Naming Text Encodings to Demystify Them

Document #: P1885R0
Date: 2019-10-06
Project: Programming Language C++
Audience: SG-16, LEWG
Reply-to: Corentin Jabot <corentin.jabot@gmail.com>

If you can't name it, you probably don't know what it is
If you don't know what it is, you don't know what it isn't
Tony Van Eerd

Target

C++23

Abstract

For historical reasons, all text encodings mentioned in the standard are derived from a locale object, which does not necessarily match the reality of how programs and system interact.

This model works poorly with modern understanding of text, ie the Unicode model separates encoding from locales which are purely rules for formatting and text transformations but do not affect which characters are represented by a sequence of code units.

Moreover, the standard does not provide a way to query which encodings are expected or used by the system, leading to guesswork and unavoidable UB.

This paper introduces the notions of literal encoding, system encoding and a way to query them.

Use cases

This paper aims to make C++ simpler by exposing information that is currently hidden to the point of being perceived as magical by many. It also leaves no room for a language below C++ by ensuring that text encoding does not require the use of C functions.

The primary use cases are:

- Ensuring a specific string encoding at compile time
- Ensuring at runtime that string literals are compatible with the system encoding

- Custom conversion function
- locale-independent text transformation

The many text encodings of a C++ system

Text in a technical sense is a sequence of bytes to which is virtually attached an encoding. Without encoding, a blob of data simply cannot be interpreted as text.

In many cases, the encoding used to encode a string is not communicated along with that string and its encoding is therefore presumed with more or less success.

Generally, it is useful to know the encoding of a string when

- Transferring data as text between systems or processes (io)
- Textual transforming of data
- Interpretation of a piece of data

In the purview of the standard, text i/o text originates from

- The source code (literals)
- The iostream library as well as system functions
- Environment variables and command-line arguments intended to be interpreted as text.

Locales provide text and transformations and conversions facilities and as such, in the current model have an encoding attached to them.

There are therefore 3 sets of encodings of primary interest:

- The encoding of narrow and wide characters and string literals
- The narrow and wide encodings used by a program when sending or receiving strings from its environment
- The encoding of narrow and wide characters attached to a std::locale object

[*Note:* Because they have different code units sizes, narrow and wide strings have different encodings. char8_t, char16_t, char32_t literals are assumed to be respectively UTF-8, UTF-16 and UTF-32 encoded. — *end note*]

[*Note:* A program may have to deal with more encoding - for example, on windows, the encoding of the console attached to cout may be different from the system encoding.

Likewise depending on the platform, paths may or may not have an encoding attached to them, and that encoding may either be a property of the platform or the filesystem itself. — *end note*]

The standard only has the notion of execution character sets (which implies the existence of execution encodings), whose definitions are locale-specific. That implies that the standard assumes that string literals are encoded in a subset of the encoding of the locale encoding.

This has to hold notably because it is not generally possible to differentiate runtime strings from compile-time literals at runtime.

This model does, however, present several shortcomings:

First, in practice, C++ software are often no longer compiled in the same environment as the one on which they are run and the entity providing the program may not have control over the environment on which it is run.

Both POSIX and C++ derives the encoding from the locale. Which is an unfortunate artifact of an era when 255 characters or less ought to be enough for anyone. Sadly, the locale can change at runtime, which means the encoding which is used by ctype and conversion functions can change at runtime. However, this encoding ought to be an immutable property as it is dictated by the environment (often the parent process). In the general case, it is not for a program to change the encoding expected by its environment. A C++ program sets the locale to "C" (see [N2346], 7.11.1.1.4) (which assumes a US ASCII encoding) during initialization, further losing information.

Many text transformations can be done in a locale-agnostic manner yet require the encoding to be known - as no text transformation can ever be applied without prior knowledge of what the encoding of that text is.

More importantly, it is difficult or impossible for a developer to diagnose an incompatibility between the locale-derived, encoding, the system-assumed encoding and the encoding of string literals.

Exposing the different encodings would let developers verify that that the system environment is compatible with the implementation-defined encoding of string literals, aka that the encoding and character set used to encode string literals are a strict subset of the encoding of the environment.

Identifying Encodings

To be able to expose the encoding to developers we need to be able to synthesize that information. The challenge, of course, is that there exist many encodings (hundreds), and many names to refer to each one. Fortunately there exist a database of registered encoding covering almost all encodings supported by operating systems and compilers. This database is maintained by IANA through a process described by [rfc2978].

This database lists over 250 registered character sets and for each:

- A name
- A unique identifier
- A set of known aliases

We propose to use that information to reliably identify encoding across implementation and systems.

Design Considerations

Encodings are orthogonal to locales

The following proposal is mostly independent of locales so that the relevant part can be implemented in an environment in which `<locale>` is not available, as well as to make sure we can transition `std::locale` to be more compatible with Unicode.

Naming

SG-16 is looking at rewording the terminology associated with text and encoding throughout the standard, this paper does not yet reflect that effort.

However "system encoding" and "literal encoding" are descriptive terms. In particular "system" is illustrative of the fact that a C++ program has, in the general case, no control over the encoding it is expected to produce and consume.

MIBEnum

We provide a `text_encoding::id` enum with the MIBEnum value of a few often used encodings for convenience. Because there is a rather large number of encodings and because this list may evolve faster than the standard, it was pointed out during early review that it would be detrimental to attempt to provide a complete list. [*Note:* MIB stands for Management Information Base, which is IANA nomenclature, the name has no particular interest beside a desire not to deviate from the existing standards and practices. — *end note*]

Name and aliases

The proposed API offers both a name and aliases. The `name` method reflects the name with which the text_encoding object was created, when applicable. This is notably important when the encoding is not registered, or its name differs from the IANA name.

Implementation flexibility

This proposal aims to be implementable on all platforms as such, it supports encoding not registered with IANA, does not impose that an implementation is aware of all registered encoding, and it let implementers provide their own aliases for IANA-registered encoding.

`const char*`

A primary use case is to enable people to write their own conversion functions. Unfortunately, most APIs expect NULL-terminated strings.

Example

```
#include <text_encoding>
#include <iostream>

void print(const std::text_encoding & c) {
        std::cout << c.name()
        << " (iana mib: " << c.mib() << ")\n"
        << "Aliases:\n";
        for(auto && a : c.aliases()) {
                std::cout << '\t' << a << '\n';
        }
}

int main() {
        std::cout << "Literal Encoding: ";
        print(std::text_encoding::literal());
        std::cout << "Wide Literal Encoding: ";
        print(std::text_encoding::wide_literal());
        std::cout << "System Encoding: ";
        print(std::text_encoding::system());
        std::cout << "Wide system Encoding: ";
        print(std::text_encoding::wide_system());
}
```

compiled with `g++ -fwide-exec-charset=EBCDIC-US -fexec-charset=SHIFT_JIS`, this program may display:

```
Literal Encoding: SHIFT_JIS (iana mib: 17)
Aliases:
        Shift_JIS
        MS_Kanji
        csShiftJIS
Wide Literal Encoding: EBCDIC-US (iana mib: 2078)
Aliases:
        EBCDIC-US
        csEBCDICUS
System Encoding: UTF-8 (iana mib: 106)
Aliases:
        UTF-8
        csUTF8
Wide sytem Encoding: ISO-10646-UCS-4 (iana mib: 1001)
Aliases:
        ISO-10646-UCS-4
        csUCS4
```

Implementation

The following proposal has been prototyped using a modified version of GCC to expose the encoding information.

On windows, the run-time encoding can be determined by `GetACP` - and then map to MIB values, while on POSIX platform it corresponds to value of `nl_langinfo` when the user (`""`) locale is set - before the program's locale is set to `C`.

On OSX `CFStringGetSystemEncoding` and `CFStringConvertEncodingToIANACharSetName` can also be used.

While exposing the literal encoding is novel, a few libraries do expose the system encoding, including Qt and wxWidget, and use the IANA registry.

Future work

Exposing the notion of text encoding in the core and library language give use the tools to simplify some problems in the standard.

Notably, it offers a sensible way to do locale-independent, encoding aware padding in `std::format` as in described in [P1868].

While this give us the tools to handle encoding, it does not fix the core wording.

Proposed wording

(which is known to be terrible)

New header <text_encoding>

A `text_encoding` describe a text encoding portably across platforms by exposing data from the Character Sets databased described by [rfc2978] and [rfc3808].

```
namespace std {

struct text_encoding {
    enum id {
        other = 1,
        unknown = 2,
        ascii = 3,
        latin1 = 4,
        utf8 = 106,
        ucs4 = 1001,
        utf16be = 1013,
        utf16le = 1014,
        utf16 = 1015,
        utf32 = 1017,
        reserved = 3000
    };

    constexpr text_encoding(const char* name);

    constexpr int mib() const noexcept;
    constexpr const char* name() const noexcept;

    constexpr auto aliases() const noexcept -> see below;

    constexpr bool operator==(const text_encoding & other) const;
    constexpr bool operator==(text_encoding::id mib) const;

    static consteval text_encoding literal();
    static consteval text_encoding wide_literal();

    static text_encoding system();
    static text_encoding wide_system();

    static text_encoding for_locale(const std::locale&);
    static text_encoding wide_for_locale(const std::locale&);

    private:
        unsigned mib_; // exposition only
        std::string name_; // exposition only
    };
}
```

A *register-character-set* is a character set registered by the process described in [rfc2978] and which is known of the implementation.

Let `bool COMP_NAME(const char* a, const char* b)` be a function that returns `true` if two ASCII string are identical equal, ignoring case and all `-` and `_` characters.

`constexpr text_encoding(const char* name);`

> For each implementation-defined alias `a` of each *register-character-set*, if `COMP_NAME(a, name.c_str())` is true, initialize `mib_` with the `MIBenum` associated with that *register-character-set*. Otherwise initialize `mib_` with `text_encoding::id::other`.
>
> *Ensures:* `name == name_`.

`constexpr int text_encoding::mib() const noexcept;`

> *Returns:* `mib_`.
>
> [*Note:* The enumerator value `text_encoding::id::unknown` is provided for compatibility with [rfc3808], `text_encoding::mib()` never returns `text_encoding::id::unknown`. — *end note*]

`constexpr const char* name() const noexcept;`

> *Returns:* `name_.c_str()`.

`constexpr auto text_encoding::aliases() const noexcept;`

> *Returns:* an implementation defined object `r` representing a sequences of aliases such that:
>
> - `ranges::view<decltype(r)>` is true,
> - `ranges::random_access_range<decltype(r)>` is true,
> - `same_as<ranges::range_value_t<decltype(r)>, string_view>` is true,
> - `!ranges::empty(r) || mib() == id::other` is true.
>
> If `mib()` is equal to the `MIBEnum` value of one of the *register-character-sets*, `r[0]` is the name of the *register-character-set*.
>
> `r` contains the aliases of the *register-character-set* as specified by [rfc2978].
>
> `r` may contain implementation defined values.
>
> `r` does not contain duplicated values - The equality of 2 values is determined by `COMP_NAME`.
>
> [*Note:* The order of elements in `r` is unspecified. — *end note*]

`constexpr bool text_encoding::operator==(const text_encoding & other) const;`

> *Returns:* `name() == other.name()` if `mib() == id::other && other.mib() == id::other` is true, otherwise `mib() == other.mib()`.

`constexpr bool text_encoding::operator==(text_encoding::id i) const;`

> *Returns:* `(mib() != id::other)? mib() == i : false`.

`static consteval text_encoding text_encoding::literal();`

> *Returns:* a `text_encoding` object representing the encoding used to encode narrow characters and string literals.

`static consteval text_encoding text_encoding::wide_literal();`

> *Returns:* a `text_encoding` object representing the encoding used to encode wide characters and string literals.

`static text_encoding text_encoding::system();`

> Return the presumed system narrow encoding. On POSIX systems this is the encoding attached to the user locale (`""`) at the start of the program.
>
> [*Note:* This function should always return the same value during the lifetime of a program and is not affected by calls to `setlocale`. — *end note*]

`static text_encoding text_encoding::wide_system();`

> Return the presumed system wide encoding. On POSIX systems this is the encoding attached to the user locale (`""`) at the start of the program.
>
> [*Note:* This function should always during the same value during the lifetime of a program and is not affected by calls to `setlocale`. — *end note*]

In <locale>

```
namespace std {
    text_encoding text_encoding_for_locale(const std::locale&);
    text_encoding text_encoding_wide_for_locale(const std::locale&);
}
```

`text_encoding text_encoding_for_locale(const std::locale& loc);`

> *Returns:* The text encoding for narrow string associated with the locale `loc`.

`text_encoding text_encoding_wide_for_locale(const std::locale& loc);`

> *Returns:* The text encoding for wide strings associated with the locale `loc`

Acknowledgments

Many thanks to Victor Zverovich and Thiago Macieira for reviewing this work and providing valuable feedback.

References

[N4830] Richard Smith *Working Draft, Standard for Programming Language C++*
https://wg21.link/n4830

[N2346] *Working Draft, Standard for Programming Language C*
http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2346.pdf

[rfc3808] I. McDonald *IANA Charset MIB*
https://tools.ietf.org/html/rfc3808

[rfc2978] N. Freed *IANA Charset MIB*
https://tools.ietf.org/html/rfc3808

[Character Sets] IANA *Character Sets*
https://www.iana.org/assignments/character-sets/character-sets.xhtml

[iconv encodings] GNU project *Iconv Encodings*
http://git.savannah.gnu.org/cgit/libiconv.git/tree/lib/encodings.def

[P1868] Victor Zverovich *Clarifying units of width and precision in std::format*
http://wg21.link/P1868