Naming Text Encodings to Demystify Them

Document #:	P1885R1
Date:	2020-01-10
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.

Revisions

Revision 1

- Add more example and clarifications
- Require hosted implementations to support all the names registered in [rfc3808].

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

Non goals

This facility aims to help **identify** text encodings and does not want to solve encoding conversion and decoding. Future text encoders and decoders may use the proposed facility as a way to identify their source and destination encoding. The current facility is *just* a fancy name.

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 (I/O)
- Textual transformation 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 transformation and conversion 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 implementations 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*]

The enum is purposefully not an `enum class` so that it can be easily compared to objects from third party librairies such as `QTextCodec`.

The enumerators `unknown` and `other` and their value are provided by the very same RFC such as:

- `other` designs an encoding not registered in the IANA Database, such that 2 encoding with the `other` mib are identical if their name compare equal.
- `unknown` is used when the encoding could not be determined. Under the current proposal, only default constructing a `text_encoding` object can produce that value. The encoding associated with the locale or environment is always known.

While `MIBEnum` was necessary to make that proposal implementable consistently across platforms, its main purpose is to remediate to the fact that an encoding can have multiple inconsistent names across implementations.

However,

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 a freestanding implementation is aware of all registered encodings, and it let implementers provide their own aliases for IANA-registered encoding. Because the process for registering encoding is documented [rfc2978] implementations can (but are not required to) provide registered encodings not defined in [rfc3808] - in the case that rfc is updated out of sync of the standard. However, [rfc3808] is from 2004 and has not been updated. As the world converge to utf-8, new encodings are less likely to be registered.

Implementations may not extend the `text_encoding::id` as to guarantee source compatibility.

`const char*`

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

Example

Listing the encoding

```
#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
```

LWG3314

[time.duration.io] specifies that the unit for micro seconds is µ on systems able to display it. This is currently difficult to detect and implement properly.

The following allows an implementation to use µ if it is supported by both the execution encoding and the encoding attached to the stream.

```
template<class traits, class Rep, class Period>
void print_suffix(basic_ostream<char, traits>& os, const duration<Rep, Period>& d)
```

```
{
        if constexpr(text_encoding::literal() == text_encoding::utf8) {
                if (os.getloc().encoding() == text_encoding::utf8) {
                        os << d.count() << "\u00B5s"; // μ
                        return;
                }
        }
        os << d.count() << "us";
}
```

A more complex implementation may support more encodings, such as iso-8859-1.

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 environment ("") 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 gives us the tools to solve 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

Add the header `<text_encoding>` to the "C++ library headers" table in `[headers]`, in a place that respects the table's current alphabetic order.

Add the macro `__cpp_lib_text_encoding` to `[version.syn]`, in a place that respects the current alphabetic order:

```
#define __cpp_lib_text_encoding 201911L (**placeholder**) // also in text_encoding
```

Add a new header `<text_encoding>`.

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

```
namespace std {

struct text_encoding final{
    enum id : unsigned {
        other = 1,
        unknown = 2,
        ascii = 3,
        latin1 = 4,
        utf8 = 106,
        utf16 = 1015,
        utf32 = 1017,
        reserved = 3000
    };

    constexpr explicit text_encoding(string_view name);

    constexpr id mib() const noexcept;
    constexpr string_view 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();

    private:
        id mib_; // exposition only

        //FIXME this may be std::string_view in freestanding
        std::string name_; // exposition only
```

```
        };
    }
```

A *registered-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 strings are identical equal, ignoring case and all `-` and `_` characters.

`constexpr explicit text_encoding(string_view name);`

Effects: If there exists an implementation-defined alias `a` of *registered-character-set* such that `COMP_NAME(a, name.c_str())` is `true`, initialize `mib_` with the `MIBenum` associated with that *registered-character-set*. Otherwise, initialize `mib_` with `text_encoding::id::other`.

Implementations must return a valid `text_encoding` object for every `name` that matches either an alias or a name of a *registered-character-set* listed in [rfc3808].

[*Note:* Freestanding implementations are not required to provide this method — *end note*]

Ensures: `name_ == name`.

`constexpr id 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 string_view name() const noexcept;`

Returns: `name_`.

`constexpr auto aliases() const noexcept;`

Returns: an implementation-defined object `r` representing a sequence 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 *registered-character-sets*, `r[0]` is the name of the *registered-character-set*.

`r` contains the aliases of the *registered-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 operator==(const text_encoding & other) const;`

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

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

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

`static consteval text_encoding literal();`

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

`static consteval text_encoding wide_literal();`

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

`static text_encoding system();`

Return the presumed system narrow encoding. On POSIX systems this is the encoding attached to the environment 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 wide_system();`

Return the presumed system wide encoding. On POSIX systems this is the encoding attached to the environment 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*]

In [locale]:

```
namespace std {
  class locale {
  public:
    [...]

    // locale operations
    string name() const;

    text_encoding encoding() const;
    text_encoding wide_encoding() const;

  };
}
```

In [locale.members]:

`string name() const;`

Returns: The name of `*this`, if it has one; otherwise, the string `"*"`.

```
text_encoding encoding() const;
```

Returns: The text encoding for narrow strings associated with the locale `*this`.

```
text_encoding wide_encoding() const;
```

Returns: The text encoding for wide strings associated with the locale `*this`.

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