Naming Text Encodings to Demystify Them

Document #:	P1885R10
Date:	2022-02-14
Programming Language C++	
Audience:	LEWG

Reply-to:	Corentin Jabot <corentin.jabot@gmail.com>
	Peter Brett <pbrett@cadence.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 systems interact.

This model works poorly with modern understanding of text, i.e., 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 environment, leading to guesswork and unavoidable UB.

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

Examples

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 << "environment Encoding: ";
    print(std::text_encoding::environment());
}
```

Compiled with `g++ -fexec-charset=SHIFT_JIS`, this program may display:

```
Literal Encoding: SHIFT_JIS (iana mib: 17)
Aliases:
    Shift_JIS
    MS_Kanji
    csShiftJIS

environment Encoding: UTF-8 (iana mib: 106)
Aliases:
    UTF-8
    csUTF8
```

LWG3314

[time.duration.io] specifies that the unit for microseconds is μ on environments 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.

Asserting a specific encoding is set

On POSIX, matching encodings is done by name, which pulls the entire database. To avoid that, we propose a method to asserting that the environment encoding is as expected. such method mixed to only pull in the strings associated with this encoding:

```
int main() {
    return text_encoding::environment_is<text_encoding::id::UTF8>();
}
```

User construction

To support other use cases such as interoperability with other libraries or internet protocols, `text_encoding` can be constructed by users

```
text_encoding my_utf8("utf8");
assert(my_utf8.name() == "utf8"sv); // Get the user provided name back
assert(my_utf8.mib() == text_encoding::id::UTF8);

text_encoding my_utf8_2(text_encoding::id::UTF8);
assert(my_utf8_2.name() == "UTF-8"sv); // Get the preferred name for the implementation
assert(my_utf8_2.mib() == text_encoding::id::UTF8);
assert(my_utf8 == my_utf8_2);
```

Unregistered encoding

Unregistered encoding are also supported. They have the `other` mib, no aliases and are compared by names:

```
text_encoding wtf8("WTF-8");
assert(wtf8.name() == "WTF-8"sv);
assert(wtf8.mib() == text_encoding::id::other);

//encodings with the \tcode{other} mib are compared by name, ignoring case, hyphens and underscores
assert(wtf8 == text_encoding("___wtf8__"));
```

Revisions

Revision 10

- Address more concerns raised by P2498R1 [2], and subsequent SG16 suggestions.

Revision 9

LEWG forwarded P1885R8 to LWG, however:

- There existed at the time a draft revision mandating `CHAR_BIT == 8`, which was approved by SG-16 and presented to LEWG. Unfortunately, it was not the revision that was polled due to a mixed up on my part.
- There was some sustained opposition to the paper during electronic polling I wanted to address.

- Two papers have been published which proposed changes to the design. I also wanted to address both of them.
- Wording fixes
- **Remove the wide methods to address concerns raised by P2491r0 [7]**
- Address concerns raised by P2498R0 [1]
- Mandates CHAR_BIT == 8
- Remove the ”object representation” wording.

Revision 8

- Add some prose to explain the expected behavior of the `wide_environment` methods, especially that it does not aim to return encodings always suitable for use with the wide-character functions. This is accompanied by wording notes.
- Add some wording to clarify that the way an implementation maps an encoding to a `registered-character-set` is implementation-defined. This is, for example, to allow implementations to return the Big5 encoding, as the name as registered may actually describe slightly different characters sets. Nevertheless, there are existing practices to respect, and we do not want to constrain implementation too much. In general, while most registered character encodings do describe a very specific set of characters and mapping, some do not. This blanket wording allows an implementation to offer a behavior that matches existing expectations.
- Add more recommended practices in the wording
- Specify that the encoding of literals is that of the object representation.
- Specify that the encodings are encoding schemes.
- Always map UTF16-LE/BE to UTF-16 (and add prose to explain why).
- Fix an example, which while correct was not representative of desired recommended practices.
- Many wording fixes.

Revision 7

- Improve the wording of `aliases()`. Make its return type part of the API. `aliases_view`. The value_type of that view is `const char*`. it was `string_view` in previous versions by mistake. This view is `borrowed_trange`, and `random_access_range` at LEWG’s request. We inherit from `view_interface` for greater usuability. All of that has been implemented. Note that this view is not `common_range` because it can be implemented more efficiently without that requirement, and, being copyable, it can be adapted into one.
- Modify the wording of `text_encoding::environment` to account for the fact locale-related environment variables can be changed at runtime. Therefore, it is not possible to enforce

that the value returned by `text_encoding::environment` is not affected by variable locale changes. However, it is implementable on some systems (We provide an implementation for Windows, FreeBSD, Linux, OSX), and on these systems, we recommend, but not mandate that changing the environment variables do not affect the value returned by `text_encoding::environment`. To account for that, the environment() functions are no longer `noexcept`.

- Clarify how the names returned by `name()` and `aliases()` relate. In particular, modify the wording to call `aliases().front()` *primary name* rather than `prefered name` as to avoid confusion.
- Clarify that `name()` can be `nullptr`.
  For example, consider `text_encoding{text_encoding::id::unknown}.name()`.
- Many wording tweaks and correction

Revision 6

- Update the list of encoding to add UTF7IMA which was registered this year.
- Replace references of [rfc3808] by [ianacharset-mib] which is the maintained list since 2004
- Explain why the underlying type of text_encoding::id is `int_least32_t`.

Revision 5

- Add motivation for name returning `const char*`
- Improve wording
- Rename `system` to `environment`
- Remove freestanding wording - will be handled separately
- Exclude a couple of legacy encodings that are problematic with the name matching algorithm

Revision 4

- Change `operator==(encoding, mib)` for `id::other`
- Add wording for freestanding
- Improve wording
- Improve alias comparison algorithm to match unicode TR22

Revision 3

- Add a list of encodings NOT registed by IANA
- Add a comparative list of IANA/WHATWG
- Address names that do not uniquely identify encodings
- Add more examples

Revision 2

- Add all the enumerators of rcf 3008
- Add a mib constructor to `text_encoding`
- Add `environment_is` and `wide_environment_is` function templates

Revision 1

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

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 environment 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 to identify their source and destination encoding. The current facility is *just* a fancy name.

Addressing concerns raised by P2498R1 [2]and subsequent discussions

The present paper is designed as an interface over the IANA database. The presence of an encoding in the IANA database is precisely what the invariant of the class is (even if non-IANA encodings are **fully** supported through the `mib` encoding.)

P2498R0 claims that "This registry [IANA] is known to be incomplete and, in some respects, does not provide a perfect match to the requirements of C++." While it is true that not all encoding are registered with IANA, P1885 support arbitrary encodings. IANA is used to establish equivalence between encodings identified by different names across platforms or on the same platform.

P2498R1 [2] proposes an implementation-defined mapping from some implementation-defined enum types to 'text_encoding::id'. This would actually, given the current and proposed wording, allow implementations to use a 16 bits type instead of a 32 bits type to store the mib, which seems to reduce any forward compatibility avenue rather than increasong it. Also, if we are going to store the mib in some implementation-defined type, there is no reason to make it an enum.

But P2498R1 [2] fails to offer an example of a possible "alternative" registry or to explain how the proposed changes would make supporting alternative registry helpful. Let's go through some options.

WhatWG

Unlike IANA, WhatWG encodings are only identified by name and do not have any associated numerical identifier of any sort. WHATWG focuses on encodings frequently involved in the rendering of web pages and as such is, with exceptions, a subset of IANA. The following illustrate how to map IANA to whatwg encodings:

```
constexpr bool is_whatwg_encoding(const text_encoding & te) {
    using e = text_encoding::id;
    text_encoding::id ids[] {
        e::ASCII,
        e::UTF8, e::IBM866,
        e::ISOLatin2, e::ISOLatin3,
        e::ISOLatin4, e::ISOLatin5, e::ISOLatin6,
        e::ISOLatinGreek, e::ISOLatinHebrew,
        e::ISO88598I, e::ISO88596E,
        e::ISO885913, e::ISO885914,
        e::ISO885915, e::ISO885916,
        e::KOI8R, e::KOI8U,
        e::Macintosh, e::TIS620,
        e::windows1250, e::windows1251,
        e::windows1252, e::windows1253,
        e::windows1254, e::windows1255,
        e::windows1256, e::windows1257, e::windows1258,
        e::GBK, e::GB18030,
```

```
        e::Big5, e::EUCPkdFmtJapanese,
        e::ISO2022JP, e::ShiftJIS,
        e::EUCKR, e::KSC56011987,
        e::ISO2022KR,
        e::UTF16, e::UTF16BE, e::UTF16LE
    };
    if(auto it = std::find(std::begin(ids), std::end(ids), te.mib()); it != std::end(ids) )
        return true;
    if(te == text_encoding("x-mac-cyrillic") || te == text_encoding("x-mac-ukrainian"))
        return true;
    return false;
}
```

Note that, because WhatWG goal is to standardize existing practices on the web, they consider ASCII, latin1, and windows-1252 to be the same encodings, while they are not under IANA (because, well, they are not).

Therefore, let's imagine `whatwg_encoding`, an object modeling the WHATWG specification.

```
whatwg_encoding("ascii") == whatwg_encoding("windows-1252");  // true
text_encoding("ascii")   == text_encoding("windows-1252");  // false
whatwg_encoding("ascii") == text_encoding("windows-1252");  // ???
```

That last expression doesn't make sense; the class have different invariants, and while it is possible to establish a conversion between the two types, it is not possible to establish a direct equivalence. Now, if 'text_encoding' was modeling two or more registries, what comparison rules should it follow?

This questions applies to other registries.

IBM, windows and vendors code pages

IBM and windows code pages do not constitute registries; they are interfaces and documentation specific to a vendor, for the purposes of their respective environment, and are not suitable to the aim of this paper: improving portability.

But the situation is the same as for WHATWG:

- It is possible to map from one set of encoding to another without any extra data member.
- When a mapping exists, a conversion between mappings can be established, but no equality is possible.

Yet that doesn't mean that vendor-specific support can't exist.

```
// Option A
struct text_encoding  {
public:
    constexpr id mib(); const
    constexpr auto __vendor_specific_codepage() const;
};
```

```
// Option B
namespace vendor_specific_namespace{
    struct vendor_text_encoding  {
        public:
        constexpr auto codepage() const;
    };
}
```

ISO 2022

ISO 2022 (INTERNATIONAL REGISTER OF CODED CHARACTER SETS TO BE USED WITH ESCAPE SEQUENCES) is unsuitable for several reasons. For one, it is not updated since 1994, so it fails to reference important encodings such as GB18030. But most importantly, ISO 2022 does not describe a general encoding registry but a set of encodings or parts of encodings for use with ISO 2022. Specifically, what is registered in that registry are escape sequences to switch from/to an encoding. The existence of such escape sequences makes the registered encoding not strictly conforming to the encoding the escape sequence pertains to. For example, the registry describes no less than 4 ”UTF-8” entries, all pertaining to different ways to insert UTF-8 in an ISO 2022 stream.

Also, absent from this database are all the windows and IBM encodings. Some of the encodings used as part of ISO2022 were registered in IANA, and others are no longer in use.

Have you considered making `text_encoding` a template classd with a registery as parameter?

`text_encoding` describes elements of the IANA database:

- An enum all of the encodings, for type safety
- An mib for compatibility with third-party libraries, type safety, and fast comparison.
- A set of aliases.
- A set of rules for the comparison of names.
- A support for unregistered encodings

Other registers may fail to provide some of these capabilities or have different rules. WhatWG doesn’t have a numerical id. Windows does not have aliases, for example.

As such, two registries may have a different interface, maintain different invariants and have different rules for comparison.

There is also no generic way to compare between registries, or to map between two registries if both of them maintain a numerical value are part of their invariant.

These would be different types, with different methods and semantics: not a good scenario for genericity.

On the legitimacy of IANA

IANA is a recognized standards organization which have been involved in the maintenance of critical parts of the Internet since 1988.

In any case, the database itself, like the time zone database which is referenced by the C++ standard, is relied on by many products and is under Creative Commons CC0 1.0. Its current maintainership have no bearing on the proposal at hand or subsequent evolutions.

Yet, P2498R1 [2]'s intent to rename `mib`' to `iana_mib` - a rename I'm not opposed to - seems to directly contradict the concerns about relying on a IANA standard.

In any case, it behooves P2498R1 [2] to determines whether the name "iana" can be used in the API from a legal standpoint.

Still, should WG21 import the entire registry as an annex?

While I do not think this is necessary... if that's something editors want to do, why not?

Should WG21 create its own registry?

That would be an excellent way to waste an enormous of committee time on the basis that everybody else's standard is unfit. The value of an industry-standard is that everybody uses the same, not in everyone making their own, differently broken version of it. Concerns that IANA is unsuitable for non-8-bits encodings are addressed further in this paper.

Should we be adding padding?

P2498R0 [1] proposed:

> When defining the member variables and layout of `text_encoding`, implementations should consider the possibility that future revisions of this standard may reference additional or alternative text encoding registries.

I would oppose adding such wording, which asks for implementation to support unknown unknowns. What does supporting alternative text encoding registries entail? What should an implementation do? Reserve more stack space? 1 byte? A hundred?

Adding paddings to arbitrary standard types is, in my opinion, not the way to tackle ABI concerns. It is not a scalable approach. I would like to observe that the `text_encoding::id` is purposefully using an `int32_t` even if there are no encodings id greater than `3000`. This, along with the 64 bytes for the name, gives ample opportunity for WG21 to alter the design in the future.

In any case, if there are concerns that the current design or the IANA database are not suitable, I would advise not pursuing this paper instead of trying to future-proof it.

Addressing concerns raised by P2491R0 [?]

Wide encodings

P2491R0 [**?**] argues that the wide encoding methods present in P1885R8 [5] should return UTF-16LE/BE instead of `UTF-16`. According to the Unicode standard, both approaches are technically valid; the question is then which option is the most uuser-friendly.

[*Note:* P2491R0 [**?**] makes further claims that the proposed behavior is not compatible with ISO-10646. However, communications with the editor of 10646 explains that the divergence between the Unicode Standard and ISO 10646 are not intentional. Furthermore, P1885 lets implementations define the mapping between their encodings and IANA names, and nothing in the wording suggests that the mapping offers stronger semantics guarantees. — *end note*]

Yet, we note that the concerns raised by P2491R0 pertaining to `iconv` BOM handling- when the ”UTF-16” encoding is used - are real. This was previously discussed by SG-16, which decided that it would not be an issue for iconv to look for a box in this scenario.

P2491R0 [**?**] further argue that UCS-2 Little Endian and UCS-2 Big Endian should be distinguished in the name of the encoding, despite the lack of usage experience.

The argument was also made that, unless P2460R0 [4] was approved (which we fail to do in a timely manner), a conforming implementation would not be able to return `UTF-16` for the wide or narrow encodings.

P2491R0 [**?**] offers, as a solution to these concerns, to add new encoding names (such as ”WIDE.UTF16”, ”WIDE.UTF32”, ”WIDE.UCS2”, and ”WIDE.UCS4”). This is deeply problematic. An encoding is orthogonal to the structure of the underlying storage. A wide string should have the same encodings whether it is stored in an array of bytes or an array of `wchar_t`. We then would have to settle how ”WIDE.UTF16” and ”UTF16” compare. From a teacheability standpoint (a motivation for my paper is the hope that introspection leads to less confusion) this would be a disaster. There would be no good answer to ”What is the difference between what is the difference between WIDE.UTF16 and UTF16?”.

This direction would also requires changes to `iconv`, `icu` and other libraries.

I am strongly opposed to such a direction. The goal of the present paper is to describe the current mess, not to add to it. WG21 should not be in the business of making up text encodings.

It is unclear, which problem, if any would be solved by this approach, except that they would not have to adhere to the definition of ISO, which is already a non problem as again the implementation can map the name to semantic the way it sees fit.

We should note that the proposed change has no usage experience at all.

Instead, **the current revision removes the `wide_literal`, `wide_environment` and `wide_locale` methods**. There are several reasons for that:

- Most, if not all, objections to this paper have stemmed from the wide methods or the systems in which `CHAR_BITS != 8`. The wide methods were added for consistency, yet it

has become increasingly clear that ”consistency” and thoroughness come at too great a cost

- Wide chars are mostly a C++ inventions, and querying the wide chars encodings is less useful. On some platforms, the wide encoding is a known constant (Linux, Windows, Mac) or an encoding not designed to be handled by users directly (FreeBSD, zOS).
- Recent discussions around P2460R0 [4] and the usability of wide characters across platforms (In WG21 and WG14 alike) are not indicative of great interest for wide character handling in general.
- Documented double bytes encodings are less frequent. A raised concern was that IANA didn't support wide encodings well. The reality is that there aren't that many double bytes encodings (encodings composed of 1 double-byte + 1 single-byte encoding - as found on IBM platforms, for example - excluded). IANA simply reflects the reality.

The result is that wide methods are contentious and poorly motivated. And so, they are removed from this paper. Should someone find a good use case, more work can be done in a future standard. It is my hope that this addresses P2491R0 [**?**] to the satisfaction of its author.

I believe this also addresses concerns raised by the electronic polling of P1885. The comments can be found in P2455R0 [6].

> The concerns expressed in [P2491R0] are very real. When wchar_t is in use and is of the right size, it is plausible to use names like UTF-16 to refer to an encoding form (trivially encoded as a sequence of wchar_t values) rather than an encoding scheme, but that would require weakening the focus on IANA in P1885 and would have only a weak connection to the intended utilities like iconv that plainly interpret such names as referring to an encoding scheme. Implementations can specify that functions like iconv work with wchar_t arrays without (and are not assisted in doing so by) any statement about object representations in the standard.
> It is also true that inventiveness in a domain already so chaotic is plainly undesirable. While it might merely reaffirm [P1885R8]'s practical direction with a different interpretation, it seems that we need to reconsider the choices made in the extremely troublesome area of non-octet-based encoding schemes.
> — Weakly Against

> This feature is very useful (see for example next poll) and I believe should be included in standard. However, after reading [P2491R0], I agree that this concern need resolution. This may be as simple, as limiting returning IANA schemes to sizeof(char_size) == 1, i.e. leaving wide_literal implementation-defined now.
> — Weakly Against

CHAR_BIT

SG-16 had unresolved questions about how UTF-16 should be encoded on a platform where CHAR_BIT is 16 bits or more. Tom Hermann did some research, and there does not seem to exist such a scenario, to the best of our knowledge, based on public information. The author has no use cases for using P1885 on DSP system. We would also like to point out that P1885 is designed to be freestanding in the future, but it is currently not, and there is seemingly no non-8 bits non-freestanding system. As such, the wording mandates `CHAR_BIT == 8`, so that the wording doesn't have to account for hypotheticals. If the need to relax this limitation emerges, it should be driven by people with expert knowledge of such an environment.

Looking at the object representation breaks an abstraction barrier

As noted by P2491R0 [**?**] an encoding is a sequence of code units. When reinterpreted as a sequence of `char`, such sequence of code units should not be separated by padding; code units should not be 0 extended, etc. (on systems where `CHAR_BIT == 8`). An encoding is a bit pattern. I will use whatever terminology core deems appropriate. Note that since `CHAR_BIT == 8` is mandated by this version of this paper, and because we removed wide methods, this point is moot. **I have removed that wording formulation in R9 of this paper.**

The many text encodings of a C++ environment

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 environments 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 environment 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, three sets of encodings of primary interest:

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

[*Note:* Because they have different code units sizes, narrow 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 environment 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 because it is not generally possible to differentiate runtime strings from compile-time literals at runtime.

This model does, however, present some shortcomings:

First, in practice, C++ software is 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++ derive 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) 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 environment-assumed encoding and the encoding of string literals.

Exposing the different encodings would let developers verify that that the 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 exists 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 encodings 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

We use the term `literal` to match the core wording. ”Environment encoding” is a descriptive term 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. It is also purposefully distinct from the term ”execution encoding” which is tied to locales.

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 besides a desire not to deviate from the existing standards and practices. — *end note*]

The enumerators `unknown` and `other`and their corresponding values, are specified in [ianacharset-mib]:

- `other` designate an encoding not registered in the IANA Database, such that two encodings with the `other` mib are identical if their names 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 the fact that encoding can have multiple inconsistent names across implementations.

For forward compatibility with the RFCs, this enumeration’s underlying type is `int_least32_t`.

The RFC definition of INTEGER can be found in RFC2578:

> The Integer32 type represents integer-valued information between -2^{31} and $2^{31} - 1$ inclusive (-2147483648 to 2147483647 decimal). This type is indistinguishable from the INTEGER type. Both the INTEGER and Integer32 types may be sub-typed to be more constrained than the Integer32 type.
> The INTEGER type (but not the Integer32 type) may also be used to represent integer-valued information as named-number enumerations. In this case, only those named numbers so enumerated may be present as a value.

> Note that although it is recommended that enumerated values start at 1 and be numbered contiguously, any valid value for Integer32 is allowed for an enumerated value and, further, enumerated values needn't be contiguously assigned.

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.

name and aliases work as follow:

- When constructed from the `string_view` constructor, `name()` returns the name passed to the constructor
- When constructed from a mib, `name()` returns an implementation defined name that exists in the list of aliases
- When constructed per the implementation, `name()` returns an implementation defined-value

In addition, `aliases.front()` is defined to return the primary name, as defined by IANA

Unique identification of encodings

The IANA database intends that the name refers to a specific set of characters. However, for historical reasons, there exist some names (like Shift-JIS) which describes several slightly different encoding. The intent of this proposal is that the names refer to the character encodings as described by IANA. Further differentiation can be made in the application through out-of-band information such as the provenance of the text to which the encoding is associated. RFC2978 mandates that all names and aliases are unique.

Implementation flexibility

This proposal aims to be implementable on all platforms. It supports encodings not registered with IANA, does not impose that a freestanding implementation is aware of all registered encodings, and it lets implementers provide their aliases for IANA-registered encoding. Because the process for registering encodings is documented [rfc2978] implementations can (but are not required to) provide registered encodings not defined in [ianacharset-mib] - in the case that document is updated out of sync with the standard. However, [ianacharset-mib] is not frequently updated. It was updated once in 2021 and previously in 2011. As the world converges to UTF-8, new encodings are less likely to be registered. Until 2004 this document was maintained in [rfc3808].

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, which is why we return a `const char*`. This is requested by users and consistent with `source_location`, `stacktrace`, … We would have considered a null-terminated `string_view` as proposed in P1402R0 [8] if such a thing was available.

When constructed from the unknown mib, `name` returns a `nullptr` rather than an empty string.

Freestanding

For this class to be compatible with freestanding environments, care has been taken to avoid allocation and exceptions. As such, we put an upper bound on the length of the name of encodings passed to `text_encoding` constructor of 63+1 characters. Per rfc2978, the names must not exceed 40 characters. There is, however, a name of 45 characters in the database. 64 has been arbitrarily chosen, being the smallest power of 2 number that would fit all the names with some extra space for future-proofing (there are ABI concerns here).

However, no wording for freestanding is provided as there are currently missing pieces (notably string_view). We propose that making this facility freestanding can be bundled with the wider work by Ben Craig.

Name comparison

Names and aliases are compared ignoring case and non-alphanumeric characters, in a way that follows Unicode recommendations

This leads to a couple of ambiguities (”iso-ir-9-1” and ”iso-ir-9-2” match ”iso-ir-91” and ”iso-ir-92”, respectively). The two problematics encodings have been excluded from our proposal entirely. They were designed in 1975 for use in newspapers in Norway and are no longer in use. Supporting them would either require a perfect match, even though we know from experience that users will find 20 creative ways to spell UTF-8, or to perform in sequence a perfect match and a loose match; we do not this is a reasonable cost to pay for algorithms that fell into disuse long ago:

Reference: iso-ir-9-1 iso-ir-9-2

Note that these are different from ISO646-NO2 which is the long obsoleted Norwegian ancestor to 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 a POSIX platform it corresponds to value of `nl_langinfo_l` when the environment ("") locale is set - before the program's locale is set to `C`.

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

Part of this proposal is available on Compiler explorer.

- Literal encodings are only supported on recent version of clang and GCC
- no `std::locale` integration
- Compiler Explorer limitations prevent the implenentation to be immune to calls to `setenv`

Handling mutation of `LC_CTYPE` at runtime

On POSIX, the environment encoding is derived from the default locale `""`, which itself is derived from the value of the environment variables `LC_CTYPE`, `LC_ALL` and `LANG` (in that order). Which means a call to `setenv` might affect the value returned by `text_encoding::environmemt()`. While this would be conforming, it would be more helpful if the implementation was impervious to such modification of the environment. We can achieve that by:

- On Linux and freebsd, parsing `/proc/self/environ` to use these values instead off the one returned by `getenv`
- On OSX, parsing the memory where the environment is stored, as returned by `sysctl({ CTL_KERN, KERN_PROCARGS, pid})`.
- On Windows, `GetACP` is not affected by this issue.

On implementations where the implementers also control the libc, better strategies may be available.

It is useful to get the locale of the environment as this represents the encoding that command-line arguments, environment variables and non-redirected standard streams are likely to use.

The encoding of the current locale may be different - because the global locale is initially set to "C", and users can set an arbitrary global locale, and therefore, that global locale-associated encoding may be different from the environment encoding.

The global locale associated encoding can be queried with `locale().encoding()`.

It would be hard for users to implement this function as it is not portably implementable. Of course, on some platforms, the recommended behavior may not be implementable, in which case implementations could return the encoding using the values of LC_ at the point of call.

Storing aliases

We found that aliases can be efficiently stored and looked up in a sorted list of alias/mib pairs. Making a `common_range` of `aliases_view` would force an implementation to find the end of the

list of aliases for a particular encoding, which is slightly efficient than what is possible, so this is not proposed. Mostly, we found little use for it to be a `common_range`.

Compatibility with third-party systems

Qt

```
// Get a QTextCodec able to convert the environment encoding to QString
auto codec = QTextCodec::codecForMib(std::text_encoding::environment().mib());
```

ICU

```
// Get a UConverter object able to convert to and from the environment encoding to
// ICU's internal encoding.
UErrorCode err;
UConverter* converter = ucnv_open(std::text_encoding::environment().name(), &err);

// Check whether a UConverter converts to the environment encoding
bool compatibleWithenvironmentEncoding(UConverter* converter)
{
    UErrorCode err;
    const char* name == ucnv_getName(converter, &err);
    assert(U_SUCCESS(err));
    return std::text_encoding(name) == std::text_encoding::environment();
}
```

ICONV

```
// Convert from UTF-8 to the environment encoding, transliterating if necessary
iconv_t converter
    = iconv_open(std::format("{}//TRANSLIT", std::text_encoding::literal().name()).c_str(), "utf-8");
```

FAQ

Why rely on the IANA registry ?

The IANA registry has been picked for several reasons:

- It can be referenced through an RFC in the standard
- It has wide vendor buy-in
- It is used as a primary source for many tools, including ICU and iconv, and many programming languages and libraries.
- It has an extensive number of entries which makes it uniquely suitable for the wide portability requirements of C++. Notably, it supports IBM codepages.
- It provides stable enum values designed for efficient and portable comparison in programming languages

- There is a well-specified support for unregistered encodings
- There is a well-specified process to register new encodings

We also considered the WHATWG Encoding specification. But this specification is designed specifically for the web and has no provision for EBCDIC encodings, provides no numerical values, etc.

Annex A provides a comparative list of IANA and WHATWG lists.

Extensive research did not found any other registry worth considering. It would be possible to maintain our own list in the standard, but this would put an undue burden on the committee and risks reducing portability with existing tools, libraries, and other languages.

Why not return a `text_encoding::id` rather than a `text_encoding` object?

Some implementations may need to return a non-register encoding, in which case they would return `mib::other` and a custom name.

`text_encoding::environment()` and `text_encoding::environment_mib()` (not proposed) would generate the same code in an optimized build.

But handling names is expensive?

To ensure that the proposal is implementable in a constrained environment, `text_encoding` has a limit of 63 characters per encoding name which is sufficient to support all encodings we are aware of (registered or not)

It seems like names and mib are separate concerns?

Not all encodings are registered (even if most are). It is therefore not possible to identify all encoding uniquely by mib. Encodings may have many names, but some platforms will have a preferred name.

The combination of a name + a mib covers 100% of use cases. Aliases further help with integration with third-party libraries or to develop tools which need encoding names.

Why can't there be vendor provided MIBs?

This would be meaningless in portable code. `mib` is only useful as a mechanism to identify encodings **portably** and to increase compatibility across third-party libraries.

It does not prevent the support of unregistered encodings:

```
text_encoding wtf8("WTF-8");
assert(wtf8.name() == "WTF-8"sv);
assert(wtf8.mib() == text_encoding::id::other);
```

Why can't there be a `text_encoding(name, mib)` constructor?

Same reason, if users are allowed to construct text_encoding from registered names or names otherwise unknown from the implementation with an arbitrary mib, it becomes impossible to maintain the invariant of the class (the relation between mib and name), which would make the interface much harder to use without providing any functionality.

I just want to check that my platform is utf-8 without paying for all these other encodings?

we added `environment_is` to that end.

```
int main() {
    assert(text_encoding::environment_is<text_encoding::id::UTF8>
              && "Non UTF8 encoding detected, go away");
}
```

This can be implemented in a way that only stores in the program the necessary information for that particular encoding (unless `aliases` is called at runtime).

On Windows and OSX, only calling `encoding::aliases` would pull any data in the program, even if calling `environment`.

What is the cost of calling `aliases`?

My crude implementation pulls in 30Ki of data when calling aliases or the name constructor, or `environment()` (on POSIX).

Why do `name()` and `aliases()` return `const char*` rather than `string_view`?

One of the design goals is to be compatible with widely deployed libraries such as ICU and iconv, which are, on most platforms, the defacto standards for text transformations, classification, and transcoding. These are C APIs that expect null-terminated strings. Returning a null-terminated string_view of which end() is dereferenced would be UB. Returning a `string` and hopping that SBO kicks in would add complexity for little reason and would preclude the name function from being provided in freestanding implementations. LEWG previously elected to use const char* in source_location, stack trace, etc

NATS-DANO and NATS-DANO-ADD

Both these ISO 646 characters encodings have aliases that conflict with other aliases under `COMP_NAME` and are excluded, as they fell into disuse a long time ago. This was necessary to allow compatibility with the Unicode-recommended name matching.

Wording strategy

The followings were important points of discussions for the elaboration of the wording:

- `text_encoding` is described as holding an encoding *scheme*,
- An encoding scheme is a sequence of octets (independently of`CHAR_BIT`), the general idea being that reinterpreting an encoded string as a `char*`, along with the encoding name should be decodable by `iconv`, assuming the encoding name is known of `iconv`.
- The determination of what the encoding schemes are is implementation-defined, but we provide recommended practices.
- Naming of non-registered encoding is implementation-defined with no recommended practice as these are not portable and should remain at the discretion of implementations.
- Further wording concerns with `CHAR_BIT != 8`. and specifically `CHAR_BIT > 16` led the authors to mandate `CHAR_BIT == 8`. The intent is for the static functions to be delected on such system.

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 described in [P1868].

Proposed wording

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

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

```
#define __cpp_lib_text_encoding 2030XX (**placeholder**) // also in text_encoding
```

[Editor's note: Add a new header `<text_encoding>`] .

> [text.encoding] describes an interface for accessing the IANA Character Sets registry.

```
namespace std {

struct text_encoding {

    inline constexpr size_t max_name_length = 63;

    enum class id : int_least32_t {
        other = 1,
        unknown = 2,
```

```
ASCII = 3,
ISOLatin1 = 4,
ISOLatin2 = 5,
ISOLatin3 = 6,
ISOLatin4 = 7,
ISOLatinCyrillic = 8,
ISOLatinArabic = 9,
ISOLatinGreek = 10,
ISOLatinHebrew = 11,
ISOLatin5 = 12,
ISOLatin6 = 13,
ISOTextComm = 14,
HalfWidthKatakana = 15,
JISEncoding = 16,
ShiftJIS = 17,
EUCPkdFmtJapanese = 18,
EUCFixWidJapanese = 19,
ISO4UnitedKingdom = 20,
ISO11SwedishForNames = 21,
ISO15Italian = 22,
ISO17Spanish = 23,
ISO21German = 24,
ISO60DanishNorwegian = 25,
ISO69French = 26,
ISO10646UTF1 = 27,
ISO646basic1983 = 28,
INVARIANT = 29,
ISO2IntlRefVersion = 30,
NATSSEFI = 31,
NATSSEFIADD = 32,
ISO10Swedish = 35,
KSC56011987 = 36,
ISO2022KR = 37,
EUCKR = 38,
ISO2022JP = 39,
ISO2022JP2 = 40,
ISO13JISC6220jp = 41,
ISO14JISC6220ro = 42,
ISO16Portuguese = 43,
ISO18Greek7Old = 44,
ISO19LatinGreek = 45,
ISO25French = 46,
ISO27LatinGreek1 = 47,
ISO5427Cyrillic = 48,
ISO42JISC62261978 = 49,
ISO47BSViewdata = 50,
ISO49INIS = 51,
ISO50INIS8 = 52,
ISO51INISCyrillic = 53,
ISO54271981 = 54,
ISO5428Greek = 55,
```

```
ISO57GB1988 = 56,
ISO58GB231280 = 57,
ISO61Norwegian2 = 58,
ISO70VideotexSupp1 = 59,
ISO84Portuguese2 = 60,
ISO85Spanish2 = 61,
ISO86Hungarian = 62,
ISO87JISX0208 = 63,
ISO88Greek7 = 64,
ISO89ASMO449 = 65,
ISO90 = 66,
ISO91JISC62291984a = 67,
ISO92JISC62991984b = 68,
ISO93JIS62291984badd = 69,
ISO94JIS62291984hand = 70,
ISO95JIS62291984handadd = 71,
ISO96JISC62291984kana = 72,
ISO2033 = 73,
ISO99NAPLPS = 74,
ISO102T617bit = 75,
ISO103T618bit = 76,
ISO111ECMACyrillic = 77,
ISO121Canadian1 = 78,
ISO122Canadian2 = 79,
ISO123CSAZ24341985gr = 80,
ISO88596E = 81,
ISO88596I = 82,
ISO128T101G2 = 83,
ISO88598E = 84,
ISO88598I = 85,
ISO139CSN369103 = 86,
ISO141JUSIB1002 = 87,
ISO143IECP271 = 88,
ISO146Serbian = 89,
ISO147Macedonian = 90,
ISO150 = 91,
ISO151Cuba = 92,
ISO6937Add = 93,
ISO153GOST1976874 = 94,
ISO8859Supp = 95,
ISO10367Box = 96,
ISO158Lap = 97,
ISO159JISX02121990 = 98,
ISO646Danish = 99,
USDK = 100,
DKUS = 101,
KSC5636 = 102,
Unicode11UTF7 = 103,
ISO2022CN = 104,
ISO2022CNEXT = 105,
UTF8 = 106,
```

```
ISO885913 = 109,
ISO885914 = 110,
ISO885915 = 111,
ISO885916 = 112,
GBK = 113,
GB18030 = 114,
OSDEBCDICDF0415 = 115,
OSDEBCDICDF03IRV = 116,
OSDEBCDICDF041 = 117,
ISO115481 = 118,
KZ1048 = 119,
UCS2 = 1000,
UCS4 = 1001,
UnicodeASCII = 1002,
UnicodeLatin1 = 1003,
UnicodeJapanese = 1004,
UnicodeIBM1261 = 1005,
UnicodeIBM1268 = 1006,
UnicodeIBM1276 = 1007,
UnicodeIBM1264 = 1008,
UnicodeIBM1265 = 1009,
Unicode11 = 1010,
SCSU = 1011,
UTF7 = 1012,
UTF16BE = 1013,
UTF16LE = 1014,
UTF16 = 1015,
CESU8 = 1016,
UTF32 = 1017,
UTF32BE = 1018,
UTF32LE = 1019,
BOCU1 = 1020,
UTF7IMAP = 1021,
Windows30Latin1 = 2000,
Windows31Latin1 = 2001,
Windows31Latin2 = 2002,
Windows31Latin5 = 2003,
HPRoman8 = 2004,
AdobeStandardEncoding = 2005,
VenturaUS = 2006,
VenturaInternational = 2007,
DECMCS = 2008,
PC850Multilingual = 2009,
PC8DanishNorwegian = 2012,
PC862LatinHebrew = 2013,
PC8Turkish = 2014,
IBMSymbols = 2015,
IBMThai = 2016,
HPLegal = 2017,
HPPiFont = 2018,
HPMath8 = 2019,
```

```
HPPSMath = 2020,
HPDesktop = 2021,
VenturaMath = 2022,
MicrosoftPublishing = 2023,
Windows31J = 2024,
GB2312 = 2025,
Big5 = 2026,
Macintosh = 2027,
IBM037 = 2028,
IBM038 = 2029,
IBM273 = 2030,
IBM274 = 2031,
IBM275 = 2032,
IBM277 = 2033,
IBM278 = 2034,
IBM280 = 2035,
IBM281 = 2036,
IBM284 = 2037,
IBM285 = 2038,
IBM290 = 2039,
IBM297 = 2040,
IBM420 = 2041,
IBM423 = 2042,
IBM424 = 2043,
PC8CodePage437 = 2011,
IBM500 = 2044,
IBM851 = 2045,
PCp852 = 2010,
IBM855 = 2046,
IBM857 = 2047,
IBM860 = 2048,
IBM861 = 2049,
IBM863 = 2050,
IBM864 = 2051,
IBM865 = 2052,
IBM868 = 2053,
IBM869 = 2054,
IBM870 = 2055,
IBM871 = 2056,
IBM880 = 2057,
IBM891 = 2058,
IBM903 = 2059,
IBM904 = 2060,
IBM905 = 2061,
IBM918 = 2062,
IBM1026 = 2063,
IBMEBCDICATDE = 2064,
EBCDICATDEA = 2065,
EBCDICCAFR = 2066,
EBCDICDKNO = 2067,
EBCDICDKNOA = 2068,
```

```
EBCDICFISE = 2069,
EBCDICFISEA = 2070,
EBCDICFR = 2071,
EBCDICIT = 2072,
EBCDICPT = 2073,
EBCDICES = 2074,
EBCDICESA = 2075,
EBCDICESS = 2076,
EBCDICUK = 2077,
EBCDICUS = 2078,
Unknown8BiT = 2079,
Mnemonic = 2080,
Mnem = 2081,
VISCII = 2082,
VIQR = 2083,
KOI8R = 2084,
HZGB2312 = 2085,
IBM866 = 2086,
PC775Baltic = 2087,
KOI8U = 2088,
IBM00858 = 2089,
IBM00924 = 2090,
IBM01140 = 2091,
IBM01141 = 2092,
IBM01142 = 2093,
IBM01143 = 2094,
IBM01144 = 2095,
IBM01145 = 2096,
IBM01146 = 2097,
IBM01147 = 2098,
IBM01148 = 2099,
IBM01149 = 2100,
Big5HKSCS = 2101,
IBM1047 = 2102,
PTCP154 = 2103,
Amiga1251 = 2104,
KOI7switched = 2105,
BRF = 2106,
TSCII = 2107,
CP51932 = 2108,
windows874 = 2109,
windows1250 = 2250,
windows1251 = 2251,
windows1252 = 2252,
windows1253 = 2253,
windows1254 = 2254,
windows1255 = 2255,
windows1256 = 2256,
windows1257 = 2257,
windows1258 = 2258,
TIS620 = 2259,
```

```
        CP50220 = 2260
    };

    using enum id;

    constexpr text_encoding() = default;
    constexpr explicit text_encoding(string_view name) noexcept;
    constexpr text_encoding(id mib) noexcept;

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

    struct aliases_view;
    constexpr aliases_view aliases() const noexcept;

    constexpr friend bool operator==(const text_encoding& encoding, const text_encoding & other) noexce
    constexpr friend bool operator==(const text_encoding& encoding, id mib) noexcept;

    static consteval text_encoding literal() noexcept;

    static text_encoding environment();


    template<id id_>
    static bool text_encoding::environment_is();

    private:
        id mib_ = id::unknown; // exposition only
        char name_[max_name_length+1] = {0}; // exposition only
    };


// hash support
template<class T> struct hash;
template<> struct hash<text_encoding>;

}
```

A *registered character encoding* is a character encoding scheme in the IANA Character Sets registry. [*Note:* The IANA Character Sets registry refers to character sets rather than character encodings. — *end note*]

The set of known registered character encoding contains every registered character encoding specified in the IANA Character Sets registry except for the following:

- NATS-DANO (33)
- NATS-DANO-ADD (34)

Each known registered character encoding is identified by an enumerator in `text_encoding::id`, has a unique *primary name* and has a set of zero or more *alias*es. The primary name of a registered character encoding is the name of that encoding specified in the IANA Character

Sets registry.

[Editor's note: The term primary name appears in RFC2978]

The set of aliases of a registered character encoding is an implementation-defined superset of the aliases specified in the IANA Character Sets registry. No two registered character encodings share any identical alias when compared by `COMP_NAME`.

[*Note:* The `text_encoding::id` enumeration contains an enumerator for each known registered character encoding. For each encoding, the corresponding enumerator is derived from the alias beginning with "cs", as follows

- the "cs" prefix is removed from each name
- `csUnicode` is mapped to `text_encoding::id::UCS2`
- `csIBBM904` is mapped to `text_encoding::id::IBM904`

— *end note*]

How a `text_encoding` object is determined to be representative of a character encoding scheme implemented in the translation or execution environment is implementation-defined.

An object `e` of type `text_encoding` maintains the following invariants:

- `e.name() == nullptr` is `true` if and only if `e.mib() == text_encoding::id::unknown` is `true`.
- `e.mib() == text_encoding(e.name()).mib()` is `true` if `e.mib() == text_encoding::id::other` is `true`.

Recommended practice:

- Implementations should not consider registered encodings to be interchangeable [Example:Shift_JIS and Windows-31J denote different encodings].
- Implementations should not refer to a registered encoding to describe another similar yet different non-registered encoding unless there is a precedent on that implementation (Example: Big5).

Let `bool COMP_NAME (string_view a, string_view b)` be a function that returns `true` if the two strings a and b encoded in the ordinary literal encoding are equal ignoring, from left-to-right,

- all elements which are not digits or letters [character.seq.general],
- character case, and
- any sequence of one or more '0' character not immediately preceded by a sequence consisting of a digit in the range [1-9] optionally followed by one or more elements which are not digits or letters.

[*Note:* This comparison is identical to the "Charset Alias Matching" algorithm described in the Unicode Technical Standard 22. — *end note*]

[*Example:*

```
    assert(COMP_NAME("UTF-8", "utf8") == true);
    assert(COMP_NAME("u.t.f-008", "utf8") == true);
    assert(COMP_NAME("ut8", "utf8") == false);
    assert(COMP_NAME("utf-80", "utf8") == false);
```

— end example]

`constexpr explicit text_encoding(string_view name) noexcept;`

Preconditions:

- `name` represents a string in the ordinary literal encoding,
- all elements in `name` are in the basic source character set,
- `name.size() <= max_name_length` is `true`, and
- `name.contains('\0')` is `false`.

Postconditions:

- If there exists a primary name or alias `a` of a known registered character encoding such that *`COMP_NAME`* `(a, name)` is `true`, `mib_` has the value of the enumerator of `id` associated with that registered character encoding. Otherwise, `mib_ == id::other` is `true`.
- `name.compare(name_) == 0` is `true`

`constexpr text_encoding(id mib) noexcept;`

Preconditions: `mib` has the value of one of the enumerators of `id`.

Postcondition:

- `mib_ == mib` is true.
- If `(mib_ == id::unknown || mib_ == id::other)` is true, `strlen(name_) == 0` is `true`. Otherwise, `ranges::find(aliases, string_view(name_)) != aliases().end()`.

`constexpr id mib() const noexcept;`

Returns: `mib_`.

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

Returns: `name_` if `(name_[0] != '\0')`, `nullptr` otherwise;

Remarks: If `name() == nullptr` is false, `name()` is an NTMBs and accessing elements of `name_` outside of the range `[name(), strlen(name())+1]` is undefined behavior.

`constexpr aliases_view aliases() const noexcept;`

Let `r` denote an instance of `aliases_view`.

If `*this` represents a known registered character encoding then:

- `r.front()` is the primary name of the registered character encoding,
- `r` contains the aliases of the registered character encoding,
- `r` does not contain duplicate values when compared with `strcmp`.

Otherwise, `r` is an empty range.

All elements in `r` are non-null, non-empty NTBS encoded in the literal character encoding and comprised only of characters from the basic character set.

Returns: `r`.

[*Note:* The order of elements in `r` is unspecified. — *end note*]

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

Mandates: `CHAR_BIT == 8` is `true`.

Returns: a `text_encoding` object representing the encoding scheme associated with the ordinary string literals [lex.charset].

```
static text_encoding environment();
```

Mandates: `CHAR_BIT == 8` is `true`.

Returns: A `text_encoding` object representing the implementation-defined character encoding scheme of the environment. On a POSIX implementation, this is the encoding scheme associated with the POSIX locale denoted by the empty string `""`.

[*Note:* This function is not affected by calls to `setlocale`. It is unspecified whether this function is affected by changes to environment variables during the lifetime of the program. — *end note*]

Recommended practice: Implementations should return a value that is not affected by calls to the POSIX function `setenv` and other functions which can modify the environment [support.runtime].

```
template<id id_>
static bool text_encoding::environment_is();
```

Returns: `environment() == id_`

🔲 class text_encoding::aliases_view [text.encoding.aliases]

```
struct text_encoding::aliases_view : ranges::view_interface<text_encoding::aliases_view> {
    using iterator = implementation-defined;
    using sentinel = implementation-defined;
    constexpr iterator begin() const;
    constexpr sentinel end() const;
};
```

`text_encoding::aliases_view` models `copyable`, `ranges::view`, `ranges::random_access_range`, and `ranges::borrowed_range`. Both `ranges::range_value_t<text_encoding::aliases_view>` and `ranges::range_reference_t<text_encoding::aliases_view>` model `same_as<const char*>`.

[Editor's note: Tomasz suggested that the definitions of begin/end are not necessary as we already say that `text_encoding::aliases_view` models `ranges::view`]

🔲 Comparison functions [text.encoding.comp]

```
constexpr bool operator==(const text_encoding & a, const text_encoding & b) noexcept;
```

Returns:

If `a.mib_ == id::other && b.mib_ == id::other` is true, then *`COMP_NAME`* `(a.name_, b.name_)`.

Otherwise, `a.mib_ == b.mib_`.

```
constexpr bool operator==(const text_encoding & encoding, id mib) noexcept;
```

Returns: `encoding.mib_ == mib`.

Remarks: This operator induces an equivalence relation on its arguments if and only if `i != id::other` is `true`.

🔲 Hash specialization [text.encoding.hash]

```
template<class T> struct hash;
template<> struct hash<text_encoding>;
```

The specialization is enabled ([unord.hash]).

🔲 Locale [locale]

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

    // locale operations
    string name() const;

    text_encoding encoding() const;

  };
}
```

In `[locale.members]`:

```
string name() const;
```

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

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

Mandates: `CHAR_BIT == 8` is `true`.

Returns: a `text_encoding` object representing the implementation-defined encoding scheme associated with the locale `*this`.

Bibliography

— ISO 4217:2015, *Codes for the representation of currencies*

— ISO/IEC 10967-1:2012, *Information technology — Language independent arithmetic — Part 1: Integer and floating point arithmetic*

— ISO/IEC/IEEE 60559:2011, *Information technology — Microprocessor Systems — Floating-Point arithmetic*

— The Unicode Consortium. Unicode Standard Annex, UAX #29, *Unicode Text Segmentation* [online]. Edited by Mark Davis. Revision 35; issued for Unicode 12.0.0. 2019-02-15 [viewed 2020-02-23]. Available from: http://www.unicode.org/reports/tr29/tr29-35.html

— IANA Character Sets Database.
Available from: https://www.iana.org/assignments/character-sets/, 2021-04-01

— *Unicode Character Mapping Markup Language* [online]. Edited by Mark Davis and Markus Scherer. Revision 5.0.1; 2017-05-31 Available from: http://www.unicode.org/reports/tr22/tr22-8.html

— IANA Time Zone Database. Available from: https://www.iana.org/time-zones

— Bjarne Stroustrup, *The C++ Programming Language, second edition*, Chapter R. Addison-Wesley Publishing Company, ISBN 0-201-53992-6, copyright ©1991 AT&T

— Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Appendix A. Prentice-Hall, 1978, ISBN 0-13-110163-3, copyright ©1978 AT&T

— P.J. Plauger, *The Draft Standard C++ Library*. Prentice-Hall, ISBN 0-13-117003-1, copyright ©1995 P.J. Plauger)

The arithmetic specification described in ISO/IEC 10967-1:2012 is called *LIA-1* in this document.

Acknowledgments

Many thanks to Victor Zverovich, Thiago Macieira, Jens Maurer, Tom Honermann, Tomasz Kamiński, Hubert Tong, and others for reviewing this work and providing valuable feedback.

Annex: Registered encodings

IANA	WHATWG
ANSI_X3.110-1983	
ASMO_449	
Adobe-Standard-Encoding	
Adobe-Symbol-Encoding	
Amiga-1251	
BOCU-1	
BRF	
BS_4730	
BS_viewdata	
Big5	Big5
Big5-HKSCS	
CESU-8	
CP50220	
CP51932	
CSA_Z243.4-1985-1	
CSA_Z243.4-1985-2	
CSA_Z243.4-1985-gr	
CSN_369103	
DEC-MCS	
DIN_66003	
DS_2089	
EBCDIC-AT-DE	
EBCDIC-AT-DE-A	
EBCDIC-CA-FR	
EBCDIC-DK-NO	
EBCDIC-DK-NO-A	
EBCDIC-ES	
EBCDIC-ES-A	
EBCDIC-ES-S	
EBCDIC-FI-SE	
EBCDIC-FI-SE-A	
EBCDIC-FR	
EBCDIC-IT	
EBCDIC-PT	
EBCDIC-UK	
EBCDIC-US	
ECMA-cyrillic	
ES	
ES2	
EUC-JP	EUC-JP
EUC-KR	EUC-KR

Extended_UNIX_Code_Fixed_Width_-for_Japanese	
GB18030	gb18030
GB2312	
GBK	GBK
GB_1988-80	
GB_2312-80	
GOST_19768-74	
HP-DeskTop	
HP-Legal	
HP-Math8	
HP-Pi-font	
HZ-GB-2312	
IBM-Symbols	
IBM-Thai	
IBM00858	
IBM00924	
IBM01140	
IBM01141	
IBM01142	
IBM01143	
IBM01144	
IBM01145	
IBM01146	
IBM01147	
IBM01148	
IBM01149	
IBM037	
IBM038	
IBM1026	
IBM1047	
IBM273	
IBM274	
IBM275	
IBM277	
IBM278	
IBM280	
IBM281	
IBM284	
IBM285	
IBM290	
IBM297	
IBM420	

IBM423	
IBM424	
IBM437	
IBM500	
IBM775	
IBM850	
IBM851	
IBM852	
IBM855	
IBM857	
IBM860	
IBM861	
IBM862	
IBM863	
IBM864	
IBM865	
IBM866	IBM866
IBM868	
IBM869	
IBM870	
IBM871	
IBM880	
IBM891	
IBM903	
IBM904	
IBM905	
IBM918	
IEC_P27-1	
INIS	
INIS-8	
INIS-cyrillic	
INVARIANT	
ISO-10646-J-1	
ISO-10646-UCS-2	
ISO-10646-UCS-4	
ISO-10646-UCS-Basic	
ISO-10646-UTF-1	
ISO-10646-Unicode-Latin1	
ISO-11548-1	
ISO-2022-CN	
ISO-2022-CN-EXT	
ISO-2022-JP	ISO-2022-JP
ISO-2022-JP-2	

ISO-2022-KR	
ISO-8859-1	
ISO-8859-1-Windows-3.0-Latin-1	
ISO-8859-1-Windows-3.1-Latin-1	
ISO-8859-10	ISO-8859-10
ISO-8859-13	ISO-8859-13
ISO-8859-14	ISO-8859-14
ISO-8859-15	ISO-8859-15
ISO-8859-16	ISO-8859-16
ISO-8859-2	ISO-8859-2
ISO-8859-2-Windows-Latin-2	
ISO-8859-3	ISO-8859-3
ISO-8859-4	ISO-8859-4
ISO-8859-5	ISO-8859-5
ISO-8859-6	ISO-8859-6
ISO-8859-6-E	
ISO-8859-6-I	
ISO-8859-7	ISO-8859-7
ISO-8859-8	ISO-8859-8
ISO-8859-8-E	
ISO-8859-8-I	ISO-8859-8-I
ISO-8859-9	
ISO-8859-9-Windows-Latin-5	
ISO-Unicode-IBM-1261	
ISO-Unicode-IBM-1264	
ISO-Unicode-IBM-1265	
ISO-Unicode-IBM-1268	
ISO-Unicode-IBM-1276	
ISO_10367-box	
ISO_2033-1983	
ISO_5427	
ISO_5427:1981	
ISO_5428:1980	
ISO_646.basic:1983	
ISO_646.irv:1983	
ISO_6937-2-25	
ISO_6937-2-add	
ISO_8859-supp	
IT	
JIS_C6220-1969-jp	
JIS_C6220-1969-ro	
JIS_C6226-1978	
JIS_C6226-1983	

JIS_C6229-1984-a	
JIS_C6229-1984-b	
JIS_C6229-1984-b-add	
JIS_C6229-1984-hand	
JIS_C6229-1984-hand-add	
JIS_C6229-1984-kana	
JIS_Encoding	
JIS_X0201	
JIS_X0212-1990	
JUS_I.B1.002	
JUS_I.B1.003-mac	
JUS_I.B1.003-serb	
KOI7-switched	
KOI8-R	KOI8-R
KOI8-U	KOI8-U
KSC5636	
KS_C_5601-1987	
KZ-1048	
Latin-greek-1	
MNEM	
MNEMONIC	
MSZ_7795.3	
Microsoft-Publishing	
NATS-DANO	
NATS-DANO-ADD	
NATS-SEFI	
NATS-SEFI-ADD	
NC_NC00-10:81	
NF_Z_62-010	
NF_Z_62-010_(1973)	
NS_4551-1	
NS_4551-2	
OSD_EBCDIC_DF03_IRV	
OSD_EBCDIC_DF04_1	
OSD_EBCDIC_DF04_15	
PC8-Danish-Norwegian	
PC8-Turkish	
PT	
PT2	
PTCP154	
SCSU	
SEN_850200_B	
SEN_850200_C	

Shift_JIS	Shift_JIS
T.101-G2	
T.61-7bit	
T.61-8bit	
TIS-620	
TSCII	
UNICODE-1-1	
UNICODE-1-1-UTF-7	
UNKNOWN-8BIT	
US-ASCII	
UTF-16	
UTF-16BE	UTF-16BE
UTF-16LE	UTF-16LE
UTF-32	
UTF-32BE	
UTF-32LE	
UTF-7	
UTF-8	UTF-8
VIQR	
VISCII	
Ventura-International	
Ventura-Math	
Ventura-US	
Windows-31J	
dk-us	
greek-ccitt	
greek7	
greek7-old	
hp-roman8	
iso-ir-90	
latin-greek	
latin-lap	
macintosh	macintosh
us-dk	
videotex-suppl	
windows-1250	windows-1250
windows-1251	windows-1251
windows-1252	windows-1252
windows-1253	windows-1253
windows-1254	windows-1254
windows-1255	windows-1255
windows-1256	windows-1256
windows-1257	windows-1257

windows-1258	windows-1258
windows-874	windows-874

Annex B: Known encodings not present in IANA

Lists of encoding known to some platforms but not registered to IANA. This might be incomplete as generating the list proved challenging. These might still be supported through the `other` mib but are not suitable for interexchange.

Windows

- 710 Arabic - Transparent Arabic
- 72 DOS-720 Arabic (Transparent ASMO); Arabic (DOS)
- 737 ibm737 OEM Greek (formerly 437G); Greek (DOS)
- 875 cp875 IBM EBCDIC Greek Modern
- 1361 Johab Korean (Johab)
- 57002 x-iscii-de ISCII Devanagari
- 57003 x-iscii-be ISCII Bangla
- 57004 x-iscii-ta ISCII Tamil
- 57005 x-iscii-te ISCII Telugu
- 57006 x-iscii-as ISCII Assamese
- 57007 x-iscii-or ISCII Odia
- 57008 x-iscii-ka ISCII Kannada
- 57009 x-iscii-ma ISCII Malayalam
- 57010 x-iscii-gu ISCII Gujarati
- 57011 x-iscii-pa ISCII Punjabi

Iconv

- CP1131
- CP1133
- GEORGIAN-ACADEMY
- GEORGIAN-PS
- CN-GB-ISOIR165

- Johab
- MacArabic
- MacCentralEurope
- MacCroatian
- MacCyrillic
- MacGreek
- MacHebrew
- MacIceland
- MacRoman
- MacRomania
- MacThai
- MacTurkish
- MacUkraine

References

[1] Peter Brett. P2498R0: Forward compatibility of text_encoding with additional encoding registries. https://wg21.link/p2498r0, 12 2021.

[2] Peter Brett. P2498R1: Forward compatibility of text_encoding with additional encoding registries. https://wg21.link/p2498r1, 1 2022.

[3] Lawrence Crowl. N2346: Defaulted and deleted functions. https://wg21.link/n2346, 7 2007.

[4] Corentin Jabot. P2460R0: Relax requirements on wchar_t to match existing practices. https://wg21.link/p2460r0, 10 2021.

[5] Corentin Jabot and Peter Brett. P1885R8: Naming text encodings to demystify them. https://wg21.link/p1885r8, 10 2021.

[6] Bryce Adelstein Lelbach. P2455R0: 2021 november library evolution poll outcomes. https://wg21.link/p2455r0, 12 2021.

[7] Jens Maurer. P2491R0: Text encodings follow-up. https://wg21.link/p2491r0, 11 2021.

[8] Andrew Tomazos. P1402R0: std::cstring_view - a c compatible std::string_view adapter. https://wg21.link/p1402r0, 1 2019.

[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

[ianacharset-mib] IANA *IANA Charset MIB*
https://www.iana.org/assignments/ianacharset-mib/ianacharset-mib

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

[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