Fix the default floating-point representation in std::format

P3505R2

Victor Zverovich (victor.zverovich@gmail.com)

Junekey Jeon (j6jeon@ucsd.edu)

Introduction

- Floating-point formatting in `std::format` was defined in terms of `std::to_chars` to simplify specification.
- While beneficial overall, this introduced a small but undesirable change compared to the original design and the reference implementation in {fmt} resulting in
  - Surprising behavior to users
  - Performance regression
  - Inconsistency with other languages that have similar facilities
- The current paper proposes fixing it.

Background

Many programming languages converged on a similar representation of FP numbers based on work by Steele and White in 70s and formulated in their 1990 paper, How to Print Floating-Point Numbers Accurately:

- No information is lost; the original fraction can be recovered from the output by rounding.
- No "garbage digits" are produced.
- The output is correctly rounded.
- It is never necessary to propagate carries on rounding.

The last bullet point is more of an optimization for retro computers and is less relevant on modern systems but the other ones are important.

Background

Many programming languages converged on a similar representation of FP numbers based on work by Steele and White in 70s and formulated in their 1990 paper, How to Print Floating-Point Numbers Accurately:

- No information is lost; the original fraction can be recovered from the output by rounding.
- **No "garbage digits" are produced.**
- The output is correctly rounded.
- It is never necessary to propagate carries on rounding.

The last bullet point is more of an optimization for retro computers and is less relevant on modern systems but the other ones are important.

No “garbage digits”

- Don’t produce more decimal digits (in the significand) than necessary to satisfy the other requirements, most importantly the round-trip guarantee.
- For example, `0.1` should be formatted as `0.1` and not `0.10000000000000001` even though they produce the same value when read back into an IEEE 754 `double`.
- This was the original shortness criteria adopted by multiple languages and {fmt}.

Shortness

- Shortness in Steele and White (1990) refers to **the number of digits in the significand** and **does not include exponent or decimal point**.
- Once the shortest decimal significand and the corresponding exponent are known the choice between fixed and exponential representation is normally based on the value range.

Shortness

- Python and Rust switch to exponential if decimal exponent >= 16 for `double`:

```
>>> 1234567890123456.0
1234567890123456.0

>>> 12345678901234567.0
1.2345678901234568e+16
```

- Swift switches to exponential notation for numbers greater or equal to 2^{53}, another reasonable choice although power of 2 might be less intuitive.

- Similarly, languages normally switch from fixed to exponential notation when the absolute value is smaller than some small decimal power of 10, usually 10^{-3} (Java) or 10^{-4} (Python, Rust, Swift).

Problems

- When `std::format` was proposed for standardization, FP formatting was defined in terms of `std::to_chars` to simplify specification with the assumption that the latter follows the industry practice described earlier.
- Great for explicit format specifiers such as `e` but, as it turned out recently, introduced an undesirable change to the default format compared to the original design in {fmt}.
- This problem is that “shortness" is defined in terms of the number of characters in the output which is different from the "shortness" of decimal significand normally used both in the literature and in the industry.

Problems

- The exponent range is much easier to reason about. For example, `100000.0` and `120000.0` are printed in the same format:

```
>>> 100000.0
100000.0
>>> 120000.0
120000.0
```

- However, if we consider the output size the two similar numbers are now printed completely differently:

```
auto s1 = std::format("{}", 100000.0); // s1 == "1e+05"
auto s2 = std::format("{}", 120000.0); // s2 == "120000"
```

- It seems surprising and undesirable.

Problems

- Adding to confusion, the output `1e+05` is not even the shortest possible number of characters, because + and the leading zero in the exponent are redundant but required, according to [charconv.to.chars]:

  > `value` is converted to a string in the style of `printf` in the `"C"` locale.

- and the exponential format is defined as follows by the C standard ([N3220]):

  > A `double` argument representing a floating-point number is converted in the style *[−]d.ddd e±dd* ...

Problems

- More importantly, the current representation violates the original shortness requirement from Steele and White:

```
auto s = std::format("{}", 1234567890123456700000.0);
// s == "1234567890123456774144"
```

- The last 5 digits, `74144`, are "garbage digits" that almost no modern formatting facilities produce by default. For example, Python avoids it by switching to the exponential format as expected:

```
>>> 1234567890123456700000.0
1.2345678901234568e+21
```

- This gives a false sense of accuracy to users.

Problems

- Producing “garbage digits” also has negative performance implications.
- Modern algorithms such as Dragonbox and Ryū are based on Steele and White criteria and require nontrivial logic to produce additional digits.
- This will likely hold for future algorithms because producing more digits than than the actual precision implied by the data type inherently requires more work.

Performance

```
double normal_input  = 1234567890123456700000.0;

// Output (current): "1234567890123456774144"
// Output (desired): "1.2345678901234568e+21"
double garbage_input = 123456789012345670000.0;

void normal(benchmark::State& state) {
  for (auto s : state) {
    auto result = std::format("{}", normal_input);
    benchmark::DoNotOptimize(result);
  }
}
BENCHMARK(normal);

void garbage(benchmark::State& state) {
  for (auto s : state) {
    auto result = std::format("{}", garbage_input);
    benchmark::DoNotOptimize(result);
  }
}
BENCHMARK(garbage);
```

Performance

Clang and libc++ on macOS:

```
-----------------------------------------------------
Benchmark             Time             CPU   Iterations
-----------------------------------------------------
normal             77.5 ns         77.5 ns      9040424
garbage            91.4 ns         91.4 ns      7675186
```

GCC and libstdc++ on GNU/Linux:

```
-----------------------------------------------------
Benchmark             Time             CPU   Iterations
-----------------------------------------------------
normal             73.1 ns         73.1 ns      9441284
garbage            90.6 ns         90.6 ns      7360351
```

MSVC and Microsoft STL on Windows:

```
-----------------------------------------------------
Benchmark             Time             CPU   Iterations
-----------------------------------------------------
normal              144 ns          143 ns      4480000
garbage             166 ns          165 ns      4072727
```

Performance

- Although the output has the same size, producing "garbage digits" makes `std::format` 15-24% slower on these inputs.
- If we exclude string construction time, the difference is even more profound, up to 50% slower:

```
garbage(benchmark::State&):
241.00 ms ... std::__1::to_chars_result
std::__1::_Floating_to_chars[abi:ne180100]<...>(c
har*, char*, double, std::__1::chars_format, int)

normal(benchmark::State&):
159.00 ms ... std::__1::to_chars_result
std::__1::_Floating_to_chars[abi:ne180100]<...>(c
har*, char*, double, std::__1::chars_format, int)
```

Locale

- Locale makes the situation even more confusing to users. Consider the following example:

```
std::locale::global(std::locale("en_US.UTF-8"));
auto s = std::format("{:L}", 1200000.0);
// s == “1,200,000"
```

- Here `s` is `"1,200,000"` even though `"1.2e+06"` would be shorter.

Proposal

Switch the default floating-point representation in `std::format` to use exponent range, fixing the issues described above.

Let `T` be `decltype(value)`. Let `fmt` be `chars_format::fixed` if the absolute value of `value` is in the range [*m*, *n*), where *m* is the nearest representable as `T` value of 10^{-4} and *n* is the nearest representable as `T` value of `std::numeric_limits<T>::radix`$^{\texttt{std::numeric\_limits<T>::digits} + 1}$ rounded down to the nearest power of 10, `chars_format::scientific` otherwise.

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, fmt)
```

otherwise.

Proposal

- Consistent, easy to reason about output format:

Code	Before	After
`std::format("{}", 100000.0)`	`"1e+05"`	`"100000"`
`std::format("{}", 120000.0)`	`"120000"`	`"120000"`

Proposal

- No "garbage digits":

Code	`std::format("{}", 12345678901234567000000.0)`
Before	`"12345678901234567741440"`
After	`"1.2345678901234568e+22"`

Proposal

- Consistent localized output (assuming `en_US.UTF-8` locale):

Code	Before	After
`std::format("{:L}", 1000000.0)`	`"1e+06"`	`"1,000,000"`
`std::format("{:L}", 1200000.0)`	`"1,200,000"`	`"1,200,000"`

Proposal

- For comparison, here are the results of running the same benchmark with `std::format` replaced with `fmt::format` which doesn’t produce "garbage digits”:

```
---------------------------------------------------------
Benchmark              Time             CPU   Iterations
---------------------------------------------------------
fmt_normal          53.0 ns         53.0 ns     13428484
fmt_garbage         53.4 ns         53.4 ns     13032712
```

- The time is nearly identical between the two cases demonstrating that the performance gap can be eliminated if this paper is accepted.

Implementation and usage experience

- The current proposal is based on the existing implementation in {fmt} which has been available and widely used for over 6 years.
- Similar logic based on the value range rather than the output size is implemented in Python, Java, JavaScript, Rust and Swift.

Impact on existing code

- This is technically a breaking change for users who rely on the exact output that is being changed.
- However, the change doesn't affect ABI or round trip guarantees.
- Reliance on the exact representation of floating-point numbers is usually discouraged so the impact of this change is likely moderate.
- In the past we had experience with changing the output format in {fmt}, `std::format` and `std::to_string` with much larger impact.

Option 2: fix to_chars

Switch the default floating-point representation in `std::to_chars` to use exponent range, fixing the issues described above.

Let `T` be *floating-point-type*. The functions that take a floating-point `value` but not a `chars_format` parameter determine the conversion specifier for `printf` as follows: The conversion specifier is `f` if the absolute value of `value` is in the range [*m*, *n*), where *m* is the nearest representable as `T` value of 10^{-4} and *n* is the nearest representable as `T` value of $\texttt{std::numeric\_limits<T>::radix}^{\texttt{std::numeric\_limits<T>::digits} + 1}$ rounded down to the nearest power of 10, `e` otherwise.

Wording

https://isocpp.org/files/papers/D3505R2.html#wording