P1885 - Naming Text Encodings

Corentin Jabot
Peter Brett

New since last time

- Better wording
- rename `system` to `environment`
- Add some explanation regarding some design choices
- Fix the algorithm used to compare encoding names
  - Remove 2 very old encoding conflicting with other names
- Add UTF7-IMAP following its registration

Problem

There are a number of text encodings used by a C++ program

- Encodings for literals (‘a’, “Hello”, L’a’, L“Hello”)
- Encoding of the environment
- Encodings associated with stream objects (`basic_ios`), or locale functions
  (`<cctype>`, `<cwctype>`, `<locale>`, `<clocale>` `<cwchar>`, `<cstdlib>`, …)

TL;DR: Everything dealing with text (as opposed to bytes) has some encoding attached to it

Problem

The encoding of literals is implementation-defined

- `/execution-charset:utf-8 (MSVC) -fexec-charset=utf-8 (GCC)`
- `MSVC uses the system locale by default to determine the execution encoding`
- `GCC defaults to UTF-8`
- `Clang always uses UTF-8 (for now)`

The encoding assumed by C/system functions is “locale-specific”

- C (and most operating systems, including POSIX) derives the encoding from the locale and changing the locale affects how text classification, transcoding and transformations behave

Problem

Is the encoding EBCDIC, UTF-8, windows-1252, shift-jis?

No way to know !

The system is a blackbox

Problem

- Can't verify that the compiler/environment behaves as desired
- Have to rely on C functions for text handling

```
mblen, mbtowc, wctomb, wctomb_s, wcstombs, wcstombs_s,
mbstowcs, mbstowcs_s, btowc, wctob, mbrlen, mbrtowc,
wcrtomb, wcrtomb_s, mbsrtowcs, mbsrtowcs_s, wcsrtombs,
wcsrtombs_s, mbrtoc16, c16rtomb, mbrtoc32, c32rtomb

mcntomwcn, mcnrtomwcn, mcsntomwcsn, mcsnrtomwcsn,
mcntoc8n, mcnrtoc8n, mcsntoc8sn, mcsnrtoc8sn,
mcntoc16n, mcnrtoc16n, mcsntoc16sn, mcsnrtoc16sn,
mcntoc32n, mcnrtoc32n, mcsntoc32sn, mcsnrtoc32sn,
c8ntomcn, c8nrtomcn, c8sntomcsn, c8snrtomcsn,
c16ntomcn, c16nrtomcn, c16sntomcsn, c16sntomcsn,
c32ntomcn, c32nrtomcn, c32sntomcsn, c32snrtomcsn
mwcntomcn, mwcnrtomcn, mwcsntomcsn, mwcsnrtomcsn,
mwcntoc8n, mwcnrtoc8n, mwcsntoc8sn, mwcsnrtoc8sn,
mwcntoc16n, mwcnrtoc16n, mwcsntoc16sn, mwcsnrtoc16sn
mwcntoc32n, mwcnrtoc32n, mwcsntoc32sn, mwcsnrtoc32sn,
c8ntomwcn, c8nrtomwcn, c8sntomwcsn, c8snrtomwcsn,
c16ntomwcn, c16nrtomwcn, c16sntomwcsn, c16snrtomwcsn,
c32ntomwcn, c32nrtomwcn, c32sntomwcsn, c32snrtomwcsn
```

(See [N2620] for newly proposed C transcoding functions,

Thanks JeanHeyd Meneide)

LWG3314

- `std::chrono::microseconds` was initially formatted with “µ” as symbol unit
- What if “µ” is not representable?

Simple solution

Add functions to indicate what the different encodings are !

```
consteval auto literal_encoding();
auto environmemt_encoding();
auto locale::encoding() const;

consteval auto wide_literal_encoding();
auto wide_environmemt_encoding();
auto locale::wide_encoding() const;
```

Wait… what do these functions return?

Or, dealing with 60 years of mess !

- Lots and lots of encodings over the decades
- Systems API return different things (string on POSIX, code page on windows, etc)
- All encodings have slightly different names across different systems

We want something that is

- Meaningful for users (no black box)
- Portable
- Can be used by different libraries as a vocabulary type and for compatibility with existing libraries (Qt, ICU, iconv)

Solution

RFC3808/ IANA Charset registry

Extensive registry of encodings (~250) with:

- Name (ascii strings)
- Unique numeric identifiers
- Name aliases

Large buy-in from vendors (windows, IBM, iconv, etc)

```cpp
struct text_encoding {

  inline constexpr size_t max_name_length = 63;
  enum class id : int_least32_t;

  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;
  constexpr auto aliases() const noexcept;

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

  static consteval text_encoding literal();
  static consteval text_encoding wide_literal();
  static text_encoding environment() noexcept;
  static text_encoding wide_environment() noexcept;
  template<id id_> static bool text_encoding::environment_is() noexcept;
  template<id id_> static bool text_encoding::wide_environment_is() noexcept;
};
```

```
struct text_encoding {

  inline constexpr size_t max_name_length = 63;
  enum class id : int_least32_t;

  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;
  constexpr auto aliases() const noexcept;

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

  static consteval text_encoding literal();
  static consteval text_encoding wide_literal();
  static text_encoding environment() noexcept;
  static text_encoding wide_environment() noexcept;
  template<id id_> static bool text_encoding::environment_is() noexcept;
  template<id id_> static bool text_encoding::wide_environment_is() noexcept;
};
```

Entire API

```
class locale {
public:
  string name() const;
  text_encoding encoding() const;
  text_encoding wide_encoding() const;
};
```

```
enum class id: int_least32_t;
```

Forward compatibility with the RFC spec, which defines the enum as an INTEGER which is 32 bits. The goal is to avoid binary compatibility issues in the future

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

Compatibility with C APIS, notably with iconv

```
iconv_t iconv_open(const char *tocode, const char *fromcode);
```

UTF-8 everywhere

```
static_assert(text_encoding::literal() == text_encoding::id::UTF8,
              "Qt must be compiled with /utf-8");
```

LWG3314

```
const char* micro_suffix = [] {
    if constexpr (text_encoding::literal() == text_encoding::id::UTF8) {
        return "µs";
    }
    else {
        return "us";
    }
}();
```

std::print

P2093R2

```
template <typename... Args>
void print(string_view fmt, const Args&... args) {
  if (text_encoding::literal() == text_encoding::id::UTF8)
    vprint_unicode(fmt, make_format_args(args...));
  else
    vprint_nonunicode(fmt, make_format_args(args...));
}
```

Compatibility across libraries

```
// Qt
auto codec = QTextCodec::codecForMib(std::text_encoding::system().mib());

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

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

Status of this proposal

- Implemented for POSIX / Windows
- Compiler intrinsics for literal() / wide_literal() upstreamed in GCC and Clang- thanks JeanHeyd 🎉
- Planned for MSVC, maybe? https://developercommunity.visualstudio.com/t/-Compiler-Feature--Macro-for-Narrow-Li/1160821
- Reviewed by SG16🎉
- We have wording

Implementation

- `nl_langinfo` on POSIX, `GetACP on Windows`
- Some functions (aliases, the constructor taking a name) cause a table of names to be ODR-used, that's why we have the templated `(wide_)system_is` functions
- Designed to be freestanding (non-allocating, non-throwing)
- The non-consteval functions are allowed to return “unknown” for implementers who do not control the libc and when `nl_langinfo is not available`

Design decisions / constraints

- The constructor takes a string_view but returns a const char*, because icu, iconv, etc expect `const char*` (so the name is copied in a fixed size buffer)
- Aliases is an implementation defined type which models view and whose element type is `const char*` (again, because of iconv, etc)
- The enum name provides all IANA registered encodings and their value.
- Support for custom encodings is provided by passing a name not matching any encoding. In which case `mib()` will be `other`

Why in the standard

- **It requires compiler magic**
- More efficient implementations are possible with some of the implementation moved to the libc
- It helps to make the encoding model less opaque which is an issue for teaching.

Note on bikesheding

- There is a CWG proposal for improving the wording of character sets during translation and execution https://wg21.link/p2314r2
- The names of the functions literal, wide_literal environment should match between wording and API.

There and Back Again

The intent of P1885 is to label known scenario

NOT TO DESCRIBE everything

Disclaimer

We are dealing with many decades of progress and missteps

- Terminology Varies
- Evolve
- Not everything fits perfectly

Encodings, reminder

Encoding	Fixed Width	Multibytes/Variable width
Single Byte	ASCII*	UTF-8/**Big5**
Double Byte	UCS2/EUC Fixed Width/IBM DBCS code scheme/**Big5**	UTF-16

- 7 bytes encodings
- UTF-9 and UTF-18 (April fool jokes)
- UTF-32 (quadruple bytes)

Wide interfaces

- Wide chars are a C/C++ inventions that map to single/double/quadruple bytes encoding depending on sizeof(wchar_t)
- Wide interfaces are provided mostly for consistency
- **Most registered characters encodings are single-byte** because most encodings, with the notable exceptions of UTF-16 on windows, **are single byte**.
- Implementations make up wide encodings because the standard force them too.

"Interesting" scenario

- CHAR_BITS !=8
  - We don't care, a byte is still a code unit, there is padding and decoder can cope with that, again because they can read out bytes
- CHAR_BITS >= 16
  - char can be UTF-16 technically
  - wchar can also be UTF-16
- CHAR_BITS == 8, sizeof(wchar_t) == 2, encoding == UTF-32
  - Non conforming
- CHAR_BITS == 8, sizeof(wchar_t) == 1, encoding == UTF-16
  - Non conforming
- CHAR_BITS == 8, sizeof(wchar_t) == 4, encoding == UTF-16
  - Conforming
  - But the sequence of byte is not valid UTF-16
  - A non-hostile implementation should not call that UTF-16

Use case: iconv

```
size_t iconv(iconv_t cd,
             char **restrict inbuf, size_t *restrict inbytesleft,
             char **restrict outbuf, size_t *restrict outbytesleft);

QString QTextDecoder::toUnicode(const char *chars, int len);

void ucnv_toUnicode(  UConverter *  converter,
                      UChar **     target,
                      const UChar *      targetLimit,
                      const char **      source,
                      const char *       sourceLimit,
                      int32_t *    offsets,
                      UBool        flush,
                      UErrorCode *       err );
```

Endianness

- A double (or quadruple) byte encoding has an endianness
- `text_encoding has no endianness invariant`
- `The C++ Abstract machine scalar type do`
- `wide_literal/wide_environment` returns an encoding object which is implied to describe the same endianness as the rest of the system.
- IF CHAR_BITS != 8 encodings are still in the platform endianness, and it works, albeit produced text are not portable.

UTF-16: Context Matter

Table 2-4. The Seven Unicode Encoding Schemes

Encoding Scheme	Endian Order	BOM Allowed?
UTF-8	N/A	yes
UTF-16	Big-endian or little-endian	yes
UTF-16BE	Big-endian	no
UTF-16LE	Little-endian	no
UTF-32	Big-endian or little-endian	yes
UTF-32BE	Big-endian	no
UTF-32LE	Little-endian	no

UTF-16: Context Matter

Encoding Scheme Versus Encoding Form. Note that some of the Unicode encoding schemes have the same labels as the three Unicode encoding forms. This could cause confusion, so it is important to keep the context clear when using these terms: character encoding forms refer to integral data units in memory or in APIs, and byte order is irrelevant; character encoding schemes refer to byte-serialized data, as for streaming I/O or in file storage, and byte order must be specified or determinable.

The Internet Assigned Numbers Authority (IANA) maintains a registry of charset names used on the Internet. Those charset names are very close in meaning to the Unicode character encoding model's concept of character encoding schemes, and all of the Unicode character encoding schemes are, in fact, registered as charsets. While the two concepts are quite close and the names used are identical, some important differences may arise in terms of the requirements for each, particularly when it comes to handling of the byte order mark. Exercise due caution when equating the two.

UTF-16: Context Matter

Encoding scheme vs encoding

- **UTF16/UTF-32 SPECIFIC TERMINOLOGY**
- **Not very (at all) relevant**

P1885

- Environment and literal encodings are always in the native endianness
- No BOM in general
- No networking or unknown files encoding involved

- Returning UTF-16 is correct, and more user friendly
- Returning UTF-16<native endianness is also correct>
  - Let's recommend one
  - Windows, and users uses "UTF-16"

P1885

- Environment and literal encodings are always in the native endianness
- No BOM in general
- No networking or unknown files encoding involved

- Returning UTF-16 is correct, and more user friendly
- Returning UTF-16<native endianness is also correct>
    - Let's recommend one
    - Windows, and users uses "UTF-16"

Recent Changes

- Can return unknown from wide_literal()

Recommended practice

- Implementations should prefer returning UTF-16/UTF-32 over UTF-16BE/UTF-16LE/UTF-32BE/UTF-32LE.
- Implementations should otherwise 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).
- The encodings returned from `wide_literal` and `wide_environments` should describe encodings in the native endianness.
- The encodings returned from `wide_literal` and `wide_environments` should describe encodings whose code unit types is represented by `sizeof(wchar_t)` octets.

P1885

- We can go with the recommended practices
- We can get rid of wide methods
- We can stop working on this paper