Extended locale-specific presentation specifiers for std::format

Document No.	**P1892 R0**	Date	2019-10-04
Reply To	Peter Brett **pbrett@cadence.com**	Audience:	SG16, LEWG

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].

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

This paper arises from a GB national body comment on the C++ Committee Draft [3]. For reference, the comment is included in 'Appendix: GB national body comment on C++20'. 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 affects access to useful floating-point formats.

This paper proposes a change to '`n`' into a modifier of other specifiers which adds locale-aware formatting. For example:

```
std::locale::global(std::locale("de_DE"));
std::format("{:e} {:ne}", 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 `std::format`.

Design

Easy to explain as an extension of existing syntax

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

```
std::locale::global(std::locale("en_GB"));
std::format("{:} {: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.

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

std::locale grouped(std::locale("C"), new group2);
std::format(grouped, "{:nx}", 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 : std::numpunct<char> {
    string_type do_truename() const { return "oui"; }
    string_type do_falsename() const { return "non"; }
};

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

Presentation of pointers

This paper proposes allowing the '`n`' modifier to be specified only with arithmetic types. Therefore a '`{:np}`' substitution would be ill-formed.

The default behaviour of `std::format` is to display pointers as if they were numbers, but a pointer is not a number.

Prohibiting the '`n`' modifier for pointer values preserves the possibility of defining a locale-dependent pointer presentation in the future, if a compelling use-case arises.

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.

Alternative modifier character

'`l`' may be a better conceptual mapping for "use the context's *locale*" than '`n`' is.

The Committee Draft [3] underspecifies the units of width and precision for string arguments, possibly creating an ambiguity in variable-width string encodings. The use of a '`l`' modifier has been proposed to cause the locale's text encoding to be used when computing the display width of a string [4]. These semantics could be added to a '`n`' or '`l`' as a locale-specific presentation modifier.

One of the attractions of `std::format` is its similarity to `str.format()` in the Python standard library, which uses '`n`' to request locale-specific presentation [5]. Retaining '`n`' permits many Python format strings to be used unchanged with `std::format`.

Post C++20 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

Proposed wording

Editing notes

All wording is relative to the post-Cologne C++ committee draft [3].

20.20.2.2 Standard format specifiers [format.string.std]

Update ¶1:

[…] The syntax of format specifications is as follows:

std-format-spec:
fill-and-align$_{opt}$ *sign*$_{opt}$ #$_{opt}$ 0$_{opt}$ *width*$_{opt}$ *precision*$_{opt}$ n$_{opt}$ *type*$_{opt}$
[…]
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 n option the causes the *locale-specific form* to be used for the conversion. This option is only valid for arithmetic types.

Update ¶13:

The available integer presentation types for integral types other than `bool` and `charT` are specified in [tab:format.type.int]. If the *locale-specific form* is requested, the context's locale is used to insert the appropriate digit group separator characters. [*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}", 1234);                // value of s3 might be "1,234"
                                                 // (depending on the locale)
```

—end example]

Update [tab:format.type.int]:

Type	Meaning
[…]	[…]
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 ¶15:

The available bool presentation types are specified in [tab:format.type.bool]. If the *locale-specific form* is requested for the textual representation, the context's locale is used to insert the appropriate string as if obtained with `numpunct::truename` or `numpunct::falsename`.

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 ¶16:

[…] [*Note*: In either case, a sign is included if indicated by the *sign* option. —*end note*] If the *locale-specific form* is requested, the context's locale is used to insert the appropriate digit group and decimal radix separator characters.

Update [tab:format.type.float]

Type	Meaning
[…]	[…]
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," 16 July 2019. [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] 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.

[4] V. Zverovich and Z. Laine, “D1868R0 width: clarifying units of width and precision in std::format,” 29 September 2019. [Online]. Available: https://fmt.dev/D1868R0.html. [Accessed 04 October 2019].

[5] “Python 3.7.5rc1 Format Specification Mini-Language,” [Online]. Available: https://docs.python.org/3/library/string.html#formatspec. [Accessed 04 October 2019].

Appendix: GB national body comment on C++20

Comments

Make locale-dependent formats for `std::format` congruent with default formatting

The design of `format` prefers "locale-independent" formatting options for performance reasons. It provides very limited support for locale-dependent formatting via the ‘`n`’ specifier.

It's particularly problematic that the ‘`n`’ specifier for floating point numbers is specifically limited to the `chars_format::general` presentation. It would be very useful to have access to `chars_format::scientific` and `chars_format::fixed` formatting with locale-dependent presentation.

Adding these features to `format` at this stage would require significant wording changes that are too large to contain in a comment. However, one approach that could be taken in the future would be to make ‘`n`’ be an additional suffix that could be added to format specifiers, rather than being a lone format specifier. This would enable locale-dependent formatting of any of the conversions of any of the arithmetic types.

In order to keep the design space open for making this change in a future version of the standard, it would be ideal for ‘`n`’ conversions to always be congruent with the default conversion. It provides an intuitive semantic: ‘`n`’ is the same as "no specifier", but with locale-dependent presentation.

The integer and `charT` presentation types currently specify ‘`n`’ conversions that are congruent with the default conversion.

The bool and floating-point presentation types have ‘`n`’ conversions that are not congruent with the default conversion.

For C++20:

Remove the ‘`n`’ conversion for `bool`.

```
auto s format("{:n}", 1);
// Committee Draft: s contains "1"
// Proposed:        ill-formed format string
```

Make the ‘`n`’ conversion for floating-point match the default conversion, i.e. dependent on whether a precision is specified.

```
auto s format("{:n} {:2n}", 12.345678, 12.345678);
// Committee Draft: s contains "12,3456 12,34"
// Proposed:        s contains "12,345678 12,34"
```

These changes are the minimum necessary to allow enhanced support for locale-dependent formatting in the standard library to be added in a backwards-compatible way in a future edition of C++.

Proposed change

In **[tab:format.type.bool]**: Remove `n`.

In **[tab:format.type.float]**: Replace the "Meaning" of the '`n`' specifier with:

> 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. The context's locale is used to insert the appropriate digit group and decimal radix separator characters.