Formatting floating-point numbers

Victor Zverovich
About me

• VIK-ter ZVE-roh-vich
• Work at Facebook on the Thrift RPC & serialization framework
• Author of the {fmt} library and C++20 std::format
• Expert in negative zero
• https://github.com/vitaut
• https://twitter.com/vzverovich
Faster float format #147

newnon opened this issue on Apr 8, 2015 · 23 comments

newnon commented on Apr 8, 2015

Have you seen this project? They have fast float to string conversions
https://code.google.com/p/stringencoders/

vitaut commented on Mar 24

Added {fmt} to dtoa_benchmark and here are some results: http://fmtlib.net/unknown_mac64_clang10.0.html. TL;DR: {fmt} is ~13x faster than iostreams, ~10x faster than sprintf and roughly as fast as double_conversion (unsurprisingly because both implement the same algorithm). The implementation is not particularly optimized yet, so might be able to squeeze 20-30% more.

https://github.com/fmtlib/fmt/issues/147
"By the end of the talk you will be able to convert binary floating-point to decimal in your mind or you will get your money back!"
A bit of history
The origin

- Floating point arithmetic was "casually" introduced in 1913 paper "Essays on Automatics" by Leonardo Torres y Quevedo, a Spanish civil engineer and mathematician.

- Included in his 1914 electro-mechanical version of Charles Babbage's Analytical Engine.
In early computers

- 1938 Z1 by Konrad Zuse used 24-bit binary floating point
- 1941 relay-based Z3 had +/- infinity and exceptions (sort of)
- 1954 mass-produced IBM 704 introduced biased exponent
FORTRAN had formatted floating-point I/O in 1950s (same time as comments were invented!):

```
WRITE OUTPUT TAPE 6, 601, IA, IB, IC, AREA
601 FORMAT (4H A= ,I5,5H  B= ,I5,5H  C= ,I5, 
& 8H AREA= ,F10.2, 13H SQUARE UNITS)
```
FP formatting in C


```c
/* print Fahrenheit-Celsius table
   for f = 0, 20, ..., 300 */
main()
{
    int lower, upper, step;
    float fahr, celsius;

    lower = 0;    /* lower limit of temperature table */
    upper = 300;  /* upper limit */
    step = 20;    /* step size */

    fahr = lower;
    while (fahr <= upper) {
        celsius = (5.0/9.0) * (fahr-32.0);
        printf("%4.0f %6.1f\n", fahr, celsius);
        fahr = fahr + step;
    }
}
```

Still compiles in 2019: https://godbolt.org/z/KsOzjr
Solved problem?

- Floating point has been around for a while
- Programmers have been able to format and output FP numbers since 1950s
- Solved problem
- We all go home now
Solved problem?

- Floating point has been around for a while
- Programmers have been able to format and output FP numbers since 1950s
- Solved problem
- We all go home now
- Not so fast
Solved problem?
Solved problem?
Meanwhile in 2019

- Neither `stdio/printf` nor iostreams can give you the shortest decimal representation with round-trip guarantees

- Performance has much to be desired, esp. with iostreams

- Relying on global locale leads to subtle bugs, e.g. JSON-related errors reported by French but not English users
Meanwhile in 2019

- Neither `stdio/printf` nor `iostreams` can give you the shortest decimal representation with round-trip guarantees

- Performance has much to be desired, esp. with `iostreams`

- Relying on global locale leads to subtle bugs, e.g. JSON-related errors reported by French but not English users 😔
Is floating point math broken?

Consider the following code:

```
0.1 + 0.2 == 0.3  ->  false
```

```
0.1 + 0.2  ->  0.30000000000000004
```

Why do these inaccuracies happen?
Is floating point math broken?

Consider the following code:

\[0.1 + 0.2 \neq 0.3 \rightarrow \text{false}\]

\[0.1 + 0.2 \rightarrow 0.30000000000000004\]

Why do these inaccuracies happen?
Floating-point math is not broken, but can be tricky

Formatting defaults are broken or at least suboptimal in C & C++ (loose precision):

```cpp
std::cout << (0.1 + 0.2) << " == " << 0.3 << " is "
<< std::boolalpha << (0.1 + 0.2 == 0.3) << "\n";
```

prints "0.3 == 0.3 is false"

The issue is not specific to C++ but some languages have better defaults: https://0.30000000000000004.com/
Desired properties

Steele & White (1990):

1. No information loss
2. Shortest output
3. Correct rounding
4. Left to right generation - irrelevant with buffering
No information loss

Round trip guarantee: parsing the output gives the original value.

Most libraries/functions lack this property unless you explicitly specify big enough precision: C stdio, C++ iostreams & to_string, Python's str.format until version 3, etc.

double a = 1.0 / 3.0;
char buf[20];
sprintf(buf, "%g", a);
double b = atof(buf);
assert(a == b);

// fails:
// a == 0.3333333333333333
// b == 0.333333

double a = 1.0 / 3.0;
auto s = fmt::format("{0}", a);
double b = atof(s.c_str());
assert(a == b);

// succeeds:
// a == 0.3333333333333333
// b == 0.3333333333333333
How much is enough?

- "17 digits ought to be enough for anyone" — some famous person (paraphrased)

- *In-and-out conversions*,
  David W. Matula (1968):

Conversions from base $B$ round-trip through base $\nu$ when $B^n < \nu^{m-1}$, where $n$ is the number of base $B$ digits, and $m$ is the number of base $\nu$ digits.

$$\lceil \log_{10}(2^{53}) + 1 \rceil = 17$$
Shortest output

The number of digits in the output is as small as possible.

It is easy to satisfy the round-trip property by printing unnecessary "garbage" digits (provided correct rounding):

```cpp
sprintf("%.17g", 0.1); // prints "0.1000000000000001"
```

```cpp
fmt::print("{:.17g}\n", 0.1); // prints "0.1"
```
Correct rounding

• The output is as close to the input as possible.

• Most implementations have this, but MSVC/CRT is buggy as of 2015 (!) and possibly later (both from and to decimal):


• Had to disable some floating-point tests on MSVC due to broken rounding in printf and iostreams
How does it work?
IEEE 754

Binary floating point bit layout:

\[
v = \begin{cases} 
(-1)^{\text{sign}} 1.\text{fraction} \times 2^{(\text{exponent}-\text{bias})} & \text{if } 0 < \text{exponent} < 1...1_2 \\
(-1)^{\text{sign}} 0.\text{fraction} \times 2^{(1-\text{bias})} & \text{if } \text{exponent} = 0 \\
(-1)^{\text{sign}} \text{Infinity} & \text{if } \text{exponent} = 1...1_2, \text{fraction} = 0 \\
\text{NaN} & \text{if } \text{exponent} = 1...1_2, \text{fraction} \neq 0 
\end{cases}
\]
IEEE 754

Double-precision binary floating point bit layout:

\[ v = \begin{cases} 
(-1)^{\text{sign}} 1.\text{fraction} \times 2^{(\text{exponent} - \text{bias})} & \text{if } 0 < \text{exponent} < 1\ldots1_2 \\
(-1)^{\text{sign}} 0.\text{fraction} \times 2^{(1 - \text{bias})} & \text{if } \text{exponent} = 0 \\
(-1)^{\text{sign}} \text{Infinity} & \text{if } \text{exponent} = 1\ldots1_2, \text{fraction} = 0 \\
\text{NaN} & \text{if } \text{exponent} = 1\ldots1_2, \text{fraction} \neq 0 
\end{cases} \]

where \( \text{bias} = 1023 \)
**Example**

π approximation as double (M_PI):

\[ v = (-1)^0 \cdot 1.10010010000111111011010100010001011011000011000 \times 2^{10000000000 - 1023_{10}} = 1.100100100001111110110101000100010110100011 \times 2 = 11.001001000011111101101010001000100010110100011 \]
Floating point formatting is ...

Floating point formatting is easy*
Floating point formatting is easy*

*conceptually (terms and conditions apply)
Table 6: Procedure Fixup

procedure Fixup;
begin
  if f = shiftl(1, p - 1) then
    comment Account for unequal gaps.
    M* → shiftl(M* + 1);
    R → shiftl(R, 1);
    S → shiftl(S, 1);
    fl;
    k → 0;
  loop while R ≤ S:
    k → k + 1;
    R → R × B;
    M* → M* × B;
    M* → M* × B;
  repeat;
  comment Perform any necessary adjustment of M* and M* to take into account the
  formatting requirements.
  case CutoffMode of
    "normal": CutoffPlace ← k;
    "absolute": CutoffAdjust;
    "relative": CutoffPlace ← k + CutoffPlace;
    CutoffAdjust;
  endcase;
  while (2 × k) + M* ≥ 2 × S:
    S → S × B;
    k → k + 1;
  repeat;
end;

Table 7: Procedure fill

procedure fill(k, c);
  comment Send k copies of the character c to the USER process. No characters are sent if k = 0.
  for i from 1 to k do USER (c); od;

Table 8: Procedure CutoffAdjust

procedure CutoffAdjust;
begin
  a ← CutoffPlace − k;
  y → S;
  cases
  a ≥ 0: for j = 1 to a do y ← y × B;
  a ≤ 0: for j = 1 to −a do y ← [y/B];
  endcase;
  assert y = [S × B*]
  M* → max(y, M*);
  M* → max(y, M*);
  if M* = y then RoundUpFlag ← true fi;
end;

Table 9: Procedure DigitChar

procedure DigitChar(U); cases U of
  comment A digit that is −1 is treated as a zero
  (one that is not significant). Here we print a
  blank for it; fixed Fortran formats might prefer
  a zero.
  −1: USERI (" ");
  0: USERI ("0");
  1: USERI ("1");
  2: USERI ("2");
  3: USERI ("3");
  4: USERI ("4");
  5: USERI ("5");
  6: USERI ("6");
  7: USERI ("7");
  8: USERI ("8");
  9: USERI ("9");
  10: USERI ("A");
  11: USERI ("B");
  12: USERI ("C");
  13: USERI ("D");
  14: USERI ("E");
  15: USERI ("F");
  endcase;
end;

Table 10: Formatting process for fixed-format output

procedure Fixed-Format;
begin
  USERI (b, e, f, p, B);
  GENERATEI (b, e, f, p, B, "normal", 0);
  GENERATEI (U, k);
  if k = 0 then
    USERI ("0");
    fi;
  else fill(−k, "0");
  fi;
  loop
    DigitChar(U);
  end;
  fi;
  if k = 0 then USERI (" "); fi;
  fi;
  GENERATEI (U, k);
  while U ≠ −1 or k ≥ −1 do;
    repeat;
    fi;
  end;

Table 11: Formatting process for fixed-format output

procedure Fixed-Format;
begin
  USERI (b, e, f, p, B, w, d);
  assert 2 ≥ 0 ∧ w ≥ max(d + 1, 2)
  c ← w − d − 1;
  GENERATEI (b, e, f, p, B, "absolute", −d);
  GENERATEI (U, k);
  if k = 0 then
    if k < 0 then
      if c > 0 then fill(c − 1, " "); USERI ("0");
    fi;
    USERI (" ");
    fill(min(c − k − 1, " "));
  else fill(c − k − 1, " ");
  fi;
  loop
    while k ≥ −d do;
      DigitChar(U);
    end;
  fi;
  if k = 0 then USERI (" ");
  fi;
  repeat;
  else fill(w, " ");
  fi;
  fi;
  USERI (" ");
  end;
Input

Input FP number $v > 0$
Neighbors

Predecessor: previous representable value

Successor: next representable value
Neighbours

Predecessor: previous representable value

Successor: next representable value
Values half way between $v$ and its neighbours.
Boundaries

Numbers in $(M^-, M^+)$ round to $v$
Find power of 10

Find largest $k$ such that $V_k = \text{round}(v / 10^k)10^k$ is in $[M^-, M^+]$
Find largest $k$ such that $V_k = \text{round}(v / 10^k)10^k$ is in $[M-, M+]$

\[\text{result} = \text{format}(\"{e}\", \text{round}(v / 10^k), k)\]

\text{round}(v / 10^k) and $k$ are ints
Example

Input: \( v = 1.23e45 \)

\[
v^- = 12299999999999999815358543982490949384520335360 = \text{0b11011100100111101101010010000011111011101000010100011} * 2^{97}
\]

\[
M^- = 12299999999999999894586706496755286978064285696 = \text{0b110111001001111011010100100000111101111010000101000111} * 2^{96}
\]

\[
v = 1229999999999999973814869011019624571608236032 = \text{0b1101110010011110110101001000001111011110100010100100} * 2^{97}
\]

\[
M^+ = 12300000000000000000053043031525283962165152186368 = \text{0b11011100100111101101010010000011110111101000010100100} * 2^{96}
\]

\[
v^+ = 123000000000000013227194039548299758696136704 = \text{0b11011100100111101101010010000011110111101000010100101} * 2^{97}
\]
Example

$v = 122999999999999997...$
Neighbours

Predecessor: 122999999999999981...

Successor: 123000000000000013...
Boundaries

\[
\frac{(v + v^-)}{2} = 122999999999999989...
\]

\[
\frac{(v + v^+)}{2} = 123000000000000005...
\]
Find power of 10

\[ V_1 = 1 \times 10^{45} \]

\[ v = 122999999999999997... \]

\[ 123000000000000005... \]
Find power of 10

$V_2 = 12e44$  
$M^{-}$  
$\nu$  
$M^{+}$

$v = 122999999999999997...$

$122999999999999989...$

$123000000000000005...$
Find power of 10

\[ V_3 = 123e43 \]

\[ m^- \quad v \quad m^+ \]

\[ 122999999999999989\ldots \]

\[ v = 122999999999999997\ldots \]

\[ 123000000000000005\ldots \]
Computations should be exact or done with high precision.
Exponent

- Full exponent range for IEEE double: $10^{-324} - 10^{308}$

- In general requires multiple precision arithmetic

- glibc pulls in a GNU multiple precision library for printf:

<table>
<thead>
<tr>
<th>Overhead</th>
<th>Command</th>
<th>Shared Object</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>57.96%</td>
<td>a.out</td>
<td>libc-2.17.so</td>
<td>[. ] __printf_fp</td>
</tr>
<tr>
<td>15.28%</td>
<td>a.out</td>
<td>libc-2.17.so</td>
<td>[. ] __mpn_mul_1</td>
</tr>
<tr>
<td>15.19%</td>
<td>a.out</td>
<td>libc-2.17.so</td>
<td>[. ] __mpn_divrem</td>
</tr>
<tr>
<td>5.79%</td>
<td>a.out</td>
<td>libc-2.17.so</td>
<td>[. ] hack_digit.13638</td>
</tr>
<tr>
<td>5.79%</td>
<td>a.out</td>
<td>libc-2.17.so</td>
<td>[. ] vfprintf</td>
</tr>
</tbody>
</table>
Exponent

- Full exponent range for IEEE double: $10^{-324} - 10^{308}$
- In general requires multiple precision arithmetic
- glibc pulls in a GNU multiple precision library for `printf`:

<table>
<thead>
<tr>
<th>Overhead</th>
<th>Command</th>
<th>Shared Object</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>57.96%</td>
<td>a.out</td>
<td>libc-2.17.so</td>
<td>[.] __printf_fp</td>
</tr>
<tr>
<td>15.28%</td>
<td>a.out</td>
<td>libc-2.17.so</td>
<td>[.] __mpn_mul_1</td>
</tr>
<tr>
<td>15.19%</td>
<td>a.out</td>
<td>libc-2.17.so</td>
<td>[.] __mpn_divrem</td>
</tr>
<tr>
<td>5.79%</td>
<td>a.out</td>
<td>libc-2.17.so</td>
<td>[.] hack_digit.13638</td>
</tr>
<tr>
<td>5.79%</td>
<td>a.out</td>
<td>libc-2.17.so</td>
<td>[.] vfprintf</td>
</tr>
</tbody>
</table>
Here be dragons: notable algorithms
Dragon

- Family of algorithms developed in 70s-80s and published in the paper "How to Print Floating-Point Numbers Accurately" by Steele & White (1990)

- The idea of tracking boundaries was introduced by White in 70s

- Dragon2: uses floating-point arithmetic for scaling by powers of 10

- Dragon4: uses multiprecision arithmetic for scaling

- Proved that fixed precision integer arithmetic can be used for some FP formats
Grisù

- Family of algorithms from the paper "Printing Floating-Point Numbers Quickly and Accurately with Integers" by Florian Loitsch (2010)

- DIY floating point: emulates floating point with extra precision (e.g. 64-bit for double giving 11 extra bits) using simple fixed-precision integer operations

- Precomputes powers of 10 and stores as DIY FP numbers

- Finds a power of 10 and multiplies the number by it to bring the exponent in the desired range

- With 11 extra bits Grisu3 produces shortest result in 99.5% of cases and tracks the uncertain region where it cannot guarantee shortness

- Relatively simple: can be implemented in 300 - 400 SLOC including some optimizations
Ryū

- An algorithm from the paper "Ryū: fast float-to-string conversion" by Ulf Adams (2018)
- Uses higher precision integer arithmetic (128-bit for double) and large precomputed tables for scaling
- Doesn't need fallback (good worst case)
What about C++?
C++17 introduced `<charconv>`

Low-level formatting and parsing primitives: `std::to_chars` and `std::from_chars`

Provides shortest decimal representation with round-trip guarantees and correct rounding 🦄

Locale-independent
```cpp
std::array<char, 20> buf; // What size?
std::to_chars_result result =
    std::to_chars(buf.data(), buf.data() + buf.size(), M_PI);
if (result.ec == std::errc()) {
    std::string_view sv(buf.data(), result.ptr - buf.data());
    // Use sv.
} else {
    // Handle error.
}
```

- `to_chars` is great but
- API is a bit too low-level
  - Manual buffer management, doesn't say how much to allocate
  - Error handling is cumbersome (slightly better with structured bindings)
- Cannot be easily & efficiently integrated into a higher-level facility
- Can't portably rely on it any time soon
C++20 std::format

- C++20 will have a higher-level formatting facility: std::format and friends

- Implemented in the {fmt} library: https://github.com/fmtlib/fmt

- The default is the shortest decimal representation with round-trip guarantees and correct rounding 🦄

- Control over locales: locale-independent by default

- Example:

  ```cpp
  std::format("{} == {} is {}\n", 0.1 + 0.2, 0.3, 0.1 + 0.2 == 0.3)
  ```

  returns "0.30000000000000004 == 0.3 is false" (no data loss)
• The default is shortest decimal representation with round-trip guarantees and correct rounding 

• Rich formatting mini-language

• Supports iterators, size computation, buffer preallocation

• High performance

• Zero dynamic memory allocations possible

• Locale control

• Portability: requires only a subset of C++11
Round-trip

```cpp
#include <fmt/core.h>

int main() {
    double a = 1.0 / 3.0;

    auto s = fmt::format("{}", a);
    double b = atof(s.c_str());
    assert(a == b);

    // succeeds:
    // a == 0.3333333333333333
    // b == 0.3333333333333333
}
```
Locale

Locale-independent by default:

```cpp
fmt::print("{}", 4.2); // prints 4.2
```

Locale-specific formatting is available via a separate format specifier:

```cpp
std::locale::global(
    std::locale("ru_RU.UTF-8");
fmt::print("{:n}", 4.2); // prints 4,2
```
fmt::print("{:*^10.2f}", 1.2345);
Mini-language

fmt::print("{:*^10.2f}", 1.2345);
Mini-language

```cpp
fmt::print("{{*:\^10.2f}}", 1.2345);
```
Mini-language

```cpp
fmt::print("{:*:^10.2f}", 1.2345);
```
Mini-language

```cpp
fmt::print("{:*^10.2f}", 1.2345);
```
fmt::print("{:.*^10.2f}", 1.2345);
fmt::print("{:*^10.2f}\n", 1.2345);

Format 1.2345 in the fixed form rounded to 2 digits after the decimal point and pad with * to 10 characters aligned to the center: ***1.23***
Zero allocations

- Dynamic memory allocations can be completely avoided & in particular the default will never allocate.

- No allocation & no need to specify buffer size:

  ```cpp
  fmt::memory_buffer buf;
  fmt::format_to(buf, "{}", 1.2345);
  // std::string_view(buf.data(), buf.size())
  // contains "1.2345"
  ```

- Single exact allocation & no extra copy (unlike to_chars):

  ```cpp
  std::string s;
  fmt::format_to(std::back_inserter(s), "{}", 1.2345);
  ```
Roundtrip precision: https://github.com/fmtlib/dtoa-benchmark (based on miloyip/dtoa-benchmark)
<table>
<thead>
<tr>
<th>Function</th>
<th>Time (ns)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>ostrstream</td>
<td>1,356.700</td>
<td>1.00x</td>
</tr>
<tr>
<td>ostrstream</td>
<td>1,202.847</td>
<td>1.13x</td>
</tr>
<tr>
<td>sprintf</td>
<td>1,002.506</td>
<td>1.35x</td>
</tr>
<tr>
<td>doubleconv</td>
<td>97.071</td>
<td>13.98x</td>
</tr>
<tr>
<td>fmt</td>
<td>96.071</td>
<td>14.12x</td>
</tr>
<tr>
<td>null</td>
<td>1.324</td>
<td>1,025.06x</td>
</tr>
</tbody>
</table>

Still a lot of optimization opportunities in fmt.


• {fmt}: https://github.com/fmtlib/fmt
Questions?