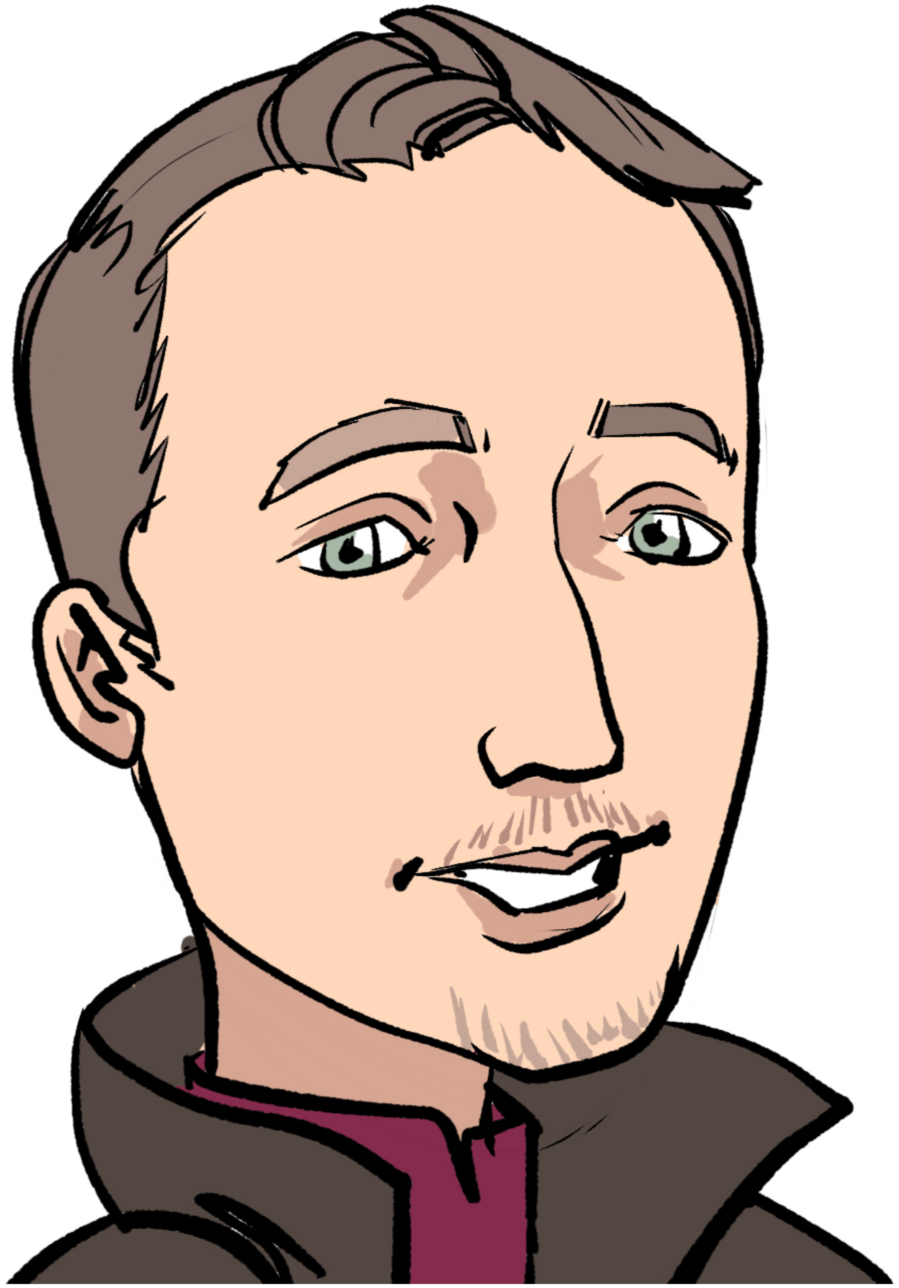
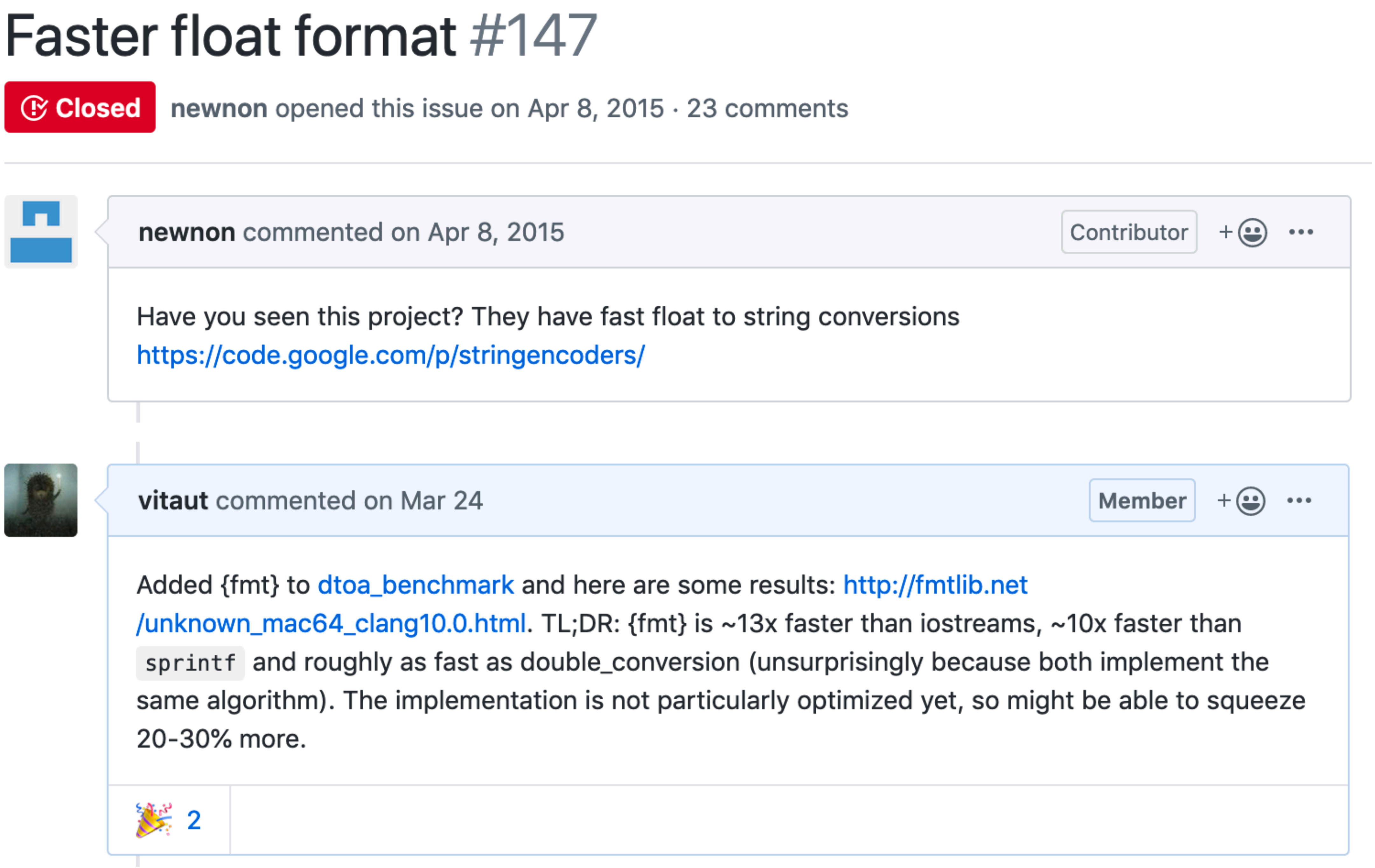
Formatting floating-point numbers

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

Closed **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.

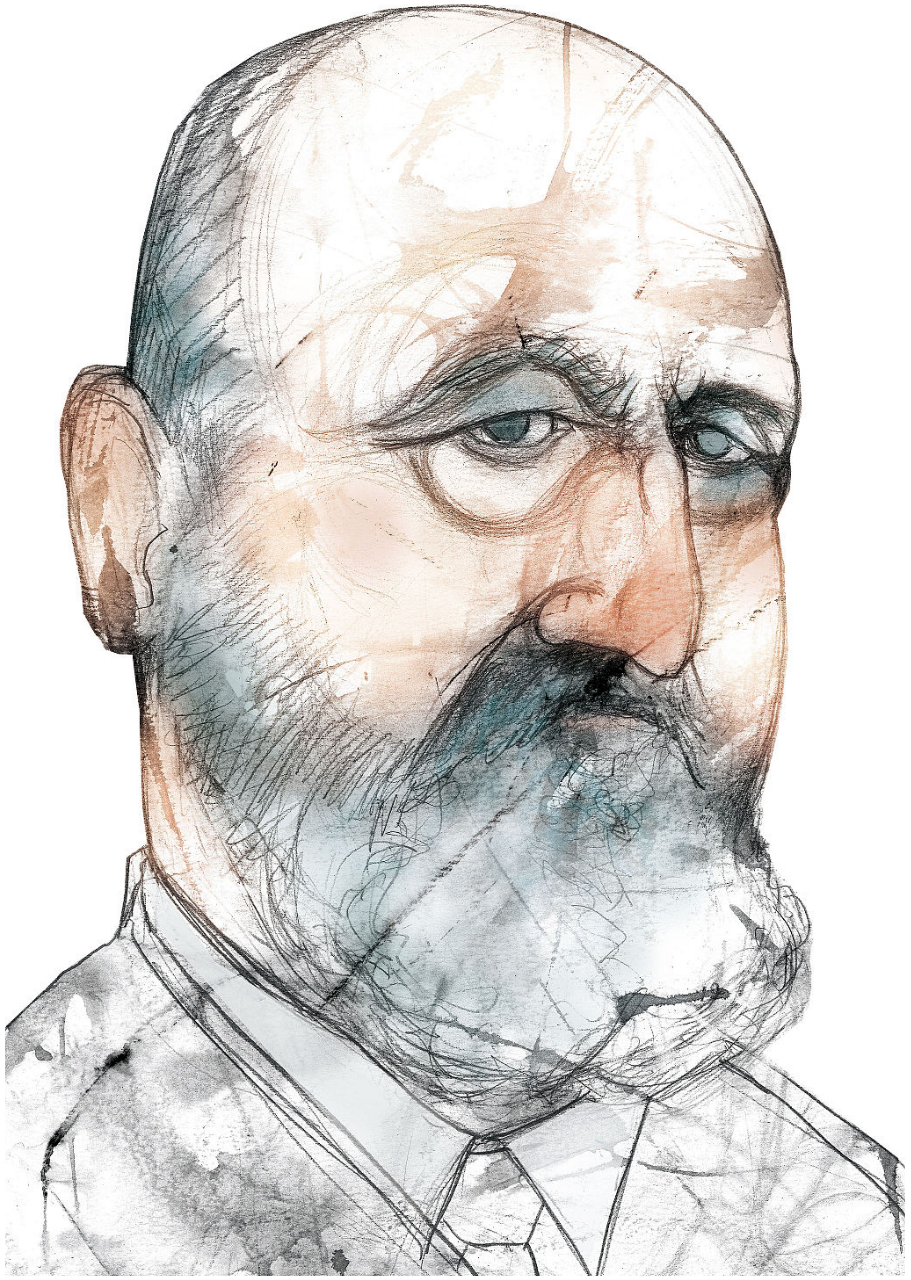
https://github.com/fmtlib/fmt/issues/147

"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



Portrait of Torres Quevedo by Eulogia Merle
(Fundación Española para la Ciencia y la Tecnología / CC BY-SA 4.0)

In early computers

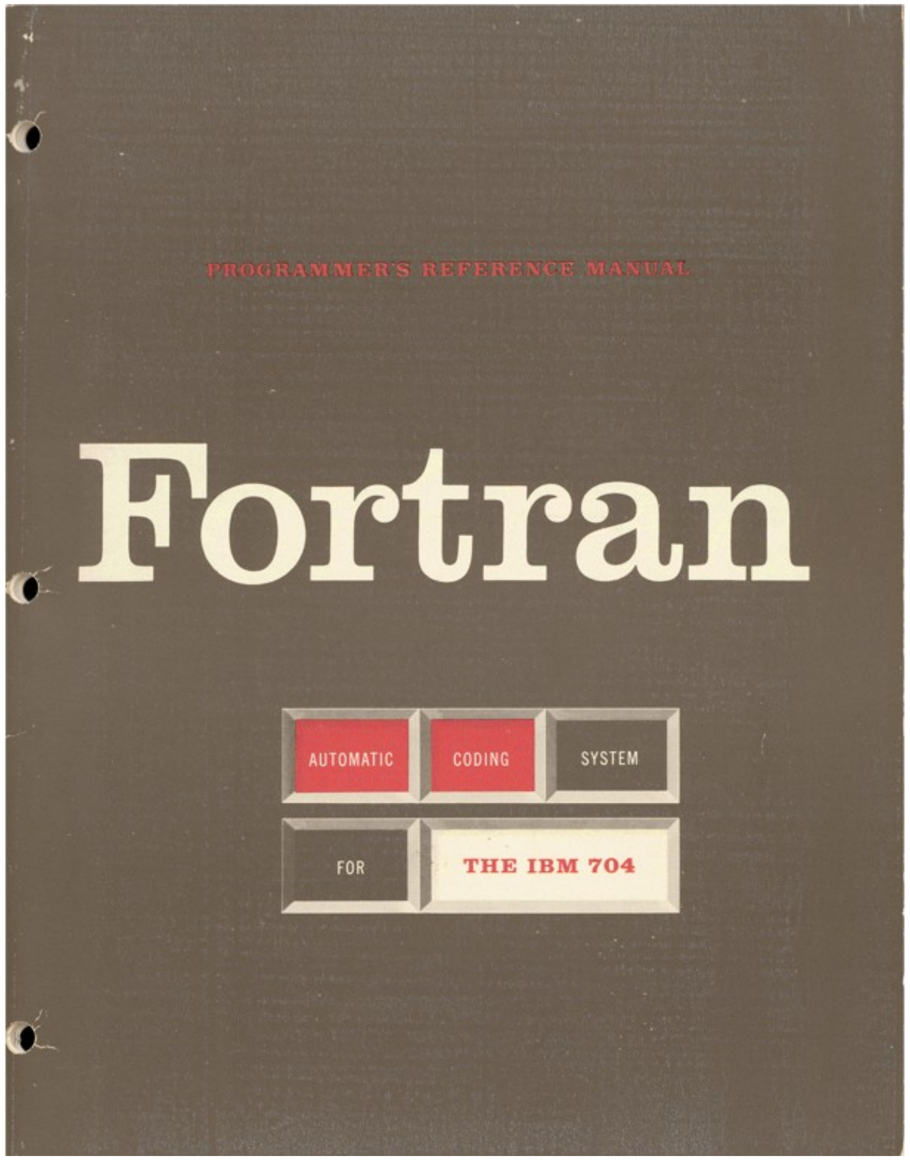
- 1938 Z1 by Konrad Zuse used 24-bit binary floating point
- 1941 relay-based Z3 had +/- infinity and exceptions (sort of)
- 1954 mass-produced IBM 704 introduced biased exponent

Replica of the Z1 in the German Museum of Technology in Berlin
(BLueFiSH.as / CC BY-SA 3.0)

Formatted I/O

FORTRAN had formatted floating-point I/O in 1950s (same time as comments were invented!):

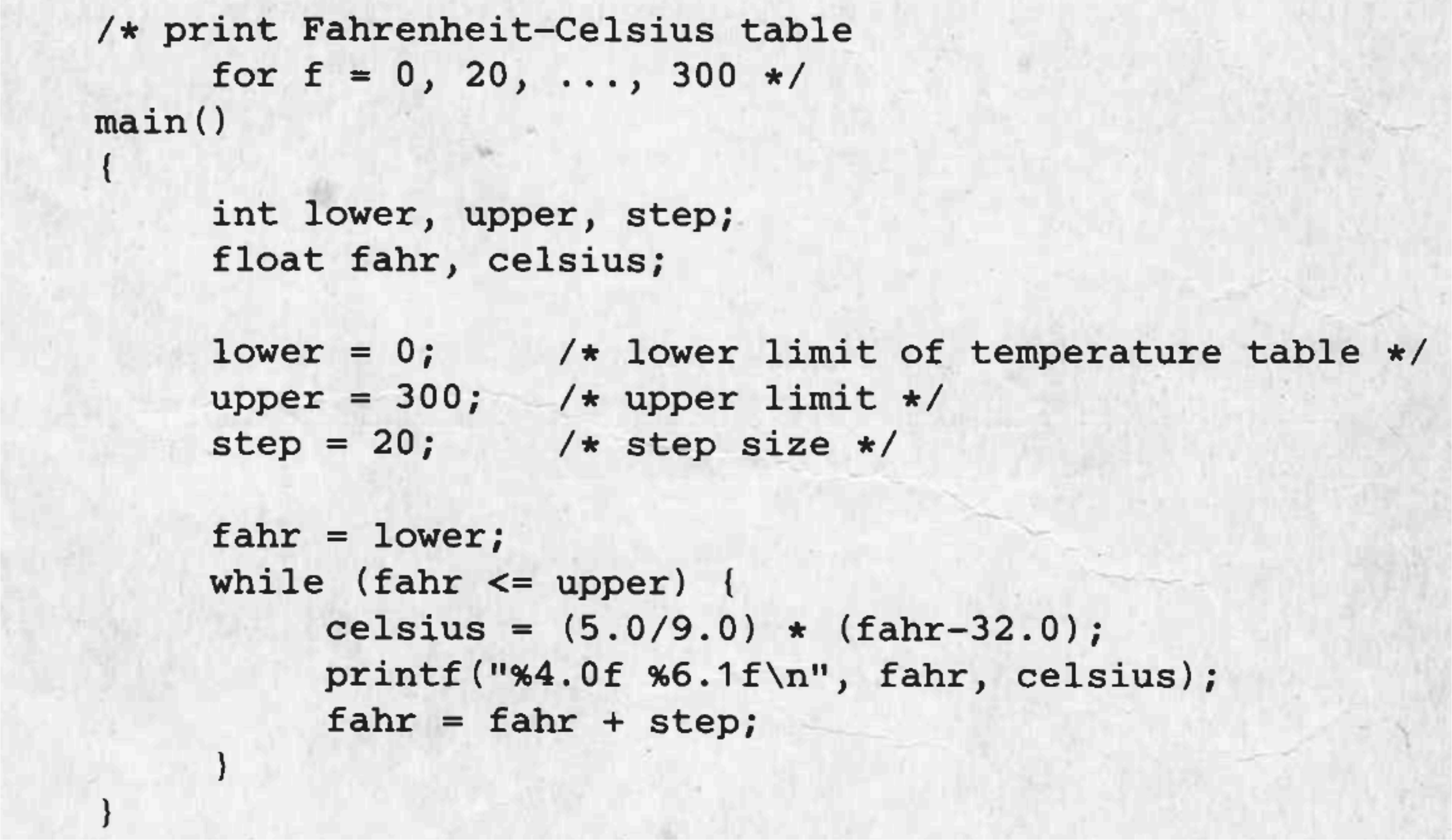
```
      WRITE OUTPUT TAPE 6, 601, IA, IB, IC, AREA
601   FORMAT (4H A= ,I5,5H  B= ,I5,5H  C= ,I5,
     &         8H  AREA= ,F10.2, 13H SQUARE UNITS)
```



Cover of The Fortran Automatic Coding System for the IBM 704 EDPM (public domain)

FP formatting in C

The C Programming Language, K&R (1978):



```
/* 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

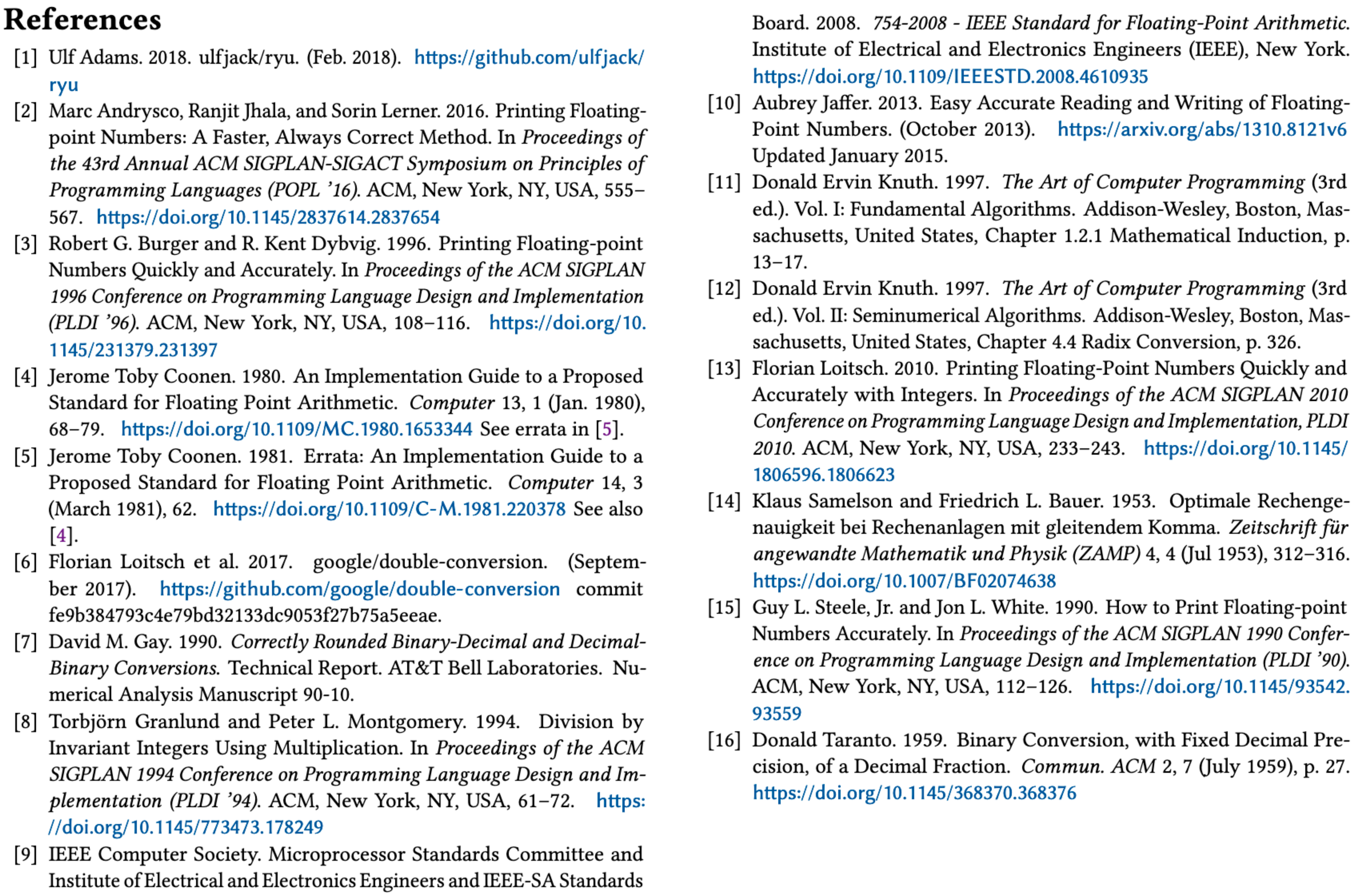
Solved problem?

- 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?



References

[1] Ulf Adams. 2018. ulfjack/ryu. (Feb. 2018). https://github.com/ulfjack/ryu

[2] Marc Andrysco, Ranjit Jhala, and Sorin Lerner. 2016. Printing Floating-point Numbers: A Faster, Always Correct Method. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 555–567. https://doi.org/10.1145/2837614.2837654

[3] Robert G. Burger and R. Kent Dybvig. 1996. Printing Floating-point Numbers Quickly and Accurately. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation (PLDI '96)*. ACM, New York, NY, USA, 108–116. https://doi.org/10.1145/231379.231397

[4] Jerome Toby Coonen. 1980. An Implementation Guide to a Proposed Standard for Floating Point Arithmetic. *Computer* 13, 1 (Jan. 1980), 68–79. https://doi.org/10.1109/MC.1980.1653344 See errata in [5].

[5] Jerome Toby Coonen. 1981. Errata: An Implementation Guide to a Proposed Standard for Floating Point Arithmetic. *Computer* 14, 3 (March 1981), 62. https://doi.org/10.1109/C-M.1981.220378 See also [4].

[6] Florian Loitsch et al. 2017. google/double-conversion. (September 2017). https://github.com/google/double-conversion commit fe9b384793c4e79bd32133dc9053f27b75a5eeae.

[7] David M. Gay. 1990. *Correctly Rounded Binary-Decimal and Decimal-Binary Conversions*. Technical Report. AT&T Bell Laboratories. Numerical Analysis Manuscript 90-10.

[8] Torbjörn Granlund and Peter L. Montgomery. 1994. Division by Invariant Integers Using Multiplication. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94)*. ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/773473.178249

[9] IEEE Computer Society. Microprocessor Standards Committee and Institute of Electrical and Electronics Engineers and IEEE-SA Standards Board. 2008. *754-2008 - IEEE Standard for Floating-Point Arithmetic*. Institute of Electrical and Electronics Engineers (IEEE), New York. https://doi.org/10.1109/IEEESTD.2008.4610935

[10] Aubrey Jaffer. 2013. Easy Accurate Reading and Writing of Floating-Point Numbers. (October 2013). https://arxiv.org/abs/1310.8121v6 Updated January 2015.

[11] Donald Ervin Knuth. 1997. *The Art of Computer Programming* (3rd ed.). Vol. I: Fundamental Algorithms. Addison-Wesley, Boston, Massachusetts, United States, Chapter 1.2.1 Mathematical Induction, p. 13–17.

[12] Donald Ervin Knuth. 1997. *The Art of Computer Programming* (3rd ed.). Vol. II: Seminumerical Algorithms. Addison-Wesley, Boston, Massachusetts, United States, Chapter 4.4 Radix Conversion, p. 326.

[13] Florian Loitsch. 2010. Printing Floating-Point Numbers Quickly and Accurately with Integers. In *Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation, PLDI 2010*. ACM, New York, NY, USA, 233–243. https://doi.org/10.1145/1806596.1806623

[14] Klaus Samelson and Friedrich L. Bauer. 1953. Optimale Rechengenauigkeit bei Rechenanlagen mit gleitendem Komma. *Zeitschrift für angewandte Mathematik und Physik (ZAMP)* 4, 4 (Jul 1953), 312–316. https://doi.org/10.1007/BF02074638

[15] Guy L. Steele, Jr. and Jon L. White. 1990. How to Print Floating-point Numbers Accurately. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI '90)*. ACM, New York, NY, USA, 112–126. https://doi.org/10.1145/93542.93559

[16] Donald Taranto. 1959. Binary Conversion, with Fixed Decimal Precision, of a Decimal Fraction. *Commun. ACM* 2, 7 (July 1959), p. 27. https://doi.org/10.1145/368370.368376

Solved problem?

References

[1] Ulf Adams. 2018. ulf
ryu
[2] Marc Andrysco, Ra
point Numbers: A
the 43rd Annual AC
Programming Langu
567. https://doi.org/1
[3] Robert G. Burger and R.
Numbers Quickly and Acc
1996 Conference on Programm
(PLDI '96). ACM, New York, NY,
1145/231379.231397
[4] Jerome Toby Coonen. 1980. An Implementation Guide to a Prop
Standard for Floating Point Arithmetic. *Computer* 13, 1 (Jan. 198
68–79. https://doi.org/10.1109/MC.1980.1653344 See errata in [5].
[5] Jerome Toby Coonen. 1981. Errata: An Implementation Guide to a Proposed Standard for Floating Point Arithmetic. *Computer* 14, 3 (March 1981), 62. https://doi.org/10.1109/C-M.1981.220378 See also [4].
[6] Florian Loitsch et al. 2017. google/double-conversion. (September 2017). https://github.com/google/double-conversion commit fe9b384793c4e79bd32133dc9053f27b75a5eeae.
[7] David M. Gay. 1990. *Correctly Rounded Binary-Decimal and Decimal-Binary Conversions.* Technical Report. AT&T Bell Laboratories. Numerical Analysis Manuscript 90-10.
[8] Torbjörn Granlund and Peter L. Montgomery. 1994. Division by Invariant Integers Using Multiplication. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94)*. ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/773473.178249
[9] IEEE Computer Society. Microprocessor Standards Committee and Institute of Electrical and Electronics Engineers and IEEE-SA Standards Board. 2008. *754-2008 - IEEE Standard for Floating-Point Arithmetic.* Institute of Electrical and Electronics Engineers (IEEE), New York. https://doi.org/10.1109/IEEESTD.2008.4610935
[10] Aubrey Jaffer. 2013. Easy Accurate Reading and Writing of Floating-Point Numbers. (October 2013). https://arxiv.org/abs/1310.8121v6 Updated January 2015.
[11] Donald Ervin Knuth. 1997. *The Art of Computer Programming* (3rd ed.). Vol. I: Fundamental Algorithms. Addison-Wesley, Boston, Massachusetts, United States, Chapter 1.2.1 Mathematical Induction, p. 13–17.
[12] Donald Ervin Knuth. 1997. *The Art of Computer Programming* (3rd ed.). Vol. II: Seminumerical Algorithms. Addison-Wesley, Boston, Massachusetts, United States, Chapter 4.4 Radix Conversion, p. 326.
13] Florian Loitsch. 2010. Printing Floating-Point Numbers Quickly and
Accurately with Integers. In *Proceedings of the ACM SIGPLAN 2010*
Conference on Programming Language Design and Implementation, PLDI
10. ACM, New York, NY, USA, 233–243. https://doi.org/10.1145/
596.1806623
Samelson and Friedrich L. Bauer. 1953. Optimale Rechenge-
t bei Rechenanlagen mit gleitendem Komma. *Zeitschrift für*
andte Mathematik und Physik (ZAMP) 4, 4 (Jul 1953), 312–316.
ttps://doi.org/10.1007/BF02074638
[15] Guy L. Steele, Jr. and Jon L. White. 1990. How to Print Floating-point Numbers Accurately. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI '90)*. ACM, New York, NY, USA, 112–126. https://doi.org/10.1145/93542.93559
[16] Donald Taranto. 1959. Binary Conversion, with Fixed Decimal Precision, of a Decimal Fraction. *Commun. ACM* 2, 7 (July 1959), p. 27. https://doi.org/10.1145/368370.368376

Meanwhile in 2019

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

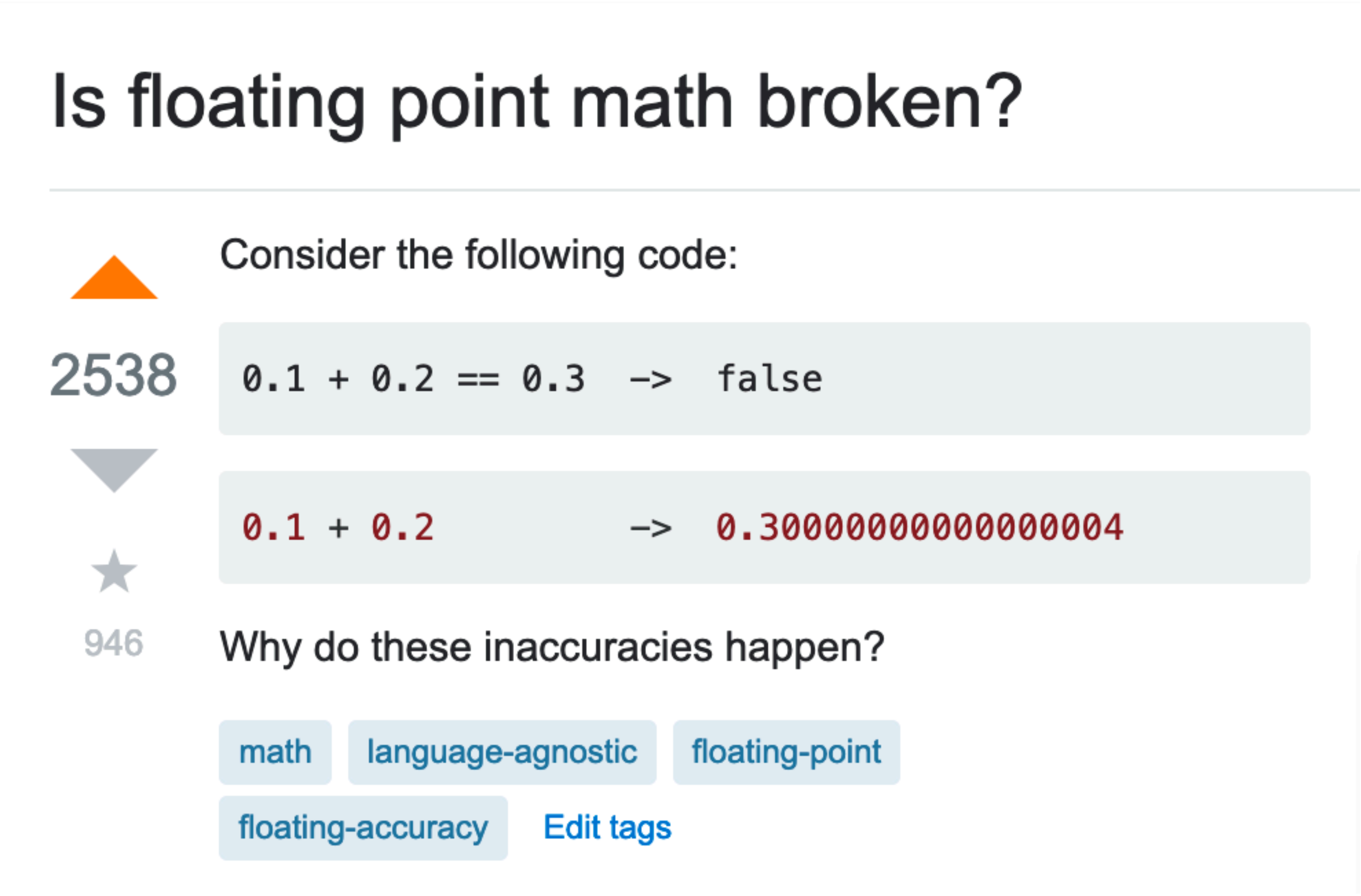
- Performance has much to be desired, esp. with iostreams

- 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

😔



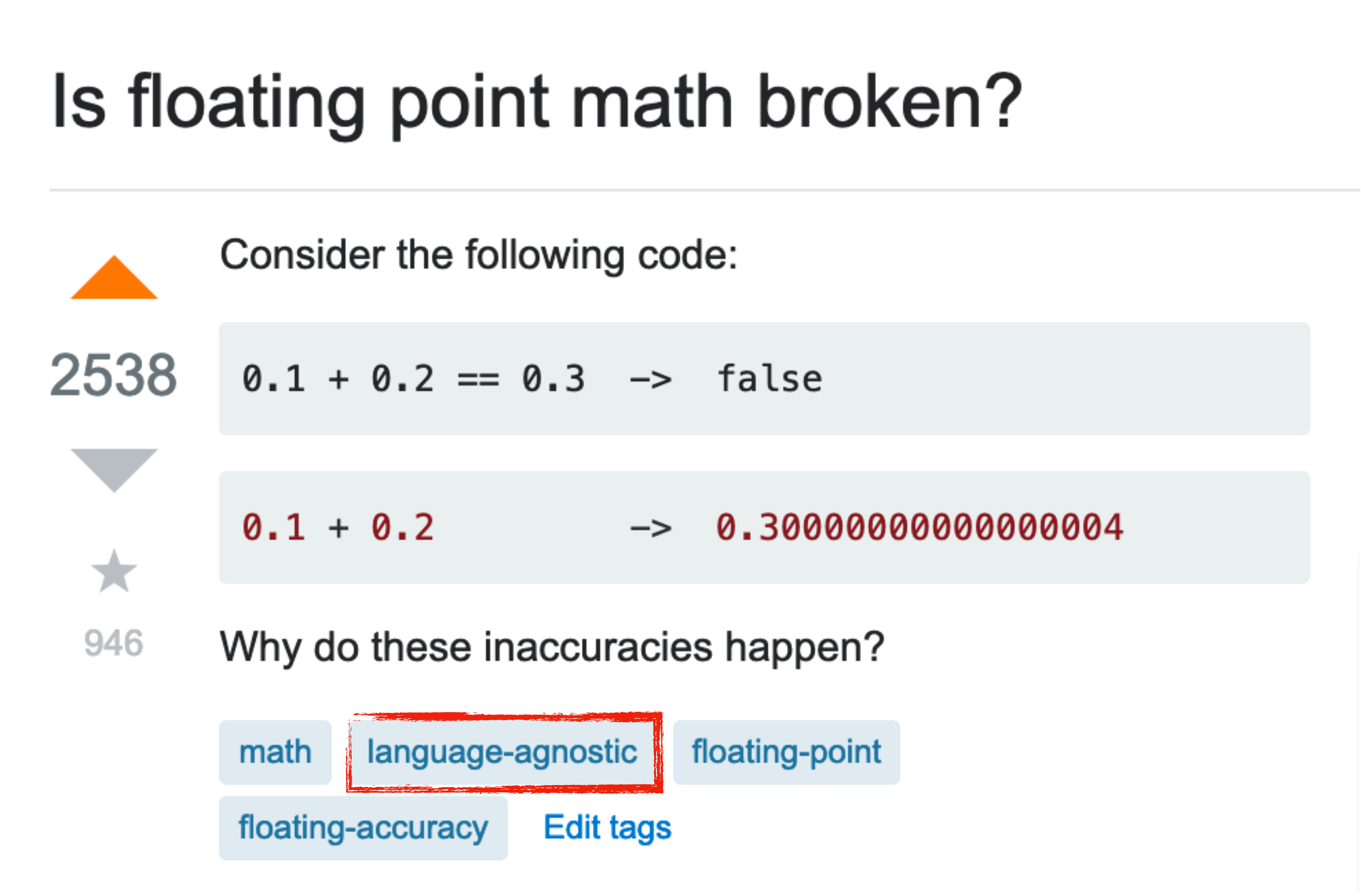
Is floating point math broken?

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 == 0.3  ->  false
```

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

Why do these inaccuracies happen?

0.30000000000000004

- Floating-point math is not broken, but can be tricky

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

```
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

(public domain)

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("{}", 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 *v* when $B^n < v^{m-1}$, where n is the number of base *B* digits, and m is the number of base *v* digits.

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

Photo of a random famous person
(public domain)

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):

```
sprintf("%.17g", 0.1);
```

prints "`0.10000000000000001`"

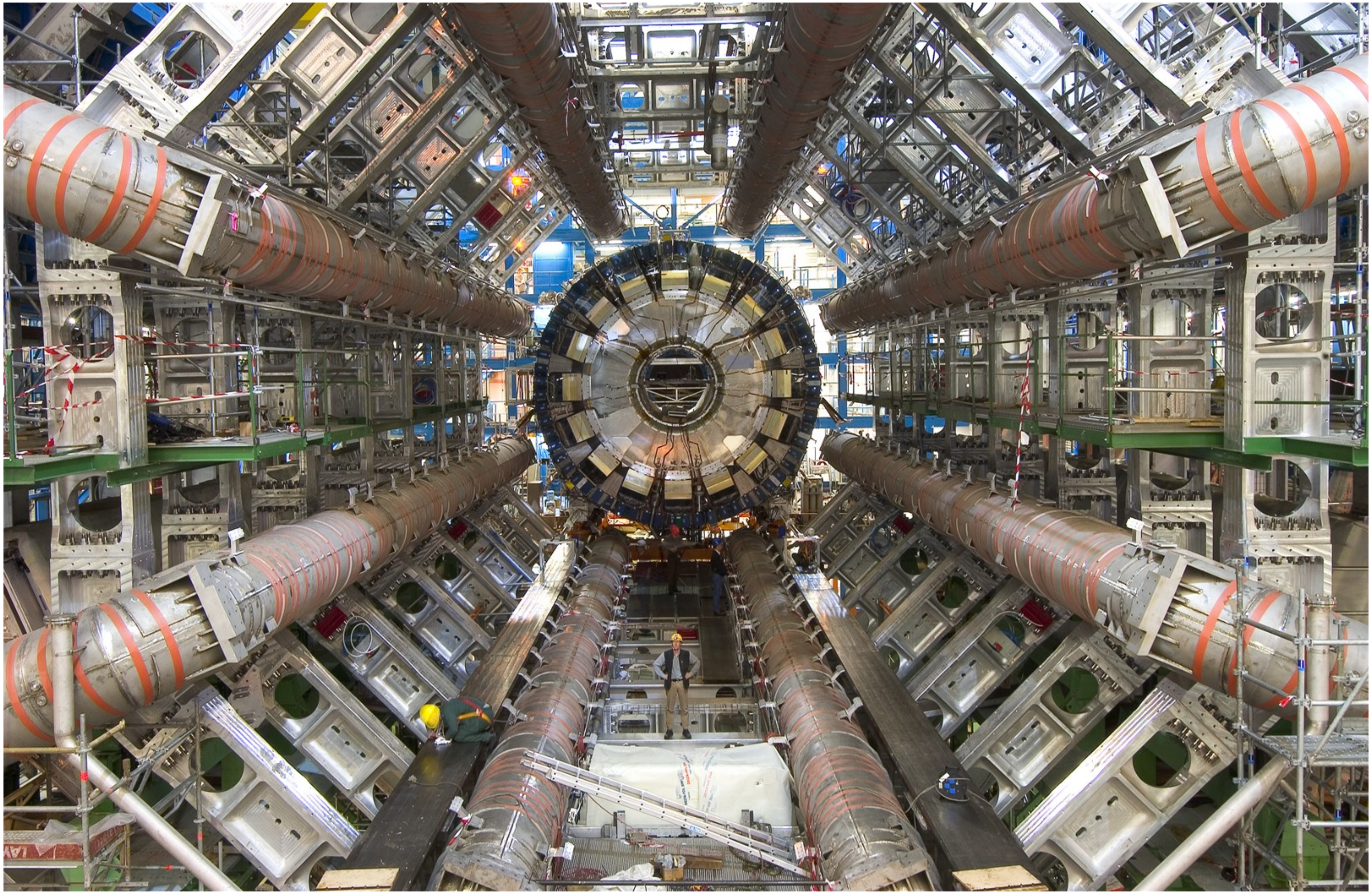
```
fmt::print("{}", 0.1);
```

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):
  - https://www.exploringbinary.com/incorrect-round-trip-conversions-in-visual-c-plus-plus/
  - https://www.exploringbinary.com/incorrectly-rounded-conversions-in-visual-c-plus-plus/
  - Had to disable some floating-point tests on MSVC due to broken rounding in `printf` and iostreams

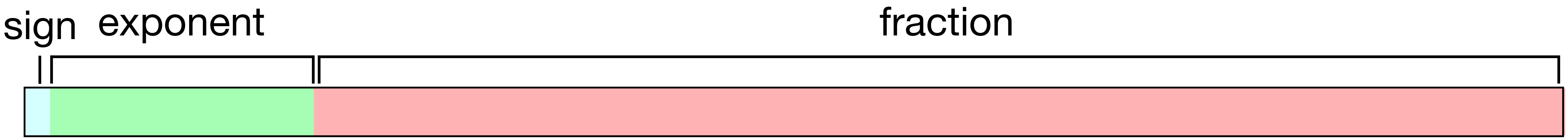
How does it work?



(老陳, CC BY-SA 4.0)

IEEE 754

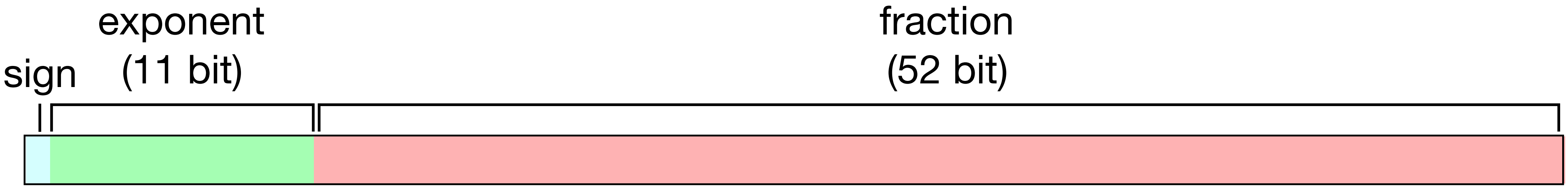
Binary floating point bit layout:



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

IEEE 754

Double-precision binary floating point bit layout:



sign

exponent
(11 bit)

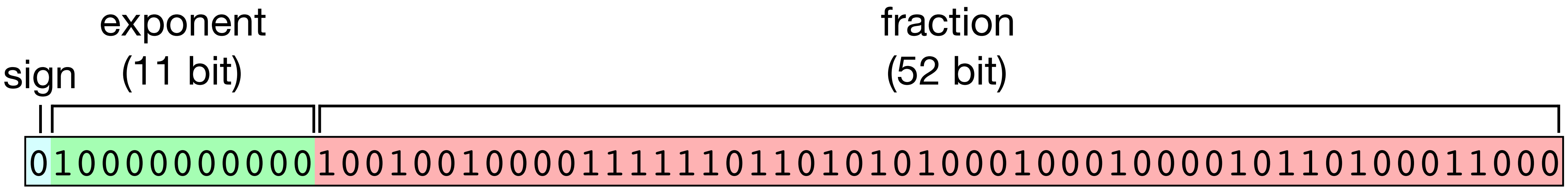
fraction
(52 bit)

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

where `bias = 1023`

Example

π approximation as `double` (`M_PI`):



$$v = (-1)^0 1.1001001000011111101101010100010001000010110100011000_2 \times 2^{(10000000000_2 - 1023_{10})} =$$

$$1.100100100001111110110101010001000100001011010001 1_2 \times 2 =$$

$$11.001001000011111101101010100010001000010110100011_2$$

Floating point formatting is …

Floating point formatting is easy*

Floating point formatting is easy*

*conceptually (terms and conditions apply)

Table 5: Procedure Dragon4 (Formatter-Feeding Process for Floating-Point Printout, Performing Free-Format Perfect Positive Floating-Point Printout)

```
process Dragon4;
  begin
    FORMAT ? $(b, e, f, p, B, CutoffMode, CutoffPlace)$;
    assert $CutoffMode = \text{"relative"} \Rightarrow CutoffPlace \leq 0$
    $RoundUpFlag \leftarrow false$;
    if $f = 0$ then FORMAT ! $(0, k)$ else
      $R \leftarrow shift_b(f, \max(e - p, 0))$;
      $S \leftarrow shift_b(1, \max(0, -(e - p)))$;
      $M^- \leftarrow shift_b(1, \max(e - p, 0))$;
      $M^+ \leftarrow M^-$;
      Fixup;
      loop
        $k \leftarrow k - 1$;
        $U \leftarrow \lfloor (R \times B)/S \rfloor$;
        $R \leftarrow (R \times B) \bmod S$;
        $M^- \leftarrow M^- \times B$;
        $M^+ \leftarrow M^+ \times B$;
        $low \leftarrow 2 \times R < M^-$;
        if RoundUpFlag
          then $high \leftarrow 2 \times R \geq (2 \times S) - M^+$
          else $high \leftarrow 2 \times R > (2 \times S) - M^+$ fi;
       while not low and not high
           and $k \neq CutoffPlace$ :
        FORMAT ! $(U, k)$;
      repeat;
      cases
        low and not high :  FORMAT ! $(U, k)$;
        high and not low :  FORMAT ! $(U + 1, k)$;
        (low and high) or (not low and not high) :
          cases
            $2 \times R \leq S$ :  FORMAT ! $(U, k)$;
            $2 \times R \geq S$ :  FORMAT ! $(U + 1, k)$;
          endcases;
      endcases;
    fi;
    comment Henceforth this process will generate as
      many "−1" digits as the caller desires, along
      with appropriate values of k.
    loop $k \leftarrow k - 1$; FORMAT ! $(-1, k)$ repeat;
  end;
```

Table 6: Procedure *Fixup*

```
procedure Fixup;
  begin
    if $f = shift_b(1, p - 1)$ then
      comment Account for unequal gaps.
      $M^+ \leftarrow shift_b(M^+, 1)$;
      $R \leftarrow shift_b(R, 1)$;
      $S \leftarrow shift_b(S, 1)$;
    fi;
    $k \leftarrow 0$;
    loop
     while $R < \lceil S/B \rceil$ :
      $k \leftarrow k - 1$;
      $R \leftarrow R \times B$;
      $M^- \leftarrow M^- \times B$;
      $M^+ \leftarrow M^+ \times B$;
    repeat;
    loop
      loop
       while $(2 \times R) + M^+ \geq 2 \times S$ :
        $S \leftarrow S \times B$;
        $k \leftarrow k + 1$;
      repeat;
      comment Perform any necessary adjustment
        of $M^-$ and $M^+$ to take into account the for-
        matting requirements.
      case CutoffMode of
        "normal" : $CutoffPlace \leftarrow k$;
        "absolute" : CutoffAdjust;
        "relative" :
          $CutoffPlace \leftarrow k + CutoffPlace$;
          CutoffAdjust;
      endcase;
     while $(2 \times R) + M^+ \geq 2 \times S$ :
    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
    $a \leftarrow CutoffPlace - k$;
    $y \leftarrow S$;
    cases
      $a \geq 0$ :  for $j \leftarrow 1$ to a do $y \leftarrow y \times B$;
      $a \leq 0$ :  for $j \leftarrow 1$ to $-a$ do $y \leftarrow \lceil y/B \rceil$;
    endcases;
    assert $y = \lceil S \times B^a \rceil$
    $M^- \leftarrow \max(y, M^-)$;
    $M^+ \leftarrow \max(y, M^+)$;
    if $M^+ = y$ then $RoundUpFlag \leftarrow true$ fi;
  end;
```

Table 9: Procedure *DigitChar*

```
procedure DigitChar(U);
  case 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 :  USER ! (" ");
     0 :  USER ! ("0");
     1 :  USER ! ("1");
     2 :  USER ! ("2");
     3 :  USER ! ("3");
     4 :  USER ! ("4");
     5 :  USER ! ("5");
     6 :  USER ! ("6");
     7 :  USER ! ("7");
     8 :  USER ! ("8");
     9 :  USER ! ("9");
    10 :  USER ! ("A");
    11 :  USER ! ("B");
    12 :  USER ! ("C");
    13 :  USER ! ("D");
    14 :  USER ! ("E");
    15 :  USER ! ("F");
  endcase;
```

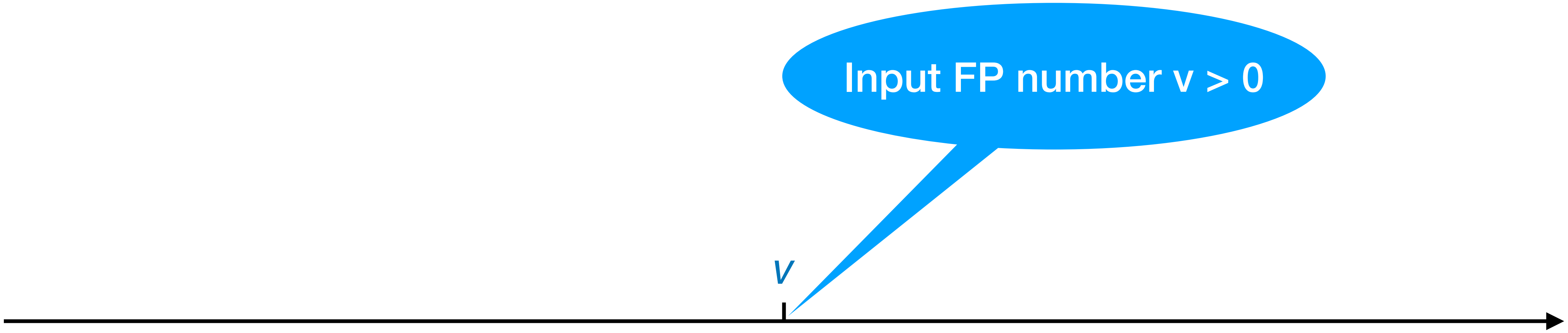
Table 10: Formatting process for free-format output

```
process Free-Format;
  begin
    USER ? $(b, e, f, p, B)$;
    GENERATE ! $(b, e, f, p, B, \text{"normal"}, 0)$;
    GENERATE ? $(U, k)$;
    if $k < 0$ then
      USER ! ("0");
      USER ! (".");
      fill($-k$, "0")
    fi;
    loop
      DigitChar(U);
      if $k = 0$ then USER ! (".") fi;
      GENERATE ? $(U, k)$;
     while $U \neq -1$ or $k \geq -1$ :
    repeat;
    USER ! ("@");
  end;
```

Table 11: Formatting process for fixed-format output

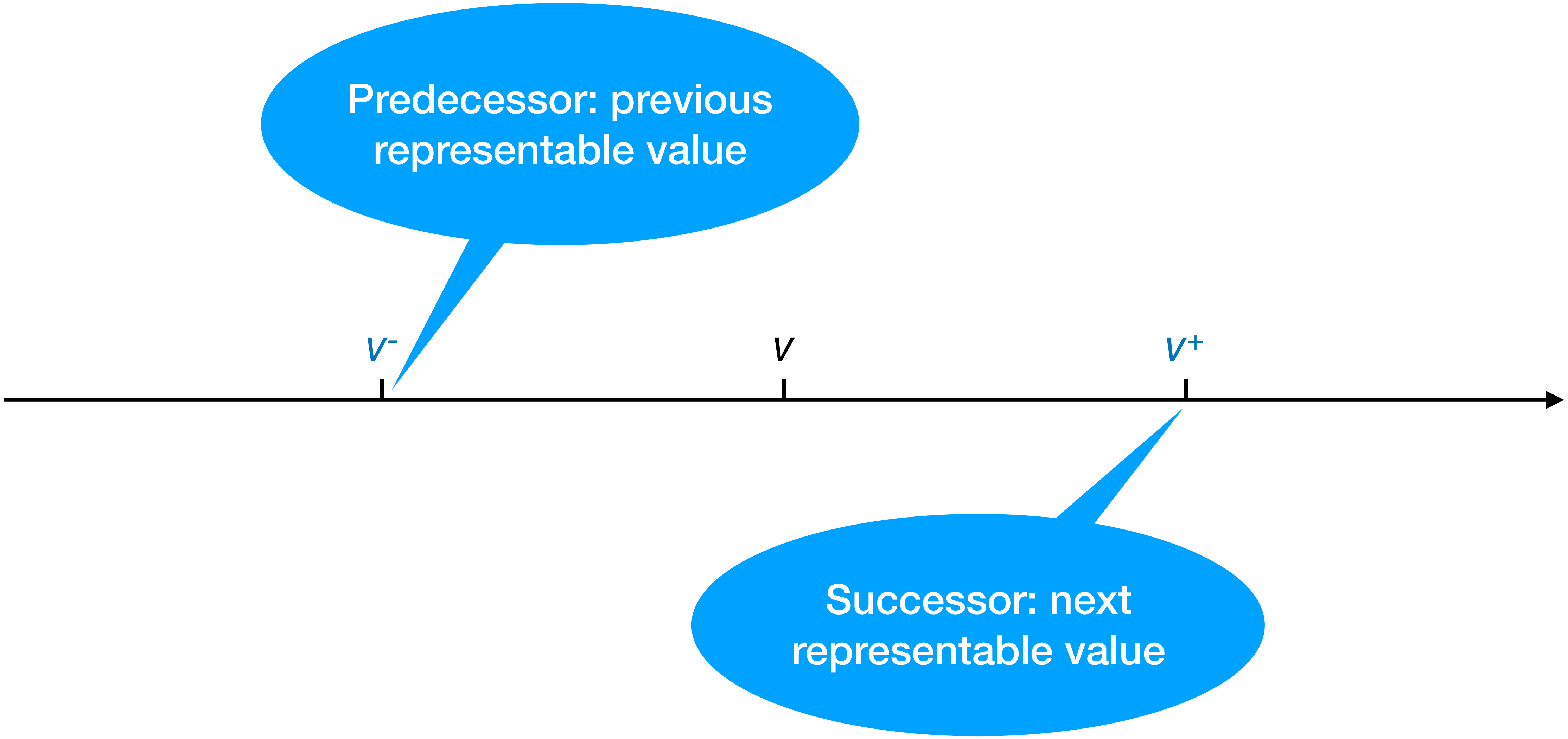
```
process Fixed-Format;
  begin
    USER ? $(b, e, f, p, B, w, d)$;
    assert $d \geq 0 \wedge w \geq \max(d + 1, 2)$
    $c \leftarrow w - d - 1$;
    GENERATE ! $(b, e, f, p, B, \text{"absolute"}, -d)$;
    GENERATE ? $(U, k)$;
    if $k < c$ then
      if $k < 0$ then
        if $c > 0$ then fill($c - 1$, " "); USER ! ("0") fi;
        USER ! (".");
        fill($\min(-k, d)$, "0");
      else fill($c - k - 1$, " ") fi;
      loop
       while $k \geq -d$ :
        DigitChar(U);
        if $k = 0$ then USER ! (".") fi;
        GENERATE ? $(U, k)$;
      repeat;
    else fill(w, "*") fi;
    USER ! ("@");
  end;
```

Input



Input FP number v > 0

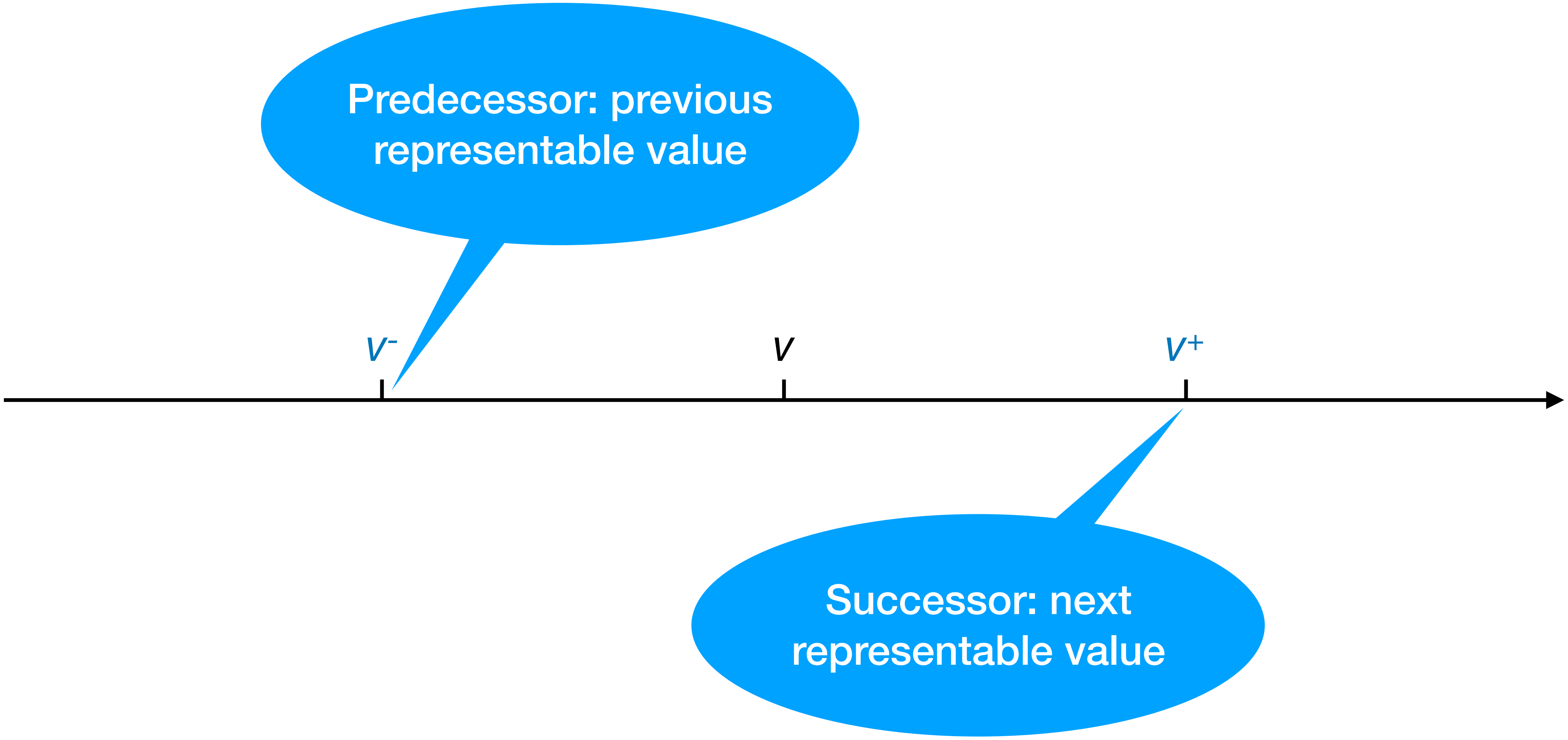
Neighbors



Predecessor: previous representable value

Successor: next representable value

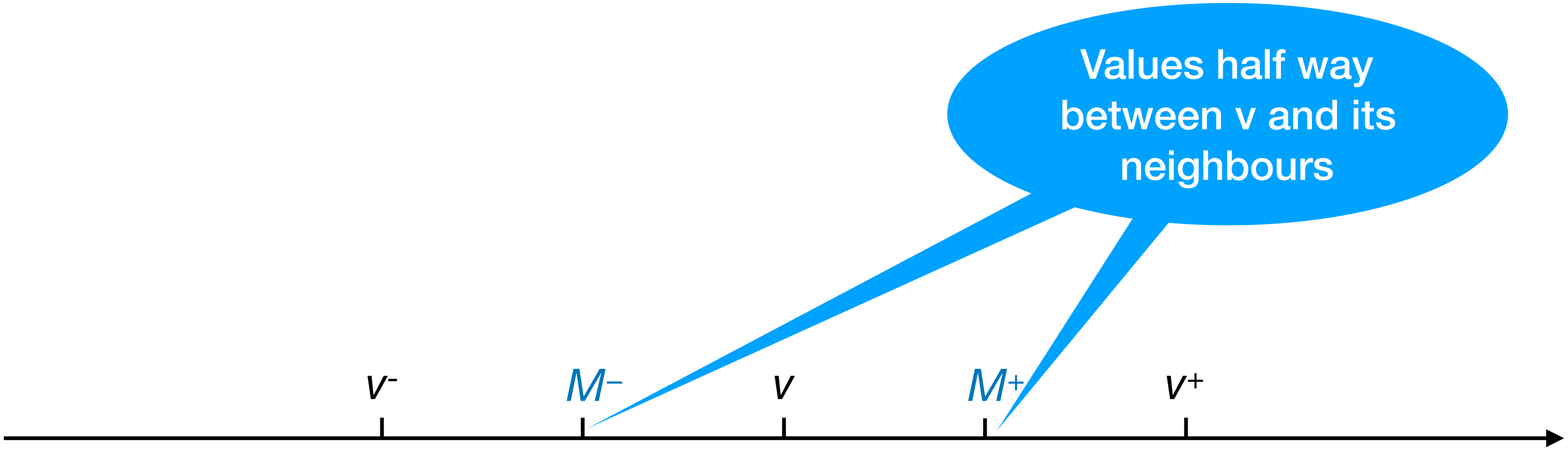
Neighbours



Predecessor: previous representable value

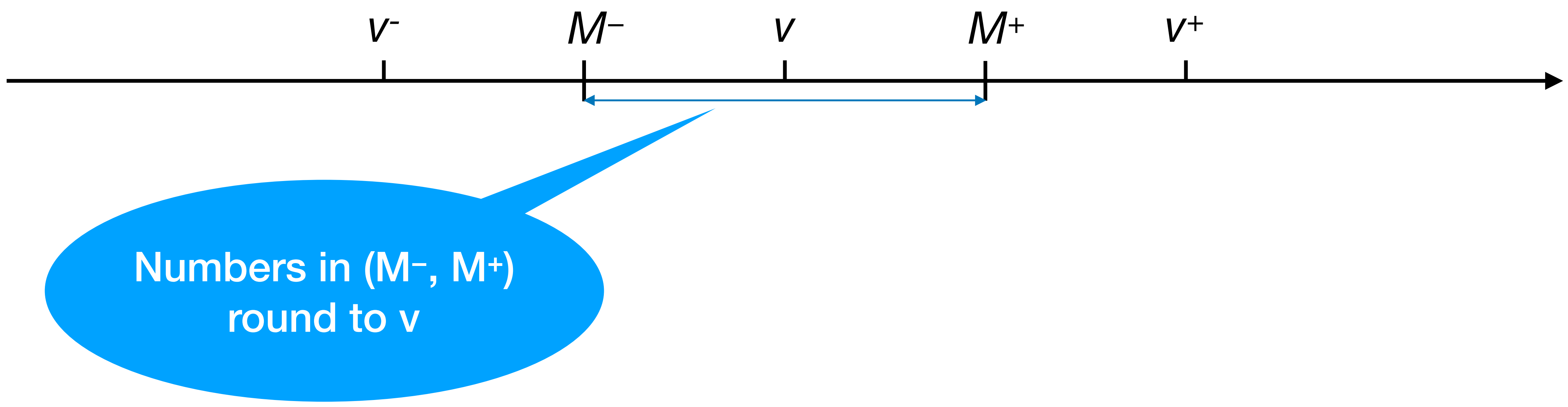
Successor: next representable value

Boundaries



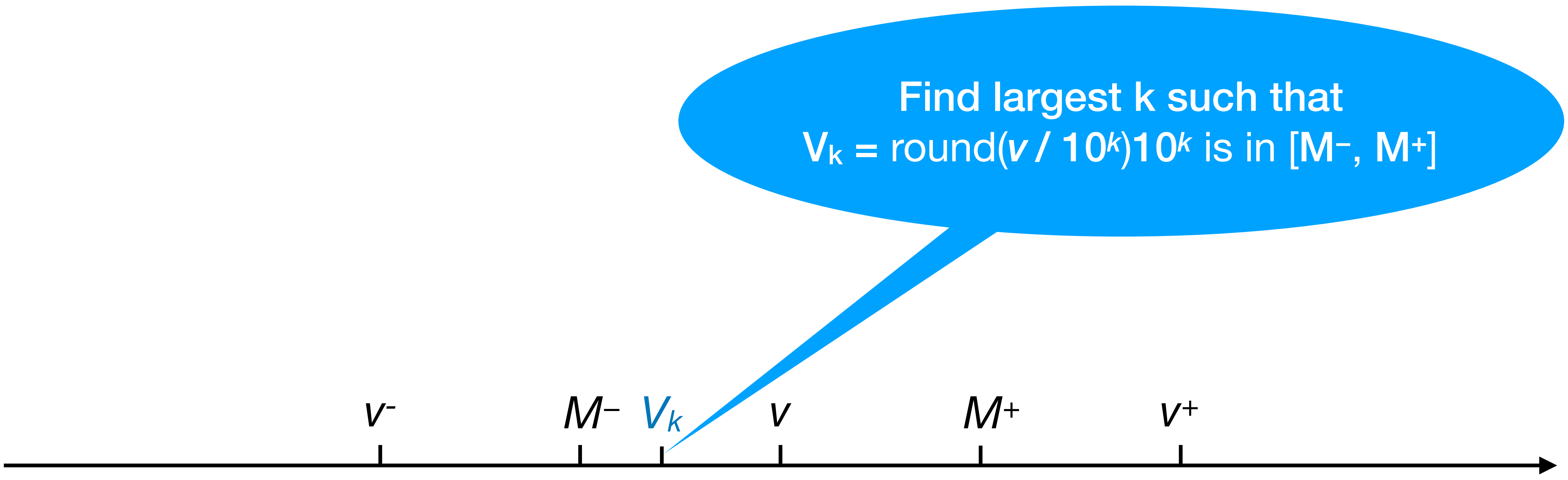
Values half way between v and its neighbours

Boundaries

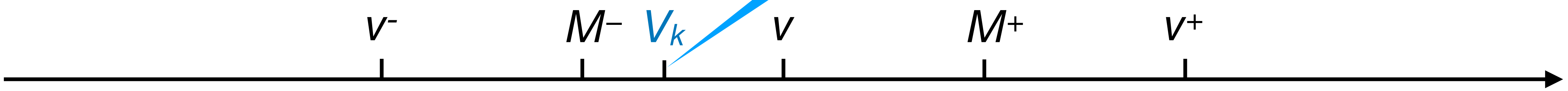


Numbers in (M^-, M^+) round to v

Find power of 10


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

Result



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

result = format("{}e{}", $\text{round}(v / 10^k)$, k)
$\text{round}(v / 10^k)$ and k are ints

Example

`Input: v = 1.23e45`

v^- `= 1229999999999999981535854398249094938452033 5360 =`
`0b11011100100111101101010010000011110111101000010100011 * 2**97`

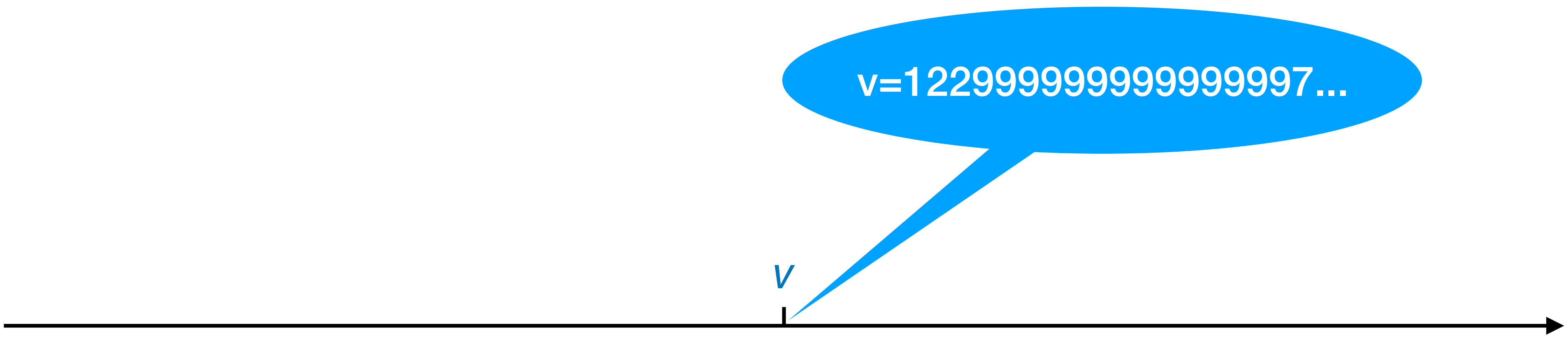
M^- `= 1229999999999999989458670649675528697806428 5696 =`
`0b110111001001111011010100100000111101111010000101000111 * 2**96`

v `= 1229999999999999997381486901101962457160823 6032 =`
`0b11011100100111101101010010000011110111101000010100100 * 2**97`

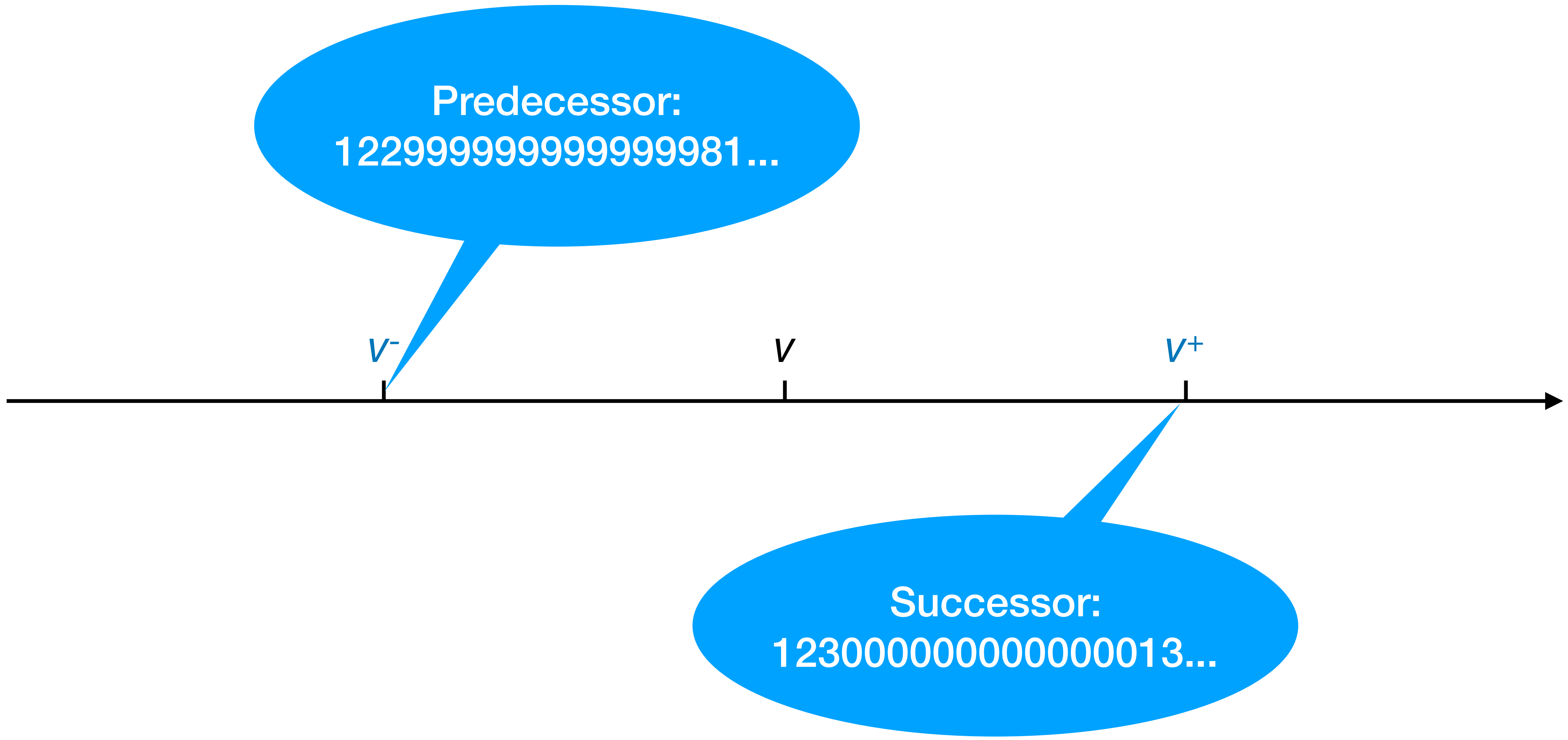
M^+ `= 1230000000000000005304303152528396216515218 6368 =`
`0b110111001001111011010100100000111101111010000101001001 * 2**96`

v^+ `= 1230000000000000013227119403954829975869613 6704 =`
`0b11011100100111101101010010000011110111101000010100101 * 2**97`

Example



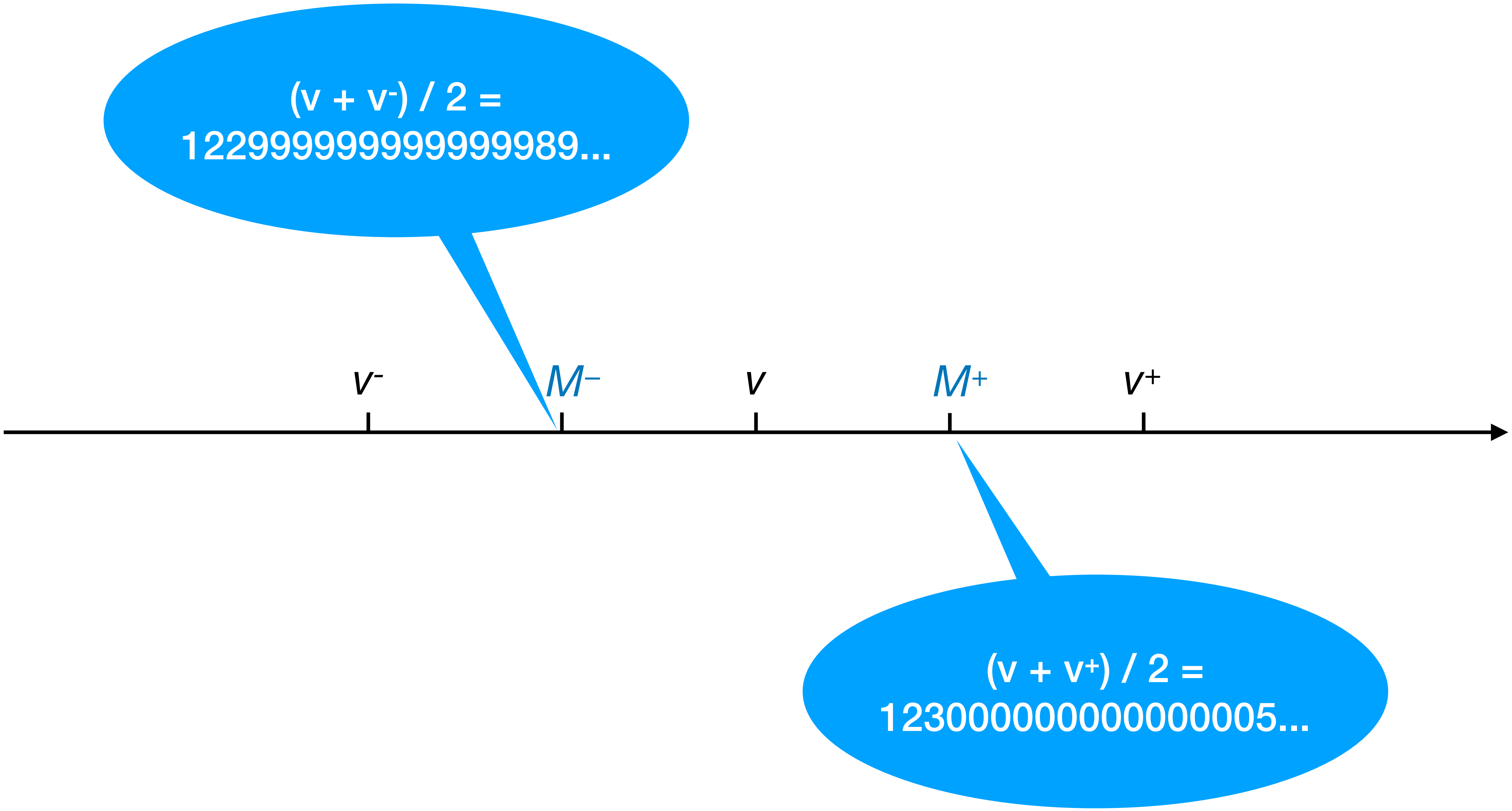
Neighbours



Predecessor:
1229999999999999981...

Successor:
123000000000000013...

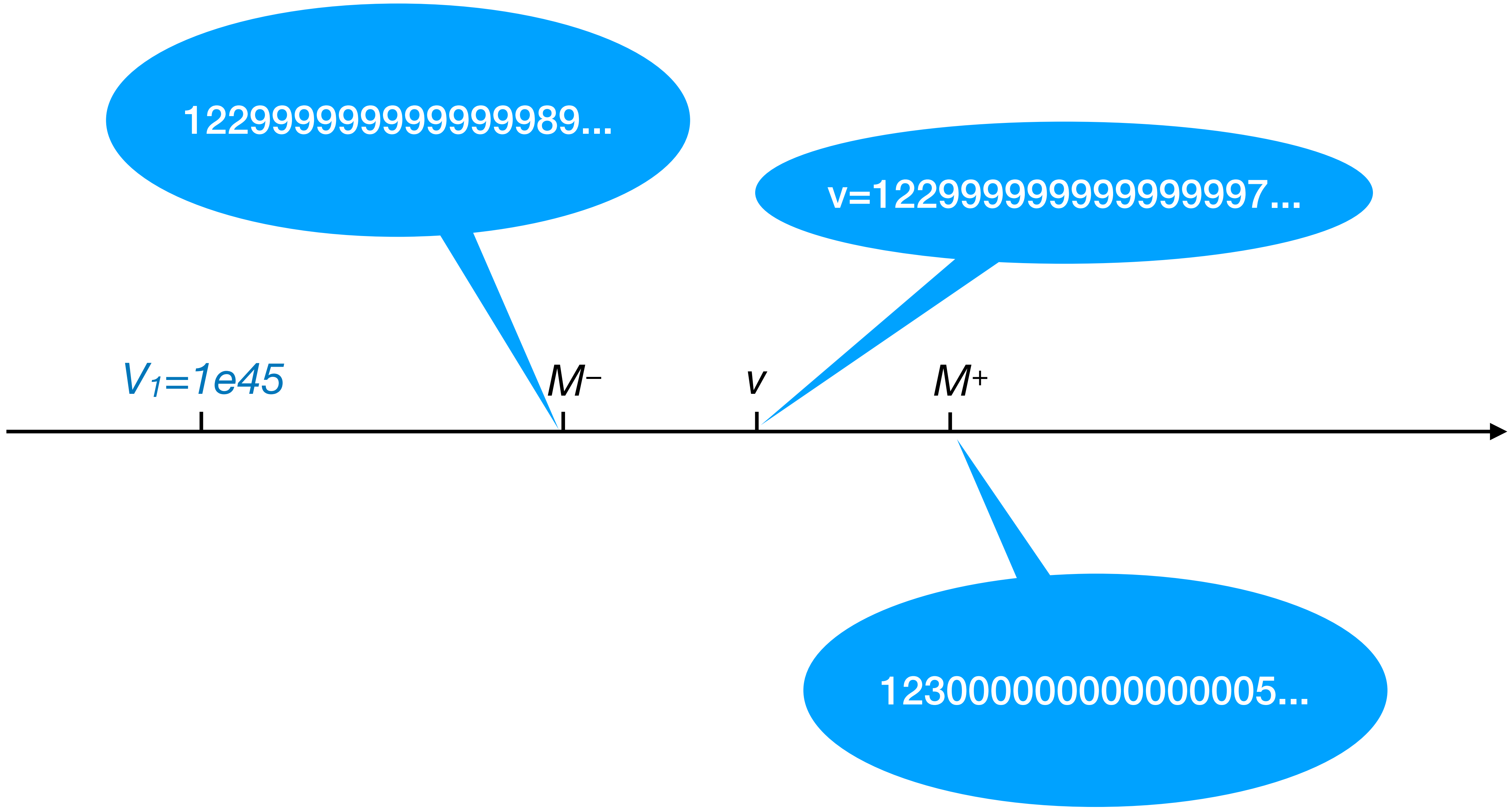
Boundaries



$(v + v^-) / 2 =$
122999999999999989...

$(v + v^+) / 2 =$
123000000000000005...

Find power of 10



122999999999999989...

v=122999999999999997...

V_1=1e45

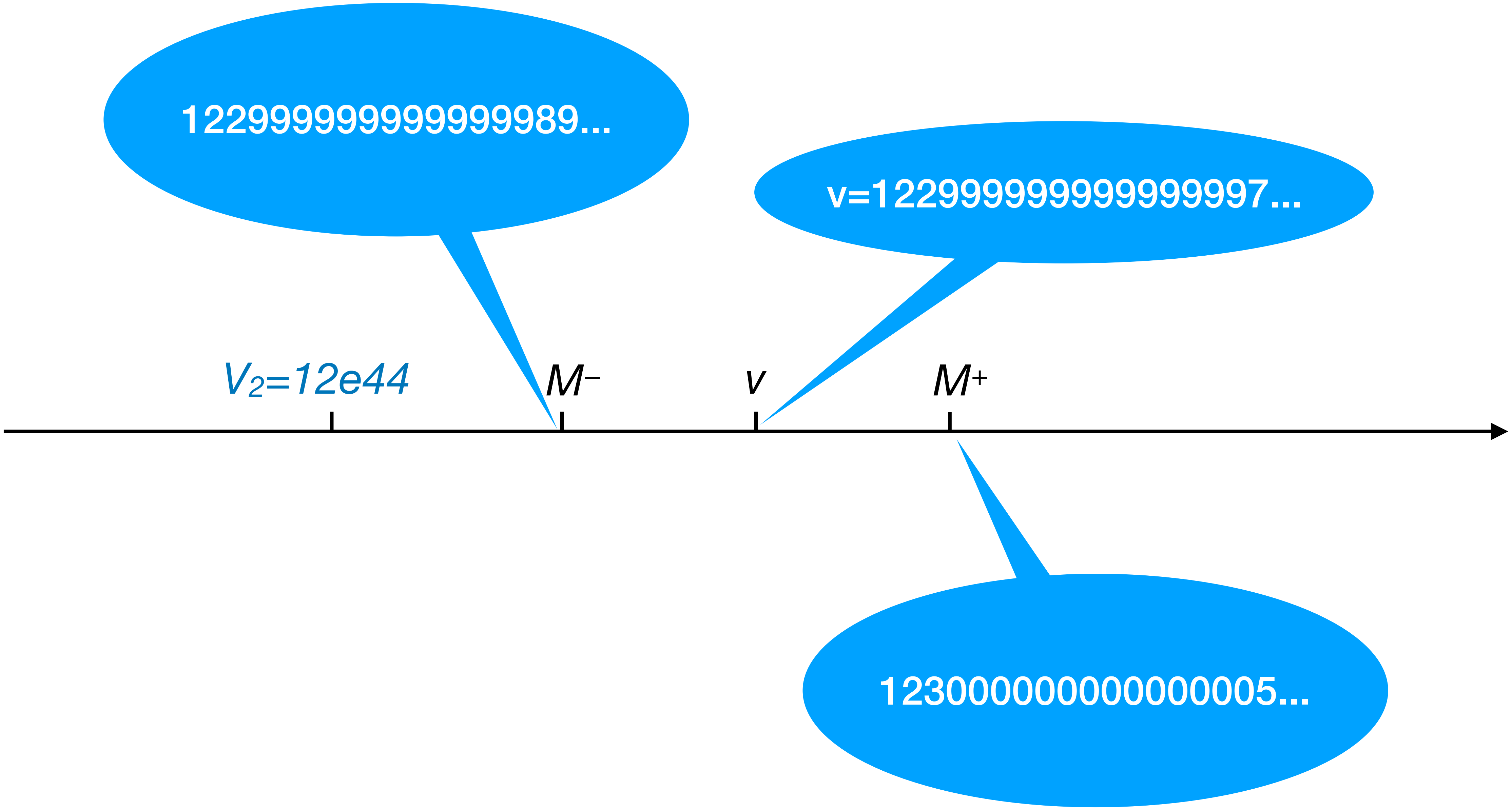
M^-

v

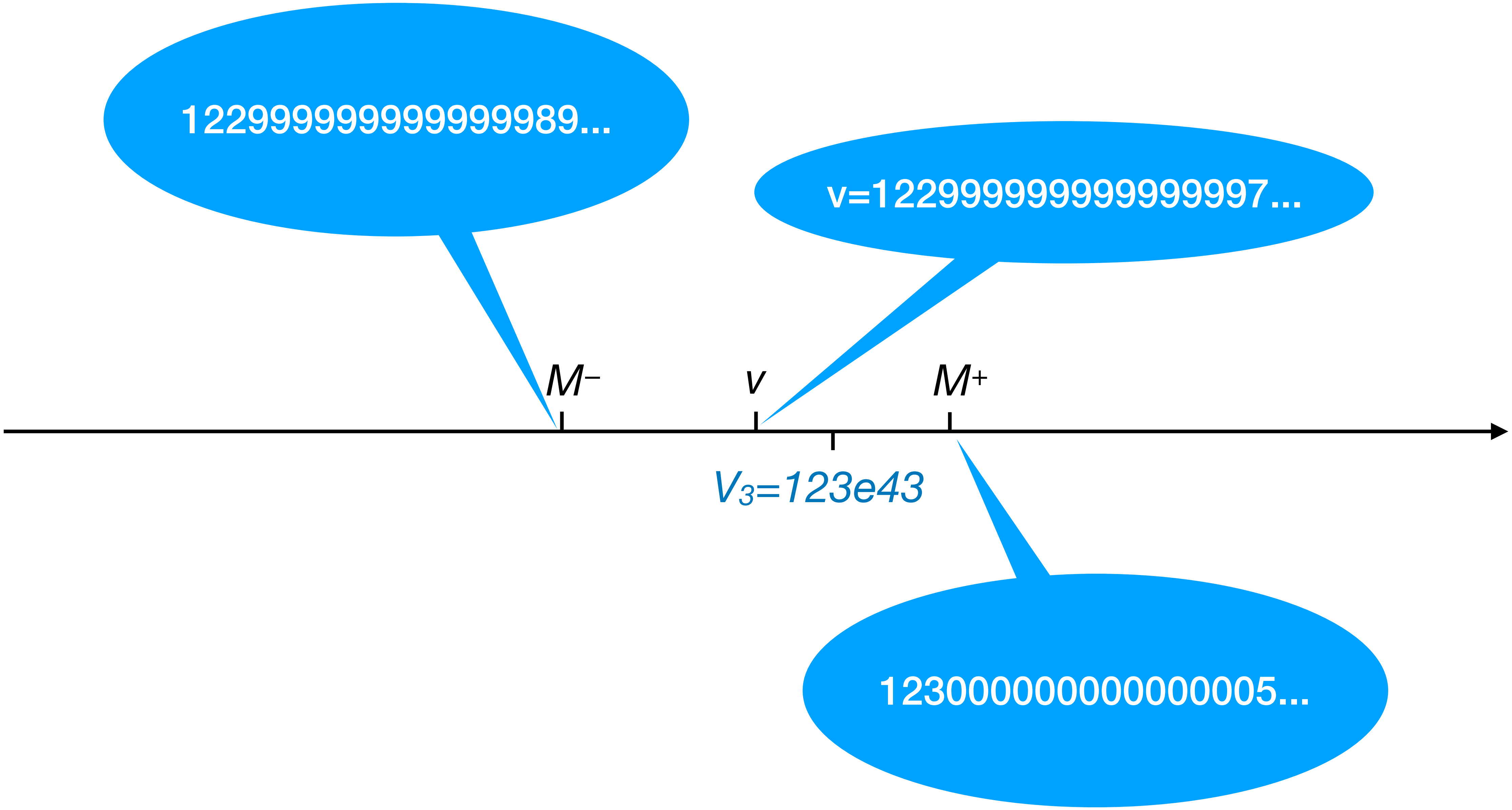
M^+

123000000000000005...

Find power of 10



Find power of 10



122999999999999989...

v=122999999999999997...

M^-

v

M^+

V_3=123e43

123000000000000005...

(image by Simon A. Eugster)

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`:

```
Overhead  Command  Shared Object       Symbol
  57.96%  a.out    libc-2.17.so        [.] __printf_fp
  15.28%  a.out    libc-2.17.so        [.] __mpn_mul_1
  15.19%  a.out    libc-2.17.so        [.] __mpn_divrem
   5.79%  a.out    libc-2.17.so        [.] hack_digit.13638
   5.79%  a.out    libc-2.17.so        [.] vfprintf
```

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`:

```
Overhead  Command  Shared Object     Symbol
  57.96%  a.out    libc-2.17.so      [.] __printf_fp
  15.28%  a.out    libc-2.17.so      [.] __mpn_mul_1
  15.19%  a.out    libc-2.17.so      [.] __mpn_divrem
   5.79%  a.out    libc-2.17.so      [.] hack_digit.13638
   5.79%  a.out    libc-2.17.so      [.] vfprintf
```

(public domain)

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++?

<charconv>

- 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

std::to_chars

```
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:

```
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)

{fmt}

- 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

```
#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:

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

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

```
std::locale::global(
  std::locale("ru_RU.UTF-8"));
fmt::print("{:n}", 4.2); // prints 4,2
```

Mini-language

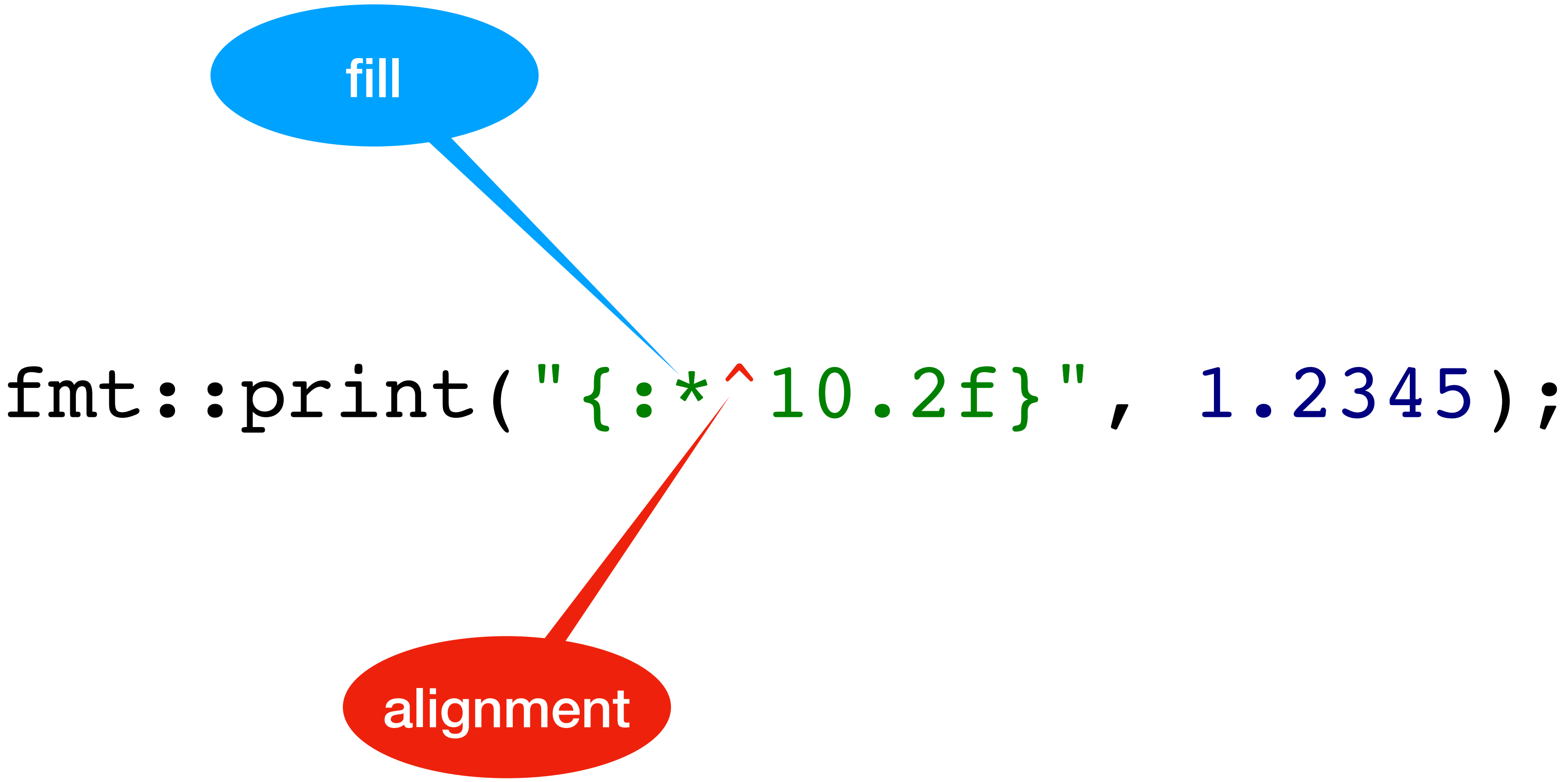
```
fmt::print("{:*^10.2f}", 1.2345);
```

Mini-language

fill

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

Mini-language

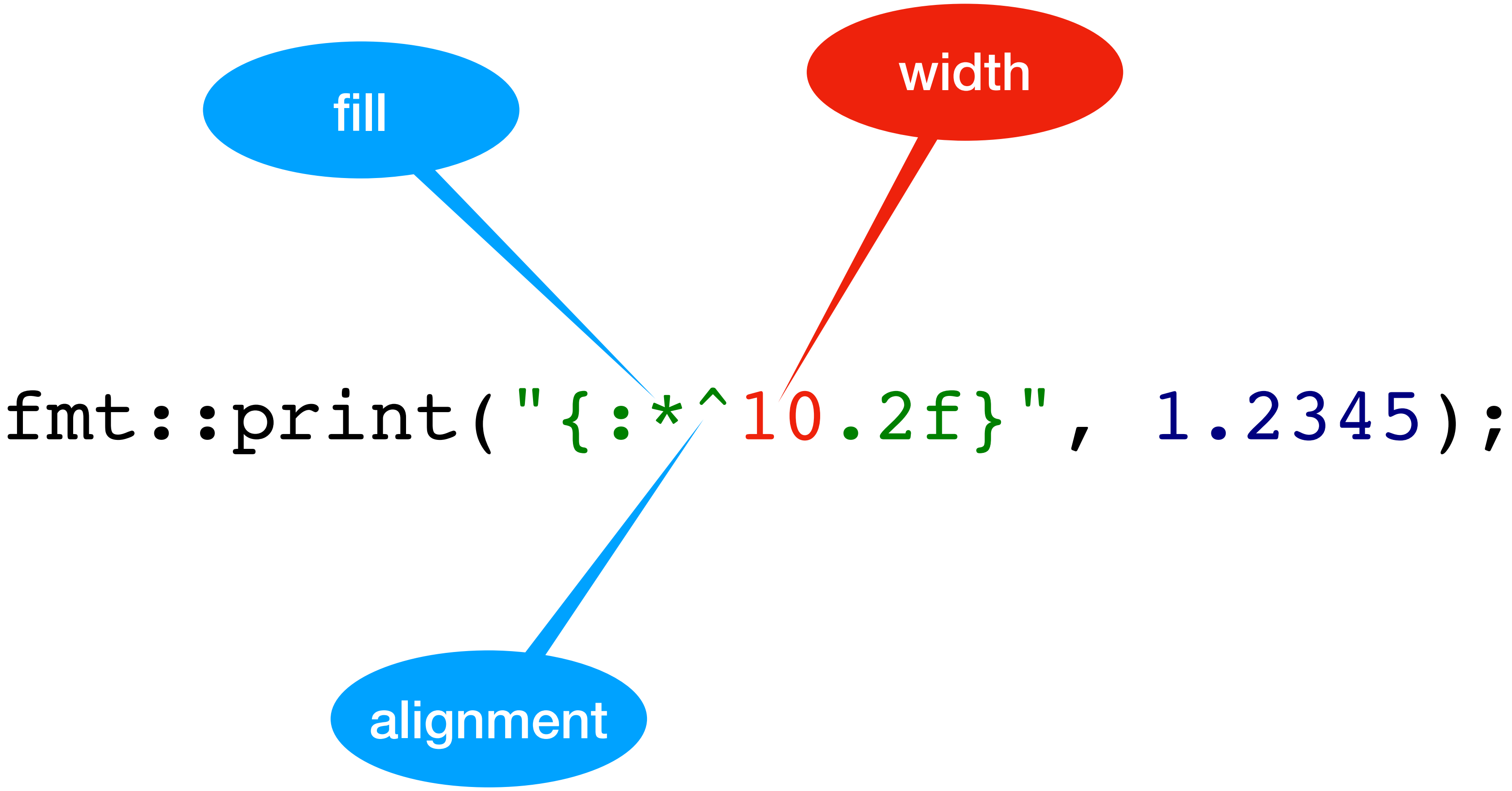


fill

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

alignment

Mini-language



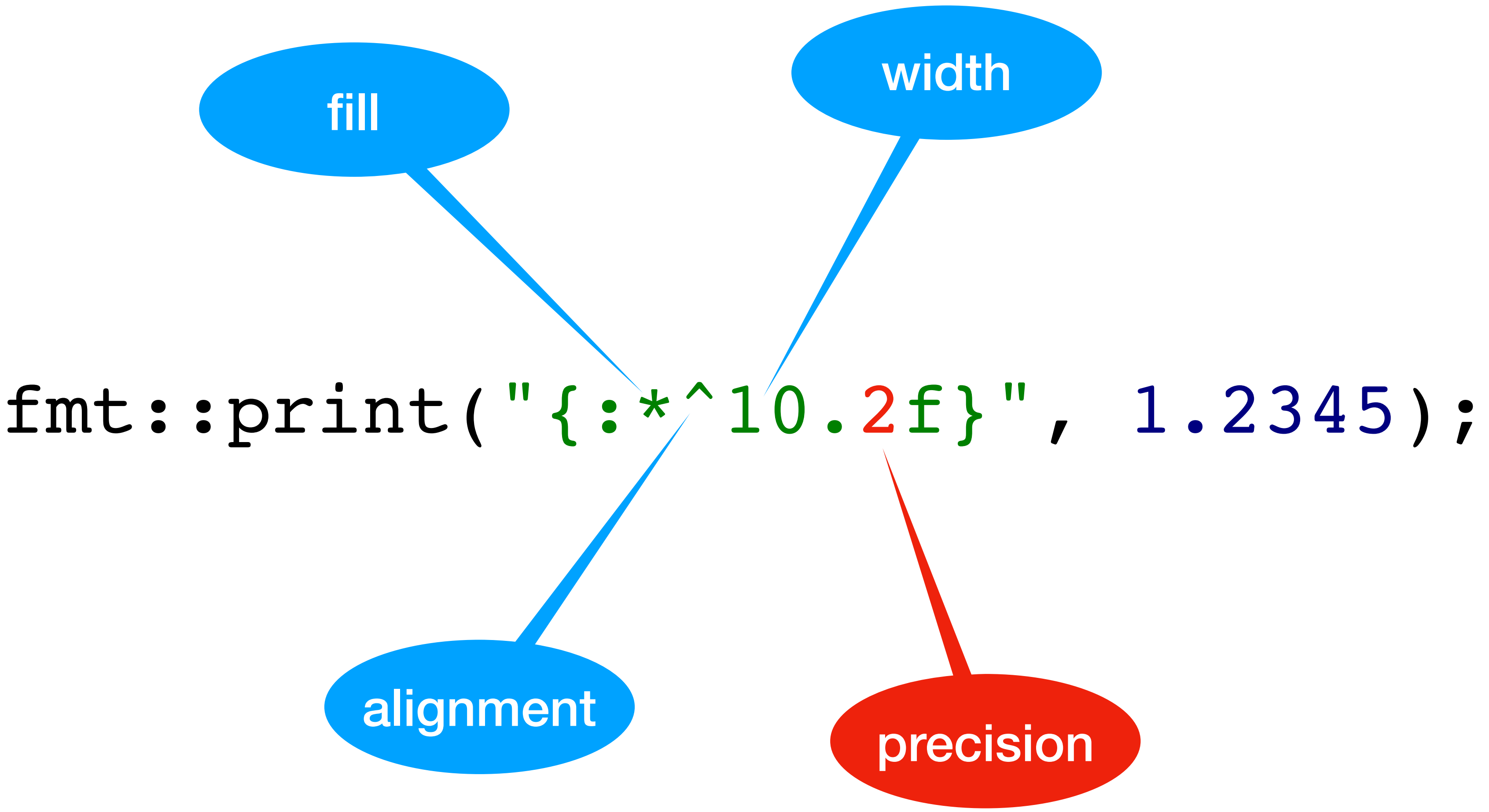
```
fmt::print("{:*^10.2f}", 1.2345);
```

fill

width

alignment

Mini-language



fill

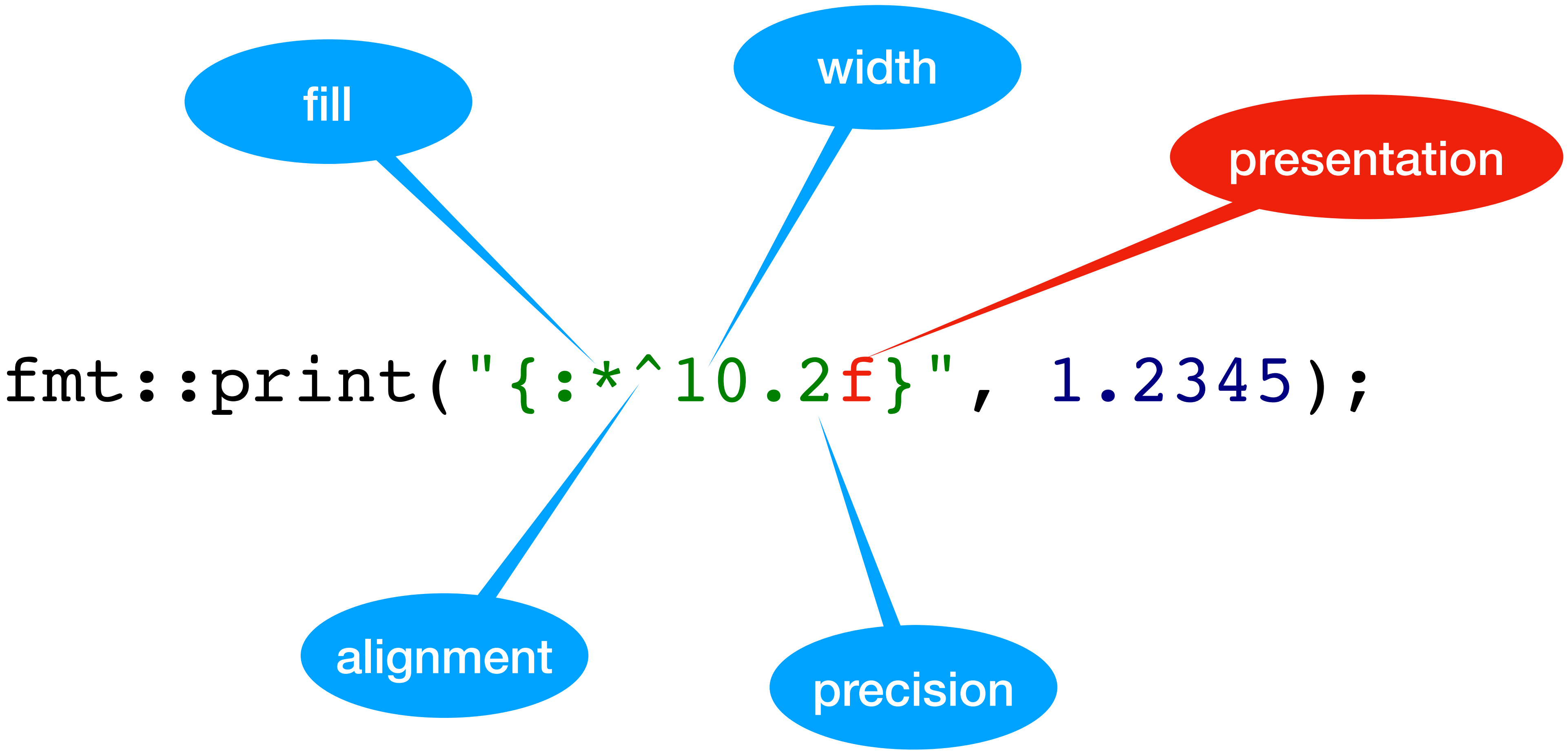
width

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

alignment

precision

Mini-language



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

fill

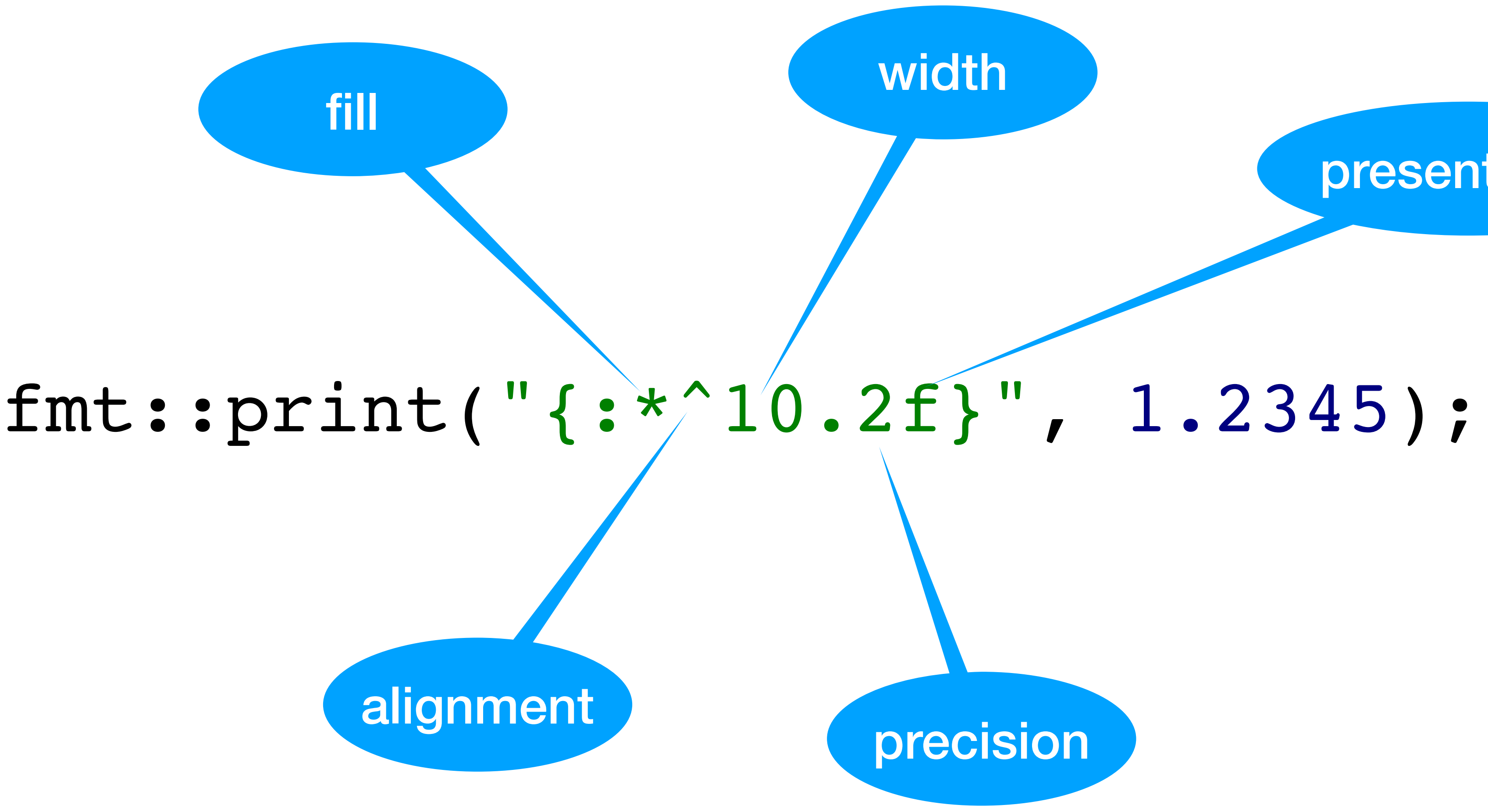
width

presentation

alignment

precision

Mini-language



```
fmt::print("{:*^10.2f}", 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

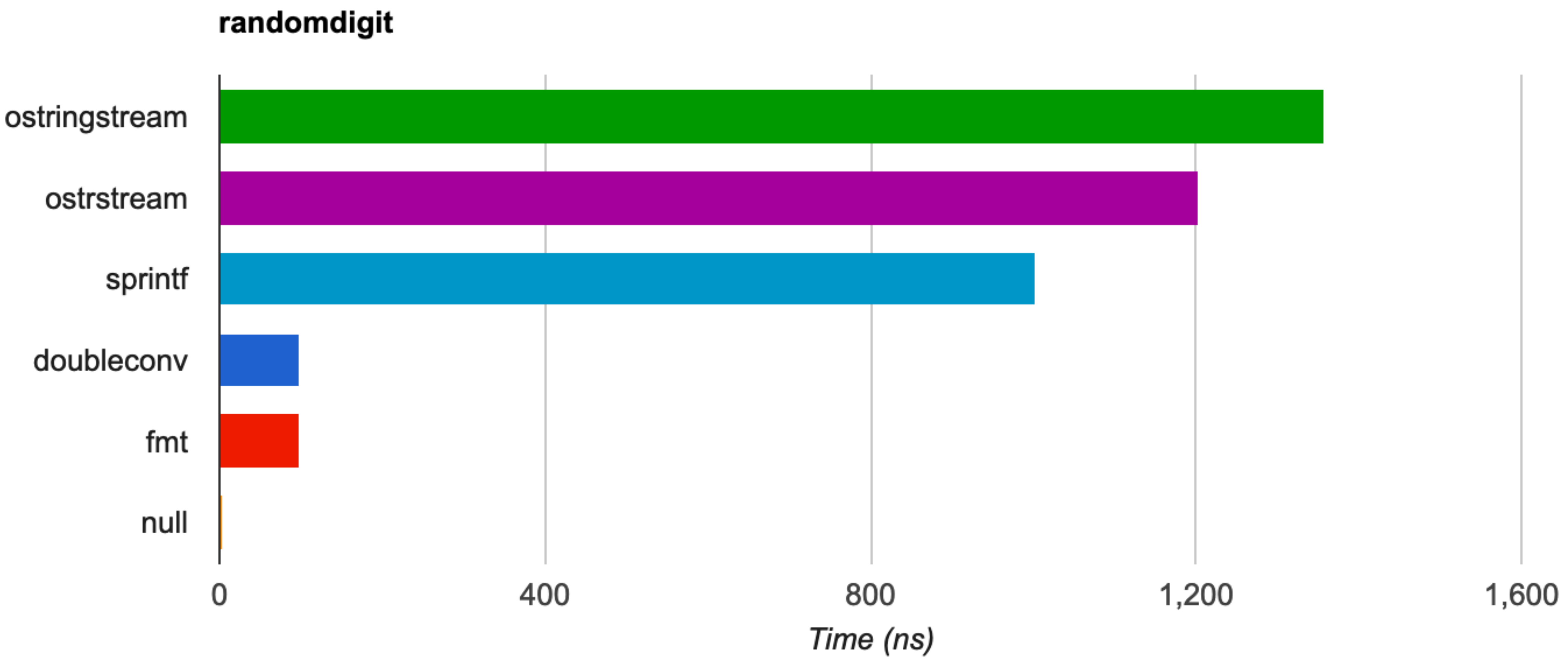
- Dynamic memory allocations can be completely avoided & in particular the default will never allocate.
- No allocation & no need to specify buffer size:

```
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`):

```
std::string s;
fmt::format_to(std::back_inserter(s), "{}", 1.2345);
```

🍎 vs 🍊



Roundtrip precision: https://github.com/fmtlib/dtoa-benchmark
(based on miloyip/dtoa-benchmark)

vs 🍊

Function	Time (ns)	Speedup
ostringstream	1,356.700	1.00x
ostrstream	1,202.847	1.13x
sprintf	1,002.506	1.35x
doubleconv	97.071	13.98x
fmt	96.071	14.12x
null	1.324	1,025.06x

Still a lot of optimization opportunities in fmt.

References 🧐

- David W. Matula. 1968. *In-and-out conversions*. Communications of the ACM. Volume 11 Issue 1, Jan. 1968, 47-50.

- Guy L. Steele Jr. and Jon L. White. 1990. *How to Print Floating-Point Numbers Accurately*. In Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI ’90). ACM, New York, NY, USA, 112-126.

- Florian Loitsch. 2010. *Printing Floating-Point Numbers Quickly and Accurately with Integers*. In Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation, PLDI 2010. ACM, New York, NY, USA, 233-243.

- Ulf Adams. 2018. *Ryū: fast float-to-string conversion*. In Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018. ACM, New York, NY, USA, 270-282.

- {fmt}: https://github.com/fmtlib/fmt

Questions?

🤷‍♂️