Extended locale-specific presentation specifiers for std::format

Document No.	**P1892 R1**	Date	2019-11-07
Reply To	Peter Brett **pbrett@cadence.com**	Audience:	LWG

Revisions

R1	Change *locale-specific form* modifier character from ‘`n`’ to ‘`L`’. Add before/after comparison tables. Add Belfast straw poll results. Revise wording based on LWG feedback.

Introduction

`std::format` provides a safe and extensible alternative to the `printf` family of functions for formatted output. By default, it does not use locale-aware numeric formatting. It provides a ‘`n`’ format specifier so that users can request locale-aware formatting if required [1].

`format` has overloads that allow specifying a particular locale; it is not limited to using the current global locale [2]. This makes `format` the best approach to locale-aware formatted output.

This paper arises from national body comment GB226 [3]. If this paper was applied to the C++20 Committee Draft, then it would resolve the NB comment.

The ‘`n`’ specifier currently only allows for a restrictive subset of locale-aware presentations. This particularly precludes access to useful floating-point formats.

This paper proposes to replace the ‘`n`’ specifier with a ‘`L`’ modifier which requests locale-aware formatting with any other specifier. For example:

```
format(locale("de_DE"), "{:e} {:Le}", 101, 202);
// => "1.01e+02 2,02e+02"
```

This would ensure that all locale-specific presentations that are provided by `printf` can be obtained using `format`.

Design

Easy to explain as an extension of existing syntax

Currently, using the ‘`n`’ specifier for formatting numeric values gives the same result as using no specifier at all, but with locale-specific digit group and decimal radix separator characters. For example:

```
format(locale("en_GB"), "{:} {:n}", 10000, 20000);
// => "10000 20,000"
```

This behaviour can be explained in terms ‘`n`’ as a modifier: here, it is *modifying* the default formatting in a locale-specific way.

The use model proposed in this paper is to start with a basic conversion specifier and then use ‘`L`’ to modify it to be locale-aware. This fits with the “opt-in to locale” design of `format`.

Locale-specific modifier is placed before the format specifier

In the current formatting mini-language, all modifiers occur before the type specifier, and the type specifier is optional. Placing the ‘`L`’ modifier immediately before the type specifier preserves this ordering, without affecting existing code.

Presentation of numbers with non-decimal radix

When formatting numbers for reading with a non-decimal radix, it can still be useful to use the locale to group the digits. Locale-specific binary, octal and hexadecimal presentation is therefore deliberately included. For example:

```
struct group2 : numpunct<int> {
    char do_thousands_sep() const { return '_'; }
    string do_grouping() const { return '\2'; }
};

locale grouped(locale("C"), new group2);
format(grouped, "{:Lx}", 12345678);
// => "BC_61_4E"
```

Presentation of Boolean values

The current default presentation for `bool` when no specifier is provided is one of the English words “true” and “false”. When modified with ‘`n`’, congruence can be achieved if the presentation uses the locale’s `numpunct` facet. For example:

```
struct french_bool : numpunct<char> {
    string_type do_truename() const { return "vrai"; }
    string_type do_falsename() const { return "faux"; }
};

locale french(locale("fr_FR"), new french_bool);
format(french, "{:L}", true);
// => "oui"
```

Impact on existing code

All previously well-formed format strings continue to work unchanged. Some format strings that were previously ill-formed become well-formed.

Replacement of ‘n’ with ‘L’

SG16 expressed a strong consensus that, if a modifier was used to provide locale-specific presentation with `format`, then that modifier should not be spelled ‘`n`’.

- ‘`n`’ does not obviously have a semantic connection with ‘locale’.
- ‘`l`’ can easily be mistaken for ‘`1`’ in many fixed-width fonts, and programmers do not always have control over the fonts used to display the code that they write
- ‘`L`’ is big and shouty about the fact that locale information is being used, and hard to overlook

Alternatives

If C++20 is not revised to make the ‘`n`’ format specifier congruent with default formatting, then this paper will be updated to propose an optional ‘`L`’ modifier and the deprecation of ‘`n`’ in C++23.

This would be inferior to generalizing ‘`n`’ in two respects:

- Many pieces of wording would need to be duplicated to describe the effects of ‘n’ and ‘L’ and their subtle differences.
- Teachability would be impaired by needing to explain the differences between ‘n’ and ‘L’ and when (not to) use them

Before/after comparison tables

Integer conversions

Non-locale		Unchanged de_DE		GB226 de_DE		P1892 de_DE	
`{}`	`123456`	`{n}`	`123.456`	`{n}`	`123.456`	`{L}`	`123.456`
`{b}`	`11010010`					`{Lb}`	`11.010.010`
`{B}`	`11010010`					`{LB}`	`11.010.010`
`{c}`	`z`					`{Lc}`	`z`
`{d}`	`123456`	`{n}`	`123.456`	`{n}`	`123.456`	`{Ld}`	`123.456`
`{o}`	`361100`					`{Lo}`	`361.100`
`{x}`	`bc614e`					`{Lx}`	`bc6.14e`
`{X}`	`BC614E`					`{LX}`	`BC6.14E`

Floating-point conversions

Non-locale		Unchanged de_DE		GB226 de_DE		P1892 de_DE	
`{}`	`1234.56789`			`{n}`	`1.234,56789`	`{L}`	`1.234,56789`
`{a}`	`bc61.4e`					`{La}`	`b.c61,4e`
`{A}`	`BC61.4E`					`{LA}`	`B.C61,4E`
`{e}`	`1.23457e+03`					`{Le}`	`1,23457e+03`
`{E}`	`1.23457E+03`					`{LE}`	`1,23457E+03`
`{f}`	`1234.57`					`{Lf}`	`1.234,57`
`{F}`	`1234.57`					`{LF}`	`1.234,57`
`{g}`	`1234.57`	`{n}`	`1.234,57`			`{Lg}`	`1.234,57`

Boolean conversions

Non-locale		Unchanged de_DE		GB226 de_DE		P1892 de_DE	
`{}`	`false`	`{n}`	`0`	`{n}`	*Ill-formed*	`{L}`	`false`
`{s}`	`false`					`{Ls}`	`false`
`{d}`	`0`	`{n}`	`0`			`{Ld}`	`0`
`{c}`	*Null byte*					`{Lc}`	*Null byte*
	Etc.						

Straw polls

SG16, Belfast

Poll	For	Neutral	Against
Accept GB226 as proposed for C++20	4	6	1
Accept GB226 for C++20 modified to removing the ‘n’ specifier?	3	5	2
Accept P1892 to resolve GB226 for C++20 (‘n’ is used as a locale specifier)?	6	2	2
Accept P1892 to resolve GB226 for C++20 modified to use ‘l’ specifier?	4	1	6
Accept P1892 to resolve GB226 for C++20 modified to use ‘L’ specifier?	8	2	1

Proposed wording

Editing notes

All wording is relative to the post-Cologne C++ committee draft [4].

20.20.2.2 Standard format specifiers [format.string.std]

Update ¶1:

Each formatter specializations described in 20.20.4.2 for fundamental and string types interprets *format-spec* as a *std-format-spec*. [*Note:* The format specification can be used to specify such details as field width, alignment, padding, and decimal precision. Some of the formatting options are only supported for arithmetic types. —*end note*] The syntax of format specifications is as follows:

std-format-spec:
  fill-and-align$_{opt}$ *sign*$_{opt}$ `#`$_{opt}$ `0`$_{opt}$ *width*$_{opt}$ *precision*$_{opt}$ `L`$_{opt}$ *type*$_{opt}$
fill-and-align:
  fill$_{opt}$ align
fill:
  any character other than `{` or `}`
align: one of
  `<` `>` `^`
sign: one of
  `+` `−` space
width:
  positive-integer
  `{` *arg-id* `}`
precision:
  `.` *nonnegative-integer*
  `.` `{` *arg-id* `}`
type: one of
  `a A b B c d e E f F g G ~~n~~ o p s x X`

Insert after ¶9:

The `L` option causes the *locale-specific form* to be used for the conversion. This option is only valid for arithmetic types. For integral types, the locale-specific form causes the context’s locale to be used to insert the appropriate digit group separator characters. For floating-point types, the locale-specific form causes the context’s locale to be used to insert the appropriate digit group and decimal radix separator characters. For the textual representation of `bool`, the locale-specific form causes the context’s locale to be used to insert the appropriate string as if obtained with `numpunct::truename` or `numpunct::falsename`.

Update ¶13:

The available integer presentation types for integral types other than `bool` and `charT` are specified in [tab:format.type.int]. [*Example*:

```
string s0 = format("{}", 42);                      // value of s0 is "42"
string s1 = format("{0:b} {0:d} {0:o} {0:x}", 42); // value of s1 is "101010 42 52 2a"
string s2 = format("{0:#x} {0:#X}", 42);           // value of s2 is "0x2a 0X2A"
string s3 = format("{:~~n~~L}", 1234);                  // value of s3 might be "1,234"
                                                   // (depending on the locale)
```

—*end example*]

Update [tab:format.type.int]:

Type	Meaning
b	`to_chars(first, last, value, 2);` the base prefix is `0b`.
B	The same as b, except that the base prefix is `0B`.
c	Copies the character `static_cast<charT>(value)` to the output. Throws `format_error` if value is not in the range of representable values for `charT`.
d	`to_chars(first, last, value).`
o	`to_chars(first, last, value, 8);` the base prefix is `0` if value is nonzero and is empty otherwise.
x	`to_chars(first, last, value, 16);` the base prefix is `0x`.
X	The same as `x`, except that it uses uppercase letters for digits above 9 and the base prefix is `0X`.
~~n~~	~~The same as d, except that it uses the context's locale to insert the appropriate digit group separator characters.~~
none	The same as d. [*Note*: If the formatting argument type is `charT` or `bool`, the default is instead `c` or `s`, respectively. —*end note*]

Update [tab:format.type.char]:

Type	Meaning
none, c	Copies the character to the output
b, B, c, d, o, x, X~~, n~~	As specified in [tab:format.type.int].

Update [tab:format.type.bool]:

Type	Meaning
none, c	Copies textual representation, either `true` or `false`, to the output.
b, B, c, d, o, x, X~~, n~~	As specified in [tab:format.type.int] for the value `static_cast<unsigned char>(value)`.

Update [tab:format.type.float]

Type	Meaning
a	If *precision* is specified, equivalent to `to_chars(first, last, value, chars_format::hex, precision)` where `precision` is the specified formatting precision; equivalent to `to_chars(first, last, value, chars_format::hex)` otherwise.
A	The same as `a`, except that it uses uppercase letters for digits above 9 and P to indicate the exponent.
e	Equivalent to `to_chars(first, last, value, chars_format::scientific, precision)` where `precision` is the specified formatting precision, or 6 if *precision* is not specified.
E	The same as `e`, except that it uses E to indicate exponent.
f, F	Equivalent to `to_chars(first, last, value, chars_format::fixed, precision)` where `precision` is the specified formatting precision, or 6 if *precision* is not specified.
g	Equivalent to `to_chars(first, last, value, chars_format::general, precision)` where `precision` is the specified formatting precision, or 6 if *precision* is not specified.

G	The same as g, except that it uses E to indicate exponent.
~~n~~	~~The same as g, except that it uses the context's locale to insert the appropriate digit group and decimal radix separator characters.~~

none	If *precision* is specified, equivalent to
	`to_chars(first, last, value, chars_format::general, precision)`
	where `precision` is the specified formatting *precision*; equivalent to
	`to_chars(first, last, value)`
	otherwise.

References

[1] V. Zverovich, "P0645R10 Text Formatting," [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0645r10.html.

[2] V. Zverovich, D. Engert and H. E. Hinnant, "P1361R2 Integration of chrono with text formatting," 17 July 2019. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1361r2.pdf.

[3] "GB226 20.20.02.2 Make locale-dependent formats for std::format() congruent with default formatting," 24 October 2019. [Online]. Available: https://github.com/cplusplus/nbballot/issues/223.

[4] R. Smith, "N4830 Committee Draft, Standard for Programming Language C++," 15 August 2019. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/n4830.pdf.