Wording improvements for encodings and character sets

Document #: P2297R0
Date: 2021-01-31
Project: Programming Language C++
Audience: SG-16, CWG
Reply-to: Corentin Jabot <corentin.jabot@gmail.com>

Abstract

Summary of changes

`wchar_t`

In the core wording, `wchar_t` is modified to be allowed to represent code units of variable length and shift state encodings. This aligns the standard with standard practices (the execution encoding is UTF-16 on windows).

🔔

The wording mandated that the executions encoding be able to encode ”alert, backspace, and carriage return”. This requirement is not used in the core wording (Tweaks of [5.13.3.3.1] may be needed), nor in the library wording, and therefore does not seem useful, so it was not added in the new wording. This will not have any impact on existing implementations.

New terminology

Basic character set

Formerly *basic source character set*. Represent the set of abstract (non-coded) characters in the graphic subset of the ASCII character set. The term ”source” has been dropped because the source code encoding is not observable nor relevant past phase 1.

The *basic character set* is used:

- As a subset of other encodings
- To restric accepted characters in grammar elements
- To restrict values in library

literal character set*, *literal character encoding*, *wide literal character set*, *wide literal character encoding

Encodings and associated character sets of narrow and wide character and string literals. Implementation defined, and locale agnostic.

execution character set*, *execution character encoding*, *wide execution character set*, *wide execution character encoding

Encodings and associated character sets of the encoding used by the library. isomorphic or supersets of their literal counterparts. Separating literal encodings from libraries encoding allows:

- To make a distinction that exists in practice and which was not previously admitted by the standard previous.
- To keep the core wording locale agnostic.

The definition for these encodings has been moved to [library.intro]

Questions and bikesheding

- Do the terms of art code unit, code point, abstract character need to be defined?
- Are we happy with `execution` for library encodings? (alternatives : runtime, system, environment, etc)
- Do we prefer *literal character encoding* or *literal ordinary character encoding* ?

Future works

- Review support for variable-length wide execution encodings in the library
- Review usages of the terms character

Wording

� Terms and definitions [intro.defs]

[...]

> Rationale: The notion of extended characters is removed, as, while the notion of basic character is useful, there are only a few places where basic characters should be handled differently from other characters (character meaning code point here).
> TODO: Should that definition apply to the UTF-8 (char8_t) encoding?

multibyte character
~~sequence of one or more bytes representing a member of the extended character set of either the source or the execution environment~~ Sequence of one or more code units representing a member of the literal or exection character set.

~~[*Note:* The extended character set is a superset of the basic character set. — *end note*]~~

[...]

� Memory and objects [basic.memobj]

� Memory model [intro.memory]

The fundamental storage unit in the C++ memory model is the *byte*. A byte is at least large enough to ~~contain any member of the basic execution character set~~ represent any code unit of the literal and execution character encodings and the eight-bit code units of the Unicode UTF-8 encoding form and is composed of a contiguous sequence of bits, the number of which is implementation-defined.

� Fundamental types [basic.fundamental]

[...]

> Rationale: The wording was not clear that it meant the basic source (rather than execution) character set. ”implementation’s basic character set” is also a fuzzy term. Is the basic source character set a coded character set?

Type `char` is a distinct type that has an implementation-defined choice of “`signed char`” or “`unsigned char`” as its underlying type. The values of type `char` can represent ~~distinct codes for all members of the implementation’s basic character set~~ all code units of the literal and execution character encodings. The three types `char`, `signed char`, and `unsigned char` are collectively called *ordinary character types*. The ordinary character types and `char8_t` are collectively called *narrow character types*. For narrow character types, each possible bit pattern of the object representation represents a distinct value. [*Note:* This requirement does not hold for other types. — *end note*] [*Note:* A bit-field of narrow character type whose width is larger than the width of that type has padding bits; see **??**. — *end note*]

> Rationale: The wording was implying that UTF-16 could not be used with wchar_t (as it is a multibyte encoding and therefore can not represent all values in a single wchar_t)

Type `wchar_t` is a distinct type that has an implementation-defined signed or unsigned integer type as its underlying type. The values of type `wchar_t` can represent ~~distinct codes for all members of the largest extended character set specified among the supported locales~~ all code units of the wide literal and wide execution character encodings.

� Phases of translation [lex.phases]

> Translation phase 1 is affected by the removal of the definition of ”extended character set”, no wording is currently provided as a few other papers are redrafting phase 1 such that further modifications should not be necessary.

� Character sets [lex.charset]

The *basic* ~~*source*~~ *character set* consists of 96 characters: the space character, the control characters representing horizontal tab, vertical tab, form feed, and new-line, plus the following 91 graphical characters:

> Edit the footnote associated with the above paragraph as follows:
> Rationale:
>
> - Abstract character is a more precise terminology to talk about the same characters in different character sets or not in any character set.
> - The second sentence seems incorrect. While the mapping in phase 1 must be documented, neither the source files nor the internal representation should be observable by the program and as such do not need to be documented. The paragraph further seems to imply that the formerly-source basic character set applies to source files
>
> > The glyphs for the members of the basic ~~source~~ character set are intended to identify abstract characters from the subset of ISO/IEC 10646 which corresponds to the ASCII character set. ~~However, the mapping from source file characters to the source character set (described in translation phase 1) is specified as implementation-defined, and therefore implementations must document how the basic source characters are represented in source files.~~

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
_ { } [ ] # ( ) < > % : ; . ? * + - / ^ & | ~ ! = , \ " '
```

The *universal-character-name* construct provides a way to name other characters.

hex-quad:
    hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit

universal-character-name:
    `\u` *hex-quad*
    `\U` *hex-quad hex-quad*

A *universal-character-name* designates the character in ISO/IEC 10646 (if any) whose code point is the hexadecimal number represented by the sequence of *hexadecimal-digit* s in the *universal-character-name*. The program is ill-formed if that number is not a code point or if it is a surrogate code point. Noncharacter code points and reserved code points are considered to designate separate characters distinct from any ISO/IEC 10646 character. If a *universal-character-name* outside the *c-char-sequence*, *s-char-sequence*, or *r-char-sequence* of a *character-literal* or *string-literal* (in either case, including within a *user-defined-literal*) corresponds to a control character or to a character in the basic ~~source~~ character set, the program is ill-formed. A sequence of characters resembling a *universal-character-name* in an *r-char-sequence* does not form a *universal-character-name*. [*Note:* ISO/IEC 10646 code points are integers in the range $[0, 10\mathrm{FFFF}]$ (hexadecimal). A surrogate code point is a value in the range $[\mathrm{D800}, \mathrm{DFFF}]$ (hexadecimal). A control character is a character whose code point is in either of the ranges $[0, 1\mathrm{F}]$ or $[7\mathrm{F}, 9\mathrm{F}]$ (hexadecimal). — *end note*]

The *basic execution character set* and the *basic execution wide-character set* shall each contain all the members of the basic source character set, plus control characters representing alert, backspace, and carriage return, plus a *null character* (respectively, *null wide character*), whose value is 0. For each basic execution character set, the values of the members shall be non-negative and distinct from one another. In both the source and execution basic character sets, the value of each character after `0` in the above list of decimal digits shall be one greater than the value of the previous. The *execution character set* and the *execution wide-character set* are implementation-defined supersets of the basic execution character set and the basic execution wide-character set, respectively. The values of the members of the execution character sets and the sets of additional members are locale-specific.

The *literal character set* and *wide literal character set* are implementation-defined characters set which shall contain all members of the *basic character set* plus an implementation-defined set of additional members.

The *literal character encoding* and *wide literal character encoding* are the implementation-defined character encodings of the *literal character set* and *wide literal character set* respectively such that:

- Each code unit is represented by a single `char` or `wchar_t` respectively
- Each code point is represented by one or more code units.
- Each member of the *basic character set* is uniquely represented by a single byte whose value is positive
- The NULL character (U+0000) is represented as a single code unit whose value, as read via a glvalue of type `char`, is 0
- The code units representing each digit in the basic character set (U+0030 to U+0039) have consecutive values

> Do we still need the gvalue bit above? My understanding is that we are trying to say `char(L'\0') == 0`.

[...]

� Character literals [lex.ccon]

> The grammar below will be further impacted by work to not replace non-basic characters in phase 1

basic-c-char:
any member of the basic ~~source~~ character set except the single-quote `'`, backslash `\`, or new-line character

> I think we want to limit to basic characters here

conditional-escape-sequence-char:
any member of the basic ~~source~~ character set that is not an *octal-digit*, a *simple-escape-sequence-char*, or the characters `u`, `U`, or `x`

[...]

The kind of a *character-literal*, its type, and its associated character encoding are determined by its *encoding-prefix* and its *c-char-sequence* as defined by . The special cases for non-encodable character literals and multicharacter literals take precedence over their respective base kinds. [*Note:* The ~~associated character encoding for ordinary and wide character literals~~ ordinary and wide literal character encodings determines encodability, but does not determine the value of non-encodable ordinary or wide character literals or ordinary or wide multicharacter literals. The examples in [lex.ccon.literal] for non-encodable ordinary and wide character literals assume that the specified character lacks representation in the ~~execution~~ literal character set or ~~execution~~ literal wide-character set, respectively, or that encoding it would require more than one code unit. — *end note*]

Table 1: Character literals

Encoding prefix	Kind	Type	Associated character encoding	Example
none	*ordinary character literal*	`char`	~~encoding of~~ literal	`'v'`
	non-encodable ordinary character literal	`int`	~~the execution~~ encoding	`'\U0001F525'`
	ordinary multicharacter literal	`int`	~~character set~~	`'abcd'`
`L`	*wide character literal*	`wchar_t`	~~encoding of~~ wide literal	`L'w'`
	non-encodable wide character literal	`wchar_t`	~~the execution~~ encoding	`L'\U0001F32A'`
	wide multicharacter literal	`wchar_t`	~~wide-character set~~	`L'abcd'`
`u8`	*UTF-8 character literal*	`char8_t`	UTF-8	`u8'x'`
`u`	*UTF-16 character literal*	`char16_t`	UTF-16	`u'y'`
`U`	*UTF-32 character literal*	`char32_t`	UTF-32	`U'z'`

� String literals [lex.string]

> The grammars below will be further impacted by work to not replace non-basic characters in phase 1

basic-s-char:
any member of the basic ~~source~~ character set except the double-quote `"`, backslash `\`, or new-line character

d-char:
any member of the basic ~~source~~ character set except:
space, the left parenthesis `(`, the right parenthesis `)`, the backslash `\`, and the control characters
representing horizontal tab, vertical tab, form feed, and newline.

[...]

Table 2: String literals

Encoding prefix	Kind	Type	Associated character encoding	Examples

none	*ordinary string literal*	array of n `const char`	~~encoding of the execution character set~~ literal encoding	`"ordinary string"`
				`R"(ordinary raw string)"`

`L`	*wide string literal*	array of n `const wchar_t`	~~encoding of the execution wide-character set~~ wide literal encoding	`L"wide string"`
				`LR"w(wide raw string)w"`

`u8`	*UTF-8 string literal*	array of n `const char8_t`	UTF-8	`u8"UTF-8 string"`
				`u8R"x(UTF-8 raw string)x"`

`u`	*UTF-16 string literal*	array of n `const char16_t`	UTF-16	`u"UTF-16 string"`
				`uR"y(UTF-16 raw string)y"`

`U`	*UTF-32 string literal*	array of n `const char32_t`	UTF-32	`U"UTF-32 string"`
				`UR"z(UTF-32 raw string)z"`

A *string-literal* that has an `R` in the prefix is a *raw string literal*. The *d-char-sequence* serves as a delimiter. The terminating *d-char-sequence* of a *raw-string* is the same sequence of characters as the initial *d-char-sequence*. A *d-char-sequence* shall consist of at most 16 characters.

[*Note:* The characters `'('` and `')'` are permitted in a *raw-string*. Thus, `R"delimiter((a|b))delimiter"` is equivalent to `"(a|b)"`. — *end note*]

[*Note:* A source-file new-line in a raw string literal results in a new-line in the ~~resulting execution~~ evaluated string literal. Assuming no whitespace at the beginning of lines in the following example, the assert will succeed:

```
const char* p = R"(a\
b
c)";
assert(std::strcmp(p, "a\\\nb\nc") == 0);
```

— *end note*]

[...]

� User-defined literals [lex.ext]

[...]

If *L* is a *user-defined-integer-literal*, let *n* be the literal without its *ud-suffix*. If *S* contains a literal operator with parameter type `unsigned long long`, the literal *L* is treated as a call of the form

`operator "" ` *X*(*n*ULL)

Otherwise, *S* shall contain a raw literal operator or a numeric literal operator template but not both. If *S* contains a raw literal operator, the literal *L* is treated as a call of the form

`operator "" ` *X*("*n*")

Otherwise (*S* contains a numeric literal operator template), *L* is treated as a call of the form

`operator "" ` *X*<'c_1', 'c_2', ... 'c_k'>()

where *n* is the source character sequence $c_1c_2...c_k$. [*Note:* The sequence $c_1c_2...c_k$ can only contain characters from the basic ~~source~~ character set. — *end note*]

If *L* is a *user-defined-floating-point-literal*, let *f* be the literal without its *ud-suffix*. If *S* contains a literal operator with parameter type `long double`, the literal *L* is treated as a call of the form

`operator "" ` *X*(*f*L)

Otherwise, *S* shall contain a raw literal operator or a numeric literal operator template but not both. If *S* contains a raw literal operator, the *literal L* is treated as a call of the form

`operator "" ` *X*("*f*")

Otherwise (*S* contains a numeric literal operator template), *L* is treated as a call of the form

`operator "" ` *X*<'c_1', 'c_2', ... 'c_k'>()

where *f* is the source character sequence $c_1c_2...c_k$. [*Note:* The sequence $c_1c_2...c_k$ can only contain characters from the basic ~~source~~ character set. — *end note*]

� Library introduction [library]

� Method of description [library.c]

� Other conventions [conventions]

� Type descriptions [type.descriptions]

� Character sequences [character.seq]

� Execution encodings [execution encodings]

The *execution encoding* is the character encoding of the *execution character set*, such that all members of the *literal character set* are represented, with the same value in the *execution character set* and any sequence of characters in the *literal character encoding* represent the same sequence of code points when interpreted as being in the *execution encoding*.

The *wide execution encoding* is the character encoding of the *wide execution character set*, such that all members of the *wide literal character set* are represented, with the same value in the *wide execution character set* and any sequence of characters in the *wide literal character encoding* represent the same sequence of code points when interpreted as being in the *wide execution encoding*.

The *execution encoding* and *wide execution encoding* are implementation-defined and may be be affected by a call to `setlocale(int, const char*)`, or by a change to a `locale` object, as described in **??** and **??**.

> The paragraph below only becomes relevant if we have constexpr text transformation, encodings or classification functions. I don't think that's the case yet.

During constant evaluation, the *execution encoding* and *execution character set* are the *literal character set* and *wide literal character set* respectively and are not affected by locale.

� General [character.seq.general]

The C standard library makes widespread use of characters and character sequences that follow a few uniform conventions:

- A *letter* is any of the 26 lowercase or 26 uppercase letters in the ~~basic execution~~ basic character set.
- The *decimal-point character* is the (single-byte) character used by functions that convert between a (single-byte) character sequence and a value of one of the floating-point

types. It is used in the character sequence to denote the beginning of a fractional part. It is represented in [??] through [??] and **??** by a period, `'.'`, which is also its value in the `"C"` locale, but may change during program execution by a call to `setlocale(int, const char*)`, or by a change to a `locale` object, as described in **??** and **??**.

- A *character sequence* is an array object *A* that can be declared as *T A*[*N*], where *T* is any of the types `char`, `unsigned char`, or `signed char`, optionally qualified by any combination of `const` or `volatile`. The initial elements of the array have defined contents up to and including an element determined by some predicate. A character sequence can be designated by a pointer value *S* that points to its first element.

� Byte strings [byte.strings]

A *null-terminated byte string*, or NTBS, is a character sequence whose highest-addressed element with defined content has the value zero (the *terminating null character*); no other element in the sequence has the value zero.

The *length of an NTBS* is the number of elements that precede the terminating null character. An *empty NTBS* has a length of zero.

The *value of an NTBS* is the sequence of values of the elements up to and including the terminating null character.

A *static NTBS* is an NTBS with static storage duration.

� Multibyte strings [multibyte.strings]

A *null-terminated multibyte string*, or NTMBS, is an NTBS that constitutes a sequence of valid multibyte characters, beginning and ending in the initial shift state.

> Edit the footnote attached to the above sentence as follow:
>
> > An NTBS that ~~contains characters only from the basic execution character set is also an NTMBS. Each multibyte character then consists of a single byte~~ only contains characters represented as a single byte is also an NTMBS .

A *static NTMBS* is an NTMBS with static storage duration.

� Locales [locales]

� Class `locale` [locale]

� `ctype` members [locale.ctype.members]

```
charT        do_widen(char c) const;
const char*  do_widen(const char* low, const char* high, charT* dest) const;
```

Effects: Applies the simplest reasonable transformation from a `char` value or sequence of `char` values to the corresponding `charT` value or values. The only characters for which unique transformations are required are those in the basic ~~source~~ character set.

For any named `ctype` category with a `ctype<charT>` facet `ctc` and valid `ctype_base::mask` value `M`, `(ctc.is(M, c) || !is(M, do_widen(c)) )` is `true`.

The second form transforms each character `*p` in the range `[low, high)`, placing the result in `dest[p - low]`.

Returns: The first form returns the transformed value. The second form returns `high`.

```
char         do_narrow(charT c, char dfault) const;
const charT* do_narrow(const charT* low, const charT* high, char dfault, char* dest) const;
```

Effects: Applies the simplest reasonable transformation from a `charT` value or sequence of `charT` values to the corresponding `char` value or values.

For any character `c` in the basic ~~source~~ character set the transformation is such that

```
do_widen(do_narrow(c, 0)) == c
```

�   Time library   [time]

Table 3: Meaning of `parse` flags

Flag	Parsed value
`%a`	The locale's full or abbreviated case-insensitive weekday name.
`%Z`	The time zone abbreviation or name. A single word is parsed. This word can only contain characters from the basic ~~source~~ character set that are alphanumeric, or one of `'_'`, `'/'`, `'-'`, or `'+'`.
`%%`	A `%` character is extracted.

�   C++ and ISO C++ 2014   [diff.cpp14]

�   ??: lexical conventions   [diff.cpp14.lex]

Change: Removal of trigraph support as a required feature.
Rationale: Prevents accidental uses of trigraphs in non-raw string literals and comments.
Effect on original feature: Valid C++ 2014 code that uses trigraphs may not be valid or may have different semantics in this revision of C++. Implementations may choose to translate trigraphs as specified in C++ 2014 if they appear outside of a raw string literal, as part of the implementation-defined mapping from physical source file characters to the basic ~~source~~ character set.

Acknowledgments

References

[N4878] Richard Smith *Working Draft, Standard for Programming Language C++*
https://wg21.link/N4878