Preserving LC_CTYPE at program start for UTF-8 locales

Document #:	P3677R0 | N3539
Date:	2025-05-03
Programming Language C++	
Audience:	SG-16, SG-22, WG14
Reply-to:	Corentin Jabot <corentin.jabot@gmail.com>

This is a C paper

The locale at the start of the program is fundamentally a property of the C standard library implementation. So this paper targets C. However because it affects C++ too, and because SG16 would probably have the most expertise, it is also targeting WG21 SG-16 and SG-22 subgroups.

Motivation

C locales, designed initially around single byte (and stateful) encodings, conflate locales and encoding. See P2020R0 [1] for a broader analysis of C locales.

At the start of a program, `setlocale(LC_ALL, "C")` is called - or rather, a program behaves as if this call was made. This is specified in the C Standard (7.11.2 The `setlocale` function).

The encoding of the "C" locale is not specified (by either C or POSIX), and that encoding will vary across platforms. But it is usually either ASCII (or IBM-037 on IBM platforms)

In general, The C standard only specifies how the C locale behaves in regard to the ctype classification functions (and codepoint classification is a character set property that is orthogonal to localization)

POSIX has a more complete definition of a C locale. (and sadly, they made using UTF-8 as the encoding of the C locale non-conforming in 2013).

The needs for the `C/POSIX` locale arise from wanting `C` program to be initially blissfully unaware of the idiosyncrasies of various cultures. A classic example is the French locale use of commas as a floating point delimiter. This confuses everyone, especially developers, who usually don't like to think about localization too much. Critically, an application might have many users or target audiences; it might be a server with connections from all over the world, produce logs, etc. Starting the program in a locale-agnostic mode is, therefore, sensible. The `C` locale is, by the way, distinct from the `en_US` locale.

In short, `C` is the non-locale locale, and `C` programs (and by extension C++ programs) start in a localisation-agnostic mode.

Which is great.

Except it isn't.

On most systems, this has the effect of not only changing the locale but also resetting the assumed environment encoding from UTF-8 to ASCII. Indeed, in a lot of systems, such as most flavors of Linux, Mac and iOS systems, Android, FreeBSD, etc, the associated encoding of the user's environment's locale is UTF-8.

Because the C locale is historically ASCII, and because the C standard mandates the use of the C locale, the C standard is effectively mandating that C programs running in a UTF-8 environment pretend they don't know UTF-8 exists. This, in turn, is the source of encoding issues (Mojibake).

That problem has been identified, and sometime around 2014, some distributions started to ship a "C.UTF-8" locale that has the same properties as the C locale (including in terms of character classification) except that the associated locale encoding is UTF-8.

That practice was upstreamed in GlibC 2.35 (in 2022). Both MUSL and Bionic (Android's C library implementation) use a UTF-8 locale by default (in fact, that's the only locale supported by Bionic).

To quote the original glibc proposal:

> Modern systems need a modern encoding system to deal with global data. The old customs of parsing data as ASCII (or ISO 8859-1) is long past and has no business in the 21st century. People still hitting Mojibake today is deplorable.
> However, there is no way today to select UTF-8 encoding without also picking a country/language locale. Many projects hardcode en_US.UTF-8, or maybe try one or two more (like en_GB.UTF-8 and de_DE.UTF-8), before giving up and failing. This is also why distros often do not select a UTF-8 locale by default since the related locale attributes are undesirable.

So, there are platforms for which a locale-agnostic UTF-8 locale exists but is not used by C programs, which instead decay to ASCII. We should encourage the use of a UTF-8 locale-agnostic locale when that would be appropriate.

- A C UTF-8 locale exists
- The encoding associated to the environment locale (`setlocale(LC_CTYPE, "")`) is also a UTF-8 locale.

That second point is important. We should preserve the encoding of the environment and not force UTF-8 when it is not expected by the parent process. Indeed, the primary use case for the environment encoding is to exchange information with the environment. Which is what is being proposed here.

Given that the C standard does not preclude the C locale from having a UTF-8 encoding and that there is no "C.UTF-8" locale (and that the "C" locale is not specified in great detail, the wording is somewhat vague but should be enough to encourage implementations. Note that there have been discussions of adding a UTF-8 C locale to POSIX; however, it has yet to materialize.

Impact

This change would only impact platforms where a C.UTF-8 locale is available. It affects conversion functions, and queries of the environment encoding. Characters classification functions - which are unsuitable to handle Unicode codepoints, anyway - are not affected. GlibC claims `LC_COLLATE` is affected and that they can sort strings using the codepoint order. However, UTF-8 is such that the codepoint order is the same as the byte order anyway. GlibC has a build option to make C.UTF-8 the default locale, and there has been some mention of making it the default. However, I could not find more information on that.

OSX can produce a UTF-8 C locale by setting `LC_ALL` to `"C"` and `LC_CTYPE` to "UTF-8". On Windows, the POSIX locales are emulated on top of Windows APIs that already decouple encoding and localization, and they could make the default locale UTF-8 when the codepage is `CP_UTF8`. In the common case, however, the code page on Windows is not UTF-8 and this paper would therefore have no impact.

Languages such as Python 3 already made this change and to some extent partially motivated the `C.UTF-8` locale.

Wording

A value of `"C"` for locale specifies the minimal environment for C translation; a value of `""` for locale specifies the locale-specific native environment.

Other implementation-defined strings may be passed as the second argument to `setlocale`.

At program startup, the equivalent of

```
    setlocale(LC_ALL, ~~"C"~~ LOCALE);
```

Where `LOCALE` is

- an implementation-defined string that describes a locale with the same properties as the locale designated by "C" except that the encoding associated with that locale is "UTF-8" if such a locale exists and if the associated encoding of the locale designated by `""` is UTF-8, or
- "C" otherwise.

[*Note:* The associated encoding of the "C" locale is implementation-defined and may be UTF-8. — *end note*]

A call to the `setlocale` function may introduce a data race with other calls to the setlocale function or with calls to functions that are affected by the current locale. The implementation shall behave as if no library function calls the setlocale function.

Annex: test program

```
#include <stdio.h>
#include <locale.h>
#include <langinfo.h>
#include <cstdlib>

int main() {
    printf("LANG: \%s\nLC_CTYPE: \%s\n",
    getenv("LANG"),
    getenv("LC_CTYPE"));

    printf("default: \%s\n", nl_langinfo(CODESET));

    setlocale(LC_ALL, "C");
    printf("C: \%s\n", nl_langinfo(CODESET));

    setlocale(LC_ALL, "C.UTF-8");
    printf("UTF-8: \%s\n", nl_langinfo(CODESET));
}
```

References

[1] Corentin Jabot. P2020R0: Locales, encodings and unicode. https://wg21.link/p2020r0, 1 2020.

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