Locales, Encodings and Unicode

Document #: P2020R0
Date: 2020-05-17
Project: Programming Language C++
Audience: SG-16, LEWG
Reply-to: Corentin Jabot <corentin.jabot@gmail.com>

Hindsight is 20/20

Abstract

This paper aims to list and explain the issues with facilities provided by the `<locale>` header and related facilities, with a focus on Unicode. We try to explain what is the scope and challenges of localization, how it relates to encoding in the context of C++.

By listing some of the issues and limitations of the current interfaces, we hope to provide the basis for fruitful discussions so that we can ultimately lay a direction to fix them.

It's complicated

Wikipedia defines a locale as "**a set of parameters that defines the user's language, region and any special variant preferences that the user wants to see in their user interface**".

There are over 4000 spoken languages, hundreds of scripts (over 140 supported by Unicode) and 195 countries. While platforms will not support as many locales as there are cultures, new locales are added regularly to platforms. The Common Locale Data Repository used by most platforms currently support over 700 locales.

Such diversity of culture challenges the assumptions we make about text, writing systems time, numbers and other localized information.

Locale and text encoding are separate concerns

For historical reasons, POSIX and later C and C++ adopted a model that conflates text encoding and locale. Therefore, changing the locale affects the encoding and the encoding is not exposed directly by C or C++ interfaces.

This of course stemmed from the fact that most, if not all, character sets used at the time these things were standardized were only able to represent a fraction of existing characters, and so were designed to be able to represent the characters most in use in a given language, or region.

There is, however, no such thing as region-specific characters (this sentence contains the characters 로케일, a word meaning ”locale” in Korean).

Locale, in the context of C++, is better understood as the set of rules governing the display and transformation of textual information, for a desired region, language, or culture.

What should be affected by locale

In a vacuum, a piece of text has an encoding, whether explicitly or not: the existence of an encoding is what defines text as opposed to a sequence of bytes that does not have an associated encoding.

Text, however, does not have an associated locale: a given piece of text is unaffected by locale. But locale may affect how that text was created and how it may be transformed.

The following text operations may be affected by locales:

- Casing (upper casing, lower casing, title casing)
- Collation
- Search
- Breaking by characters, words, and sentences
- Text rendering

The case of a given character is not affected by regional preferences. Properties of abstract characters are immutable across languages and cultures. For example, while uppercasing **ss** may yield different results across languages, **s** is universally a lower case letter. This is true regardless of the character set used. The question of whether **s**, **S** or **ß** are representable in a given encoding is a separate concern.

Formatting of numbers, units, time and other measurements are also affected by locale. However, the unit of a given value is not affected by locale. The user may have preferred units for some measurement in a given locale but that may require conversion of the value.

Note that because of Han Unification, some CJK Unicode characters may need to be shaped according to locale during text rendering. While this is not a concern, it is worth keeping in mind if the committee wanted to standardize features involving rendering text.

Numeral systems

C and C++ make the assumption that numbers are represented with the Arabic numerals system, when scanning and printing them. However there exist many numeral systems in use, most of which require characters only supported by multi bytes encoding.

Timezones and Calendars

Timezones are not related to locales. A timezone may span multiple locale regions and a single region may extend to many locales. However, locales do have an associated preferred calendar. For a given calendar, such as the Gregorian calendar, locales may differ on what days are considered week days, or by which day is considered the first of the week. This doesn't seem to be an issue in C++ at the present as a single calendar is provided.

Measurement Units

In the context of [**?**], it is useful to keep in mind that locales affect:

- What is the preferred unit for a given measurement (Kilometers vs Miles) - although in many cases this can be very context dependent.
- How units are formatted (suffixes, prefixes, presentation of ratios and fractions).

Specific issues with locale handling in C++

Character Classification

- Classification functions are affected by locales and work on code unit.
- Classifications functions should be encoding dependent but not locale-dependent.
- Abstract characters properties are not affected by locale.

`std::isalpha( 'a' )` should be universally true.

But the question answered by `isalpha` is whether the code unit corresponding to **a** is representable in the global or specified locale - assuming the global locale associated encoding is a superset of the execution encoding-. This often leads these functions to be misused.

More critically these functions operate on code units rather than codepoints.

As such it is not possible to use them with locales associated with multi-bytes encoding such as UTF-8, Shift-JS. P1628 aims at fixing these issues for Unicode specifically.

Nevertheless, it might be necessary to deprecate the C functions (taking no locale) and introduce better overloads, such as:

```
bool isalpha(text_encoding, uint32_t codepoint);
```

for the entire set of classification functions.

By dissociating from locale, it would possible to provide constexpr implementation for some encodings. Reorganizing the order of parameters makes the functions usable with bind_front.

This would be accompanied by functions to convert a range of code units to a code point for the given encoding.

Alternatively, we might consider providing classification functions for Unicode only (see P1628 [**?**]), along with functions to convert a range of code units of a given encoding to a single Unicode codepoint. This would reduce the aggregated volume of data needed for every supported encoding while separating the two conflated question `std::isalpha` tries to answer:

- Is this character representable by that encoding?
- Is this abstract character known to be a letter?

Only providing classification for Unicode would greatly reduce the complexity of both the API and implementation and several people expressed a desire to only support Unicode classification. Others encoding would be supported by explicit codepoint conversion functions.

`tolower/toupper` are broken

The two characters-based transformations tolower and toupper functions provided by C and C++ suffer from all the same issues as classification functions (see the previous section). They do make an additional broken assumption: they return a single value representing a single code unit. `std::toupper` is declared as follows:

```
template< class charT >
charT toupper( charT ch, const locale& loc );
```

These functions being transformations they may, in fact, be dependent of locale (regardless of encoding), the canonical example being the letter `i` which might be uppercased `I` or `İ` (LATIN CAPITAL LETTER I WITH DOT ABOVE in a Turkish locale).

There are also cases where a single codepoint is mapped to multiple code points when operating a case transformation. An example is ß which might be uppercased as SS. The German language recently adopted ẞ (LATIN CAPITAL LETTER SHARP S), both SS and ẞ being valid and widespread upper casing of ß.

And so a more accurate interface would be

```
auto toupper(text_encoding encoding,
uint32_t codepoint) -> rangeof<codepoint>;
auto toupper(text_encoding encoding, std::locale locale, uint32_t codepoint) -> rangeof<codepoint>;
```

Note that, while there are some use cases for the codepoint-based casing, correct casing requires to be done over an entire string or word.

Unfortunately, the ctype functions operating on character sequences are also specified to operate on a code unit per code unit basis.

Notably, many Greek letters will have different case transformations depending on the position within a word or the preceding or succeeding letter. Because of the complexity of casing in the general case, it might be interesting to provide multiple sets of interfaces.

```
auto ascii_toupper(char) -> char;
auto unicode_toupper(codepoint, std::locale) -> rangeof<codepoint>;
```

The encoding of the system is not preserved at the start of the program

The [**?**] standard specifies (C17 7.11.1.1.5) that at program startup, the equivalent of `setlocale(LC_ALL, "C");` is executed.

There is no doubt that having the default behavior of C++ be independent of regional and localization settings is the right default. Many programs do not interact with users or need consistent behavior across environments that require they should not be affected by regionalization preferences.

However, locale and encoding being tied under the `POSIX` and `C` models, the `"C"` locale specifies a limited character set that can be represented by the `ASCII` or `EBCDIC` encoding - although an encoding is not specified.

However, in many cases, the environment is set up to handle UTF-8. On Linux, OSX, Android, iOS and many other environments, the environment encoding is UTF-8. The locale will often be of the form `lang_COUNTRY.UTF-8`.

By setting the locale to `"C"` at the start of the program, the encoding information is lost. While P1885 [**?**] offers a mechanism to query the environment encoding, we think the encoding information should be preserved.

More generally, the encoding used by the global locale (and therefore the standard default I/O functions) cannot be modified by the program without encoding issues (Mojibake) when doing I/O. Programs should respect the environment(s) encoding. Therefore C++ (nor C) cannot force the encoding associated to be UTF-8 either. While mentioned frequently, this is simply something outside of our purview. Languages that do force UTF-8 are either less portable than C++ or force text conversions on I/O in environments that are not UTF-8. Several options can improve the situation

1. The initial call to `setlocale(LC_ALL, "C")` should respect the encoding, such that if the locale environment is `xx_YY.FOOBAR`, the call to `setlocale` should be `setlocale(LC_ALL, "C.FOOBAR")` on platforms on which such locale exist. For example, if the environment locale is `"en_GB.UTF-8"`, the locale at the beginning of the program should be `"C.UTF-8"`.
2. `setlocale` should not modify the encoding implicitly or at all. Under a better model, locale and encoding should not be tied. However, not all encodings can represent all locales. In fact, only encodings mapping the entire Unicode character sets can represent all locale.

Instead of passing the desired encoding in the locale string (i.e. `"en_US[.ENCODING]"`), `setlocale` could take an extra parameter `setlocale(int category, const char* name, text_encoding narrow_encoding);`

locale is a global state

Calls to setlocale are notoriously not thread-safe.

The C standard (ISO/IEC 9899:2017 7.11.1.5) mentions

> 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.

This has a couple of well-documented issues:

- The lack of thread safety make usages of functions depending on the global locale racy and as such unusable in multi-threaded programs (the majority of programs going forward).
- The existence of a mutable global state force libraries to fend each other against improper `setlocale` usage, knowing there is no proper usage of `setlocale`.

This stem from a few incorrect assumptions:

- **All text processing on behalf of a user is done in the same locale**. In practice, programs will need to use both the C locale (the non-locale) and the user environment locale for different tasks.
- **A program only has one user at any given time**. In practice some applications such as web servers, communications applications may serve many users with different locales. Or an application may print receipts for clients of many different nationalities, etc.

I don't think there are many ways to fix that.

Functions depending implicitly on the global locale need to be replaced and deprecated. This work has started with `std::format` which, by default is independent of locale and provides localization for dates and numbers as an explicit opt-in.

A first step towards that work would be to list the list of locale-specific behavior in the standard (direct and indirect).

`iostream` and `regex` are not impacted as they take ownership of a copy of the global locale. Having a global object is not bad per se (as long as the locale constructor and global methods were thread-safe) as a means to have preferred locale developers can refer to, as long as it is not used implicitly.

Still, it might be interesting to give `iostreams` a constructor accepting a locale in addition to the `imbue` method.

Facets and `iostream` are tightly coupled

Some facets, including `num_get`, `num_put`, `time_get`, `time_put`, etc are tightly coupled with `iostream`. This makes them unusable with non-iostream based interfaces such that `std::format` and the proposed text parsing facility [**?**] - which then have to resort to `numpunct`, `moneypunct`.

Text conversions function have a hard to use interface and poor error management

See [**?**] for these issues and solutions. In any, case transcoding is affected by encoding but not by locale.

Facets do not support standard UTF types

In addition to the many functions failing to account for multi bytes encoding (including all the classification and codepoint transformation functions), facets are not required to have a specialization for `char8_t`, `char16_t`, `char32_t`

Some facets do more than localization

Once again because encoding and locales are hopelessly tied, and because many encodings have small charsets, many encodings only have one or a few currency symbols. Sometimes a currency symbol such as **$** and a generic currency symbol (¤ CURRENCY SIGN).

Because of that, the symbol used for currency formatting is tied to the locale in use. But of course, the currency is tied to the value, not to the locale (if you send $10 to a European, they should get $10, not 10 €).

Selecting the correct currency is a difficult exercise [**?**]

Solutions to this problem include:

- Deprecating monetary formatting
- Introducing a strong type to represent an amount + a currency along with ways to format it (ICU approach)
- Or alternatively, introduce a facility that let the user specify which symbol or name to use ([**?**] approach), and which precision to use (precision is currently tied to locale)

Translation support is not required in the standard

Translation support - wherein at runtime a string, format string or key will be replaced by a translation suitable for the language and region is a more complex problem than the formatting of standard types.

Notably, translation requires tooling support, pluralization (with sometimes complex rules), support for positional arguments. The Standard can help with some of these issues (for example `std::format` supports positional arguments), but given the many frameworks and tools that exist for translation and the complexity involved, we think this is a problem best left for third party libraries. A quick search reveals that the existing translation facility, `std::messages` has very little use in open source code. It is tailored for `catgets`, does not support pluralization and has no implementation suited to some operating systems such as Android or iOS.

Locale names are not portable across platforms and implementation.

Different implementations might use or accept slightly different scheme as locale names:

```
Greek_Greece.ACP
el_GR
el-GR.UTF-8
```

Some implementations (OSX, ICU) support a script parameter, ie `el_Latn_GR`.

Some implementations (notably ICU) support additional properties `el_GR@collation=phonebook`.

Moreover, constructing a locale requires the user to resort to string manipulation and once constructed it is hard to extract individual components from the locale.

A solution might be to better specify what format(s) and capabilities need to be supported as well as adding more explicit constructors to the locale object, for example

```
class locale {
    locale(string_view lang, string_view country, string_view script = {});
    void set_option(string_view key, string_view value);
    string_view country() const;
    string_view language() const;
    string_view script() const;
    string_view option(string_view key) const;
    text_encoding encoding() const;
};
```

Additionally, it is not possible for a user to know which locales are supported - nor for them to implement fallback mechanisms. A way to list supported locales and country/lang combination would be incredibly useful.

Qt provides:

```
    QLocale::matchingLocales( QLocale::AnyLanguage, QLocale::AnyScript, QLocale::AnyCountry);
```

ICU has `Locale::getAvailableLocales()`

Where do we go from there?

The design of locale in C++ is inherited and built upon the C model, itself mirroring the POSIX model.

That design has been adopted by many platforms and popular languages (including Python) and it seems difficult to get rid of it.

Some of the issues described here might benefit from input from the C committee. However, fixing these issues in C is not strictly necessary. In fact, on some platforms such as Windows, the POSIX behavior has to be emulated as win32 API differentiate locale and encoding already.

The key point in C++ is to deprecate any use of functions that rely implicitly on a global locale and to provide the necessary replacements.

Here is a short summary of the possible directions envisioned by the present paper:

- Use the preferred environment encoding when setting the global `"C"` locale at program startup
- Document and deprecates facilities depending on a global locale, either directly or indirectly
- Deprecate ctype classification functions and replace them with functions working on codepoints and text encoding.
- Or, alternatively, deprecate ctype transformation functions and replace them with functions working on **Unicode** codepoints exclusively & text encoding and returning codepoint sequences
- Separate locale from encoding and decide on a case by case basis what each function should operate on.
- Provides classification and transformation functions for Unicode, possibly `constexpr`
- Provides `constexpr` classification and transformation functions for ASCII
- Provide transcoding function not related to locale
- Provide functions supporting multi bytes encoding and deprecate or fix functions which do not
- Deprecate or improve locales facilities that are tightly coupled to `iostream`
- Better specify what are valid locale names and provide better ways to construct and query locales and their components
- Provide a way to list supported locales
- Deprecate facilities that change the semantic of values upon locale-based formatting such as `money_put`/`money_get`
- Deprecate incomplete and not widely used facilities such as `std::messages`
- Add support for `char8_t`, `char16_t` in locale facilities

Given the scope of the work I am not quite sure whether the best path forward is to deprecate all of `<locale>` and create an entirely new set of facilities or to incrementally fix what we have.

Guideline for future C++ facilities

- Facilities providing byte manipulation should depend on neither locale nor encoding
- Facilities doing text classification should depend on encoding explicitly

- Facilities doing text transformation, sorting or display should provide overloads for a locale-independent, possibly `constexpr` mode whenever applicable and a locale dependent consuming a locale object (like `std::format` does) In doubt, read the Unicode documentation

Acknowledgments

With many thanks to Matteo Italia, Thiago Macieira, Zach Laine for providing valuable feedback and much needed proof-reading.

[TR35] Unicode® *Unicode Language and Locale Identifiers*
http://unicode.org/reports/tr35/

[C Standard] https://web.archive.org/web/20181230041359if_/http://www.open-std.org/jtc1/sc22/wg14/www/abq/c17_updated_proposed_fdis.pdf

[QT6] *Qt 6 RFC: Defaulting to or enforcing UTF-8 locales on Unix systems - Thiago Macieira*
https://lists.qt-project.org/pipermail/development/2019-October/037791.html

[Case] *Truths programmers should know about case - James Bennett*
https://www.b-list.org/weblog/2018/nov/26/case/

[Boost] *Boost Locale documentation*
https://www.boost.org/doc/libs/1_72_0/libs/locale/doc/html/rationale.html#rationale_why

[QLocale] *Qt QLocale documentation*
https://doc.qt.io/qt-5/qlocale.html

[ICU] *ICU Locale documentation*
http://userguide.icu-project.org/locale

[POSIX] *POSIX Locales specification*
https://pubs.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap07.html

[Stackoverflow] *Currency format in C++*
https://stackoverflow.com/questions/51481646/currency-format-in-c