodings

Document #:	P3670R0
Date:	2025-04-16
Programming Language C++	
Audience:	SG-16

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

Motivation

C++ juggles multiple encodings per type (literal, execution, and environment). We like to consider them separate because they are controlled by different actors (person compiling the program versus the person running it). However, we found it is hardly possible to reason about the behavior of a program that makes that distinction.

A char is just an integer, and interpreting it differently depending on whether it ”comes from” a literal (across copies and other transformations) or not is simply not possible. Same for multibyte characters and strings.

It is also not something users can be expected to do. Users will concatenate strings originating from literals, execution, and files and expect a sensible outcome.

SG-16 has spent countless hours (not a hyperbole!) discussing the merit of referring to literal versus execution encodings for various interfaces, recently thread names, standard exceptions, `format`/`print`.

In particular, `std::format` assumes that the literal encoding is a good proxy for the encoding of the format string and any string argument, even if all of these elements can be in the execution encoding.

We decided a thread name was also in the literal encoding because that theoretically leads to better outcomes on some platforms - because the execution encoding is degraded at startup (Per the C standard (7.11.2 The `setlocale` function)), whereas the literal encoding is not. Yet thread names functions being system interfaces deal in string in the execution encoding.

The reason these decisions pan out in practice is that

- On many platforms, these encodings are always the same anyway.
- In places where they aren’t, no amount of wording will lead to a better or worse outcome.

Internet was a mistake

In the days of yore, compiling a program in the environment where it would run was the norm. When the compiler picks the execution encoding on the platform it runs on, and the compiled

program runs on the same platform, mojibake does not occur. How people deployed their C++ software changed, and C++ did not adapt.

The Windows conundrum

Most Linux, Apple, and Android systems default to UTF-8 for the execution encoding. Windows doesn't. And yet, we see users compile with the `/utf8` MSVC flag - a flag that sets both the source file encoding and the literal encoding. (`/execution-charset`).

Clang-cl only supports UTF-8 as execution encoding, and all packages in the vcpkg repository use UTF-8 as the execution encoding.

So, Windows is a platform where the literal and execution encoding are routinely different. You would expect that in places where the standard infers the execution encoding from the literal encoding, things would go haywire. Yet they seemingly do not - or rather fine enough not to inconvenience a majority of users.

This paradox can be explained by the fact we have started to leverage Unicode support in Windows, bypassing standard utilities and their locale-related behavior (std::print calls `WriteConsoleOutputW` on Windows), and in part because things have been broken long enough that users learn not to rely on locale-related standard features to handle characters outside of the basic character sets.

Mojibake, normatively

So, we are in a situation where conflating the literal encoding and the execution encoding is incorrect. Yet, it is inevitable. The status quo is unfixable yet seemingly good enough.

I'm trying to get us (SG-16) to acknowledge that we cannot reason about environments where literal and execution encoding differ. And that this is fine. And to codify it in the standard. Again, this is a precedent set by `std::print`, and no one complained.

In particular, what we want to do is hint at mojibake in some scenarios. In previous discussions, people were concerned about describing Mojibake as UB.

I think UB is certainly the most apt description, given that unintentionally changing the meaning of text is certainly not reasonable from a correctness standpoint, and that arguably garbled text can be leveraged to exploit vulnerabilities.

However, any formulation that roughly said, "This is not a use case that we can support, and you get what you get" would be an acceptable outcome.

Proposed change

This paper adds non-normative notes reflecting our understanding. No functional change is proposed.

Future work

Ultimately, there isn't much we can do better interpret the encoding of an arbitrary `char`, Besides tracking what the industry is doing and educating our users on the pitfalls of mixing different encodings, and on the effects of various flags and platform settings.

A lot of languages have solved that problem by assuming UTF-8 consistently, in line with the UTF-8 everywhere, the idea being that if every single actor in a system independently assumes the same encoding (UTF-8), then mojibake simply cannot happen.

Alas, C++ is less in a position to make that shift today, at least not on every platform. We should, however, improve `char8_t`, as this type communicates its encoding clearly.

We should also revisit that a program behaves as-if `setlocale(LC_ALL, "C")` is called at startup. Changing the locale makes sense - arguably (and certainly can't be changed). However, if the initial locale is, for example, `ja_JP.utf8`, then we probably want the initial locale to be set by `setlocale(LC_ALL, "C.utf8")` on platforms where this is available. The encoding is a property of the environment and, ideally, would not be modified by the program.

Wording

🔲 Character sequences [character.seq]

🔲 General [character.seq.general]

The C standard library makes widespread use of characters and character sequences that follow a few uniform conventions:

- Properties specified as *locale-specific* may change during program execution by a call to `setlocale(int, const char*)` [clocale.syn], or by a change to a `locale` object, as described in [locales] and [input.output].
- The *execution character set* and the *execution wide-character set* are supersets of the basic literal character set [lex.charset]. The encodings of the execution character sets and the sets of additional elements (if any) are locale-specific. Each element of the execution wide-character set is encoded as a single code unit representable by a value of type `wchar_t`. [*Note:* The encodings of the execution character sets can be unrelated to any literal encoding. — *end note*]
  [*Note:* If any element of the literal character set does not have the same (or any) representation in the execution encoding as it does in the literal encoding, passing a sequence of characters encoded in the literal encoding to a standard library function expecting an argument in the execution encoding can produce unexpected effects or result in undefined behavior.

  Similarly, library functions, which expect their arguments in the literal encoding, may produce unexpected effects or result in undefined behavior when passed character sequences in the execution encoding, which are not valid in the literal encoding.] — *end note*]

[*Note:* Sequences of characters are never assumed to be in the execution or wide execution encodings during constant evaluation] — *end note*]

References

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