Formatting of Negative Zero

Document number:	**P1496R1**
Date:	2019-10-06
Reference Document:	N5410
Audience:	Library Evolution Working Group, Library Working Group

Reply to:	**Alan Talbot**	cpp@alantalbot.com
	Jorg Brown	jorg.brown@gmail.com

History

During the Library Evolution Working Group review of Victor Zverovich's Text Formatting proposal (P0645R7) at the Kona 2019 meeting, it was suggested that the user be given a choice of whether or not to display negative zero when formatting floating point numbers. We wrote and presented the first version of this paper (P1496R0) during the meeting to propose that change. The vote on the issue was a palindromic paragon of equivocation:

SF	F	N	A	SA
1	5	2	5	1

This was quite appropriately deemed no consensus for change, but we believe that our paper did not correctly describe the problem. (Alan takes full credit for this oversight.) We realized after the vote that the examples of alternative solutions we presented did not actually address the real issue. This revision will hopefully provide clarification.

Status Quo

Floating point format in the new text formatting facilities in C++20 (20.20 [format]) is expressed in terms of `to_chars` which in turn is expressed in terms of `printf`. The C standard does not appear to explicitly specify whether `printf` should add a minus sign to negative zeros in the formatted output. Several popular C++ Standard Library implementations do add minus signs to negative zeros, and there is no way to prevent this.

The Problem

The concern here is *not* about handling the IEEE 754 floating point negative zero value. The number of cases where that particular value is involved are assumed to be vanishingly small, and dealing with that case is trivially easy. The problem arises because of negative values *near* zero which are rounded *up to* zero by the requested formatting precision.

In almost all applications floating point values are displayed at some appropriate precision which depends on the domain and is often controlled by the user. Negative zeros appear all the time, especially in cases where zero is a common calculated answer, because floating point calculations which would result in zero mathematically often end up being very small numbers on either side of zero.

Unfortunately there is no easy way to catch negative zero (unless you write your own text formatter) because you don't know if you will have a zero until the rounding is done. Your options boil down to:

A. Round the number *first*, detect -0.0 and change the sign, *then* format the number using `std::format` (in C++20) or lower level utilities.
B. Parse the text representation *after* formatting to detect the negative zero character pattern, then remove the minus sign.

Neither of these solutions is efficient or easy to get right, and both will inevitably result in writing home-grown wrappers for the text formatting operations, thus partially defeating the purpose of the new standard text formatting facilities.

We believe that this should be handled internally by `std::format`, where it can be done with maximum efficiency and perfect reliability. The formatting algorithm can detect this trivially, and removing the minus sign once the string is formed is an $O(1)$ operation.

Proposed Solution

We propose adding a new format specifier 'z' to suppress the output of the minus sign for negative zeros. With the 'z' option a negative zero *after rounding* is formatted as a (positive) zero.

Currently supported by N5410	
`format("{0:.0} {0:+.0} {0:-.0} {0: .0}", 0.1)`	`0 +0 0  0`
`format("{0:.0} {0:+.0} {0:-.0} {0: .0}", -0.1)`	`-0 -0 -0 -0`
Additionally supported with this proposal	
`format("{0:z.0} {0:+z.0} {0:-z.0} {0: z.0}", 0.1)`	`0 +0 0  0`
`format("{0:z.0} {0:+z.0} {0:-z.0} {0: z.0}", -0.1)`	`0 +0 0  0`

Who Benefits?

In a use case Alan is familiar with, an engineering application generates dozens of different tabular reports, each of which has between tens and hundreds of columns, most of which display floating point numbers rounded to various domain-specific precisions. These numbers are frequently calculated values near zero, leading to lots of negative zeros. The customers who receive these reports do not care about or understand negative zeros, and report them as a must-fix issue. The problem was addressed in this software using option A above, which has proven to be a source of subtle bugs.

Who	What
Everybody (millions)	Most applications round floating point numbers and do not want negative zeros.
Power user (10,000)	Industrial engineering and financial applications also round floating point numbers and do not want negative zeros.
Expert (1,000)	Certain specialized mathematical and scientific applications care about negative zeros. The default behavior is unchanged, so these use cases are not affected by this proposal.

Default Behavior

As mentioned above, whether `std::format` shows negative zero is inherited from `printf`, and therefore is implementation-defined. This proposal does not change that.

However, the fact that all but a very few specialized applications are unlikely to want negative zeros strongly suggests that suppressing them should be the default behavior. Making the default be to *suppress* negative zero and adding a new option to *show* negative zero would have two benefits over this proposal: it would make the default unsurprising to most users, and it would provide a way to *force* negative zero to be shown. This would directly support the expert use cases where negative zero *is* desired. These cases are not supported today or under this proposal if the library vendor chooses not to show negative zero.

(Yet another solution would be to leave the default undefined and provide two new options, hide and show, but that seems unnecessarily elaborate.)

Proposed Wording

20.20.2.2 Standard format specifiers [format.string.std]

¶ 5:

The `sign` option applies to floating-point infinity and NaN. [*Example:*

```
double inf = numeric_limits<double>::infinity();
double nan = numeric_limits<double>::quiet_NaN();
string s0 = format("{0:},{0:+},{0:-},{0: }", 1);    // value of s0 is "1,+1,1, 1"
string s1 = format("{0:},{0:+},{0:-},{0: }", -1);   // value of s1 is "-1,-1,-1,-1"
string s2 = format("{0:},{0:+},{0:-},{0: }", inf); // value of s2 is "inf,+inf,inf, inf"
string s3 = format("{0:},{0:+},{0:-},{0: }", nan); // value of s3 is "nan,+nan,nan, nan"
string s4 = format("{0:z.0},{0:+z.0},{0:-z.0},{0: z.0}", -0.1);
   // value of s4 is "0,+0,0, 0"
```

— *end example*]

Table 59: Meaning of *sign* options [tab:format.sign]

Option	Meaning
+	Indicates that a sign should be used for both non-negative and negative numbers.
-	Indicates that a sign should be used only for negative numbers (this is the default behavior).
space	Indicates that a leading space should be used for non-negative numbers, and a minus sign for negative numbers.
`z`	Indicates that a sign should not be used for negative numbers that display as zero (after rounding to the formatting precision).

[Editorial issue: In N5410 Tables 58 and 59 are interleaved with the code in the Example in ¶ 5.]