Formatting of Negative Zero

Document number: **P1496R2**
Date: 2020-01-12
Reference Document: N4842
Audience: Library Evolution Working Group, Library Working Group
Reply to: **Alan Talbot** cpp@alantalbot.com

R2 History

The R1 version of this paper was reviewed at the Belfast 2019 meeting. There was initially some concern that no new information had been provided and reopening the issue was out of order, but ultimately LEWG voted with moderate consensus that there was enough new information to warrant another look:

SF	F	N	A	SA
4	7	3	3	1

I presented the R1 version and LEWG voted with strong consensus to advance the proposal to LWG for inclusion in C++20:

SF	F	N	A	SA
5	10	2	2	0

After the vote I became aware that there remained some concerns regarding the proposal. I have attempted to address these concerns in this R2 revision. I corrected or removed some sections, and I added sections which speak to the concerns as I understand them. (See **Concerns** below.)

R1 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. Victor, Jorg Brown and I wrote and presented the first version of this paper (P1496R0) during the meeting to propose that change. The vote on the issue was mixed and did not represent a consensus for change, but I later realized that our paper did not correctly describe the problem. (I take full responsibility for this oversight.) R1 provided some clarification.

Status Quo

Floating point formatting in the new text formatting facilities in C++20 (20.20 [format]) is defined in terms of `to_chars` which in turn is defined in terms of `printf`. The C standard states that `printf` shall add a minus sign to all negative zeros in the formatted output. There is no way provided 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 special 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 magnitude numbers on either side of zero.

Unfortunately, there is no easy way to catch numbers rounded to negative zero (unless you write your own text formatter) because you don't know if you will have a zero until *after* the rounding is done by the formatting process. 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.
C. Write your own text formatter.

None of these solutions is easy to get right, and the first two are inefficient and will inevitably result in writing home-grown wrappers for the text formatting operations, thus partially defeating the purpose of standardizing text formatting.

Proposed Solution

I believe that showing or suppressing negative zero 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.

I propose adding a new format specifier 'z' to suppress the output of the minus sign for negative zeros. With the 'z' option, if a negative zero results after rounding, it is formatted as a (positive) zero. (Using 'z' as the syntax is just a suggestion. LEWG or LWG may wish to change it.)

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?

The engineering application I work on 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 invariably report them as a bug. The problem was addressed in this particular application using option A above, which has proven to be a source of subtle bugs.

Who	What
Everybody (millions)	Most applications round floating point numbers to reasonable precisions and do not want negative zeros.
Power user (10,000)	Industrial engineering and financial applications also round floating point numbers and usually 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. (See **Performance** below.)

Concerns

Default Behavior

As mentioned above, `std::format` defaults to showing minus signs on zeros. This proposal does not change that.

Performance

It is my understanding that this option can be implemented in a way which has close to zero overhead when it is not present, and is both *O*(1) and very fast when it is present. Given the nature of text formatting, I don't think that the cost of (for example) a single branch would ever be a significant factor, and the additional code volume should be small.

Implementation

I have discussed this feature with implementers, and I have been assured that it is both reasonable to design and (as I say above) sufficiently fast not to be a concern. However, this feature has **not** been implemented as far as I know.

Interaction with <chrono>

The chrono library specializes `std::formatter` on several types, but these specializations do not include a **sign** option so this proposal has no effect on them. It is possible that a **sign** option might be provided for durations by some future chrono proposal, but that is unrelated to and outside the scope of this proposal. If such a change were made, the enhancement provided herein would not interfere, and in fact would be of benefit.

Locales

A suggestion was made that this option could be provided as part of a locale. The main argument against this (and I believe it is a compelling one) is that all of the other minus sign formatting is handled by the **sign** option of the format string. It would be inconsistent and confusing to put the negative zero option somewhere else.

Target

This proposal does not have or create dependencies—it is a pure addition to the formatting library. It is therefore not necessary to put this into C++20. However, since the formatting library is new in C++20, if we want to do it I believe we should do it now. It will be easier for implementers to design their code to support it in the first place than to add it later, and in general I think that we should prefer shipping complete components whenever possible.

Proposed Wording

20.20.2.2 Standard format specifiers **[format.string.std]**

¶ 1:

sign: one of

+ - space

optionally followed by

z

¶ 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 58: 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 N4842 Tables 57 and 58 are interleaved with the code in the Example in ¶ 5.]