

+ 25

Using Floating-point

What Works, What Breaks, and Why

EGOR SUVOROV



20
25



Using Floating-point: What Works, What Breaks, and Why

CppCon 2025
September 16, 2025

Egor Suvorov
Senior Software Engineer

TechAtBloomberg.com

© 2025 Bloomberg Finance L.P. All rights reserved.

Bloomberg
Engineering

Scope

We focus on binary floating-point numbers like `float`, `double`

- Math vs. standards vs. C++ implementations

Adjacent: Decimal floating-point numbers (`libdfp`, `bdldfp`, N3871)

- Might be more intuitive and give human-readable results
- Some floating-point issues still apply
- Much slower, not the default, unpopular

Not covered:

- Fixed-point arithmetic (binary or decimal)
- Arbitrary precision libraries: GNU GMP, GNU MPFR
- Interval arithmetic: GNU MPFI

Bloomberg

Engineering



Floating-point types in C++

Bloomberg

Engineering

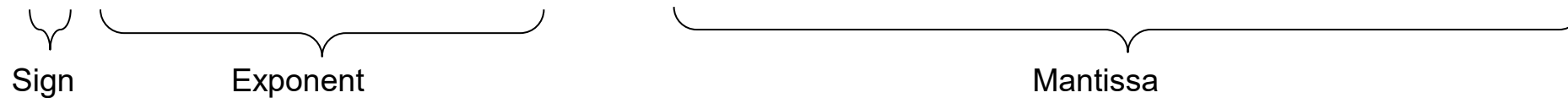
Types in C++

- Standard floating-point-types
 - `float` \subseteq `double` \subseteq `long double`
- Extended floating-point-types, e.g.,
 - GCC extensions like `__float80`
 - Optional since C++23:
 - `std::float_16t`
 - `std::float_32t`
 - `std::float_64t`
 - `std::float_128t`
- Implementation-defined [basic.fundamental]/12
 - Typically IEEE-754/IEC 60559 from hardware, today's focus
 - `std::numeric_limits<float>::is_iec559 == true`
 - ~~Bugs and omissions~~ tradeoffs are everywhere
 - Might feel like UB: “works on my machine”

sizeof	GCC	MSVC
float	4	4
double	8	8
long double	10	8
std::float128_t	16	N/A

“Normal” binary floating-point numbers

- (double) 123.75 = $495/4 = 0b1'1110'1111 / 0b100 =$
- = $0b\ 1'1110'11.11 = 0b\ 1.1110'1111e+6$
- Exponent is stored “biased”, not “two complement”
 - $6+1023=1029$
- $0|100\ 0000\ 0101|(1.)1110\ 1111\ 0000\ 0000\dots$

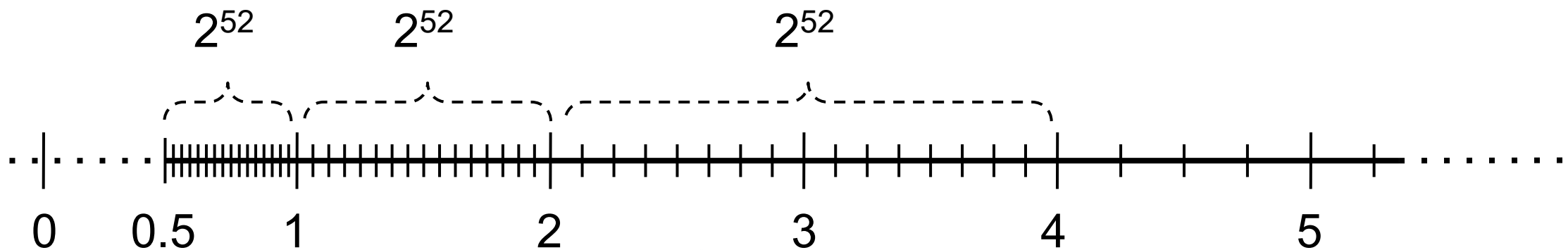


Type (x86 GCC)	float	double	long double
Bits (bytes)	32 (4)	64 (8)	80 (10), may be 64 (8)
Sign bits	1	1	1
Exponent bits	8	11	15
Mantissa bits (stored)	23	52	63
Range	Should not matter		

Precision changes with magnitude

```
cout << std::nextafter(1, 2)          - 1;      // 2.22045e-16
cout << std::nextafter(1e9, 1e10)     - 1e9;    // 1.19209e-07
```

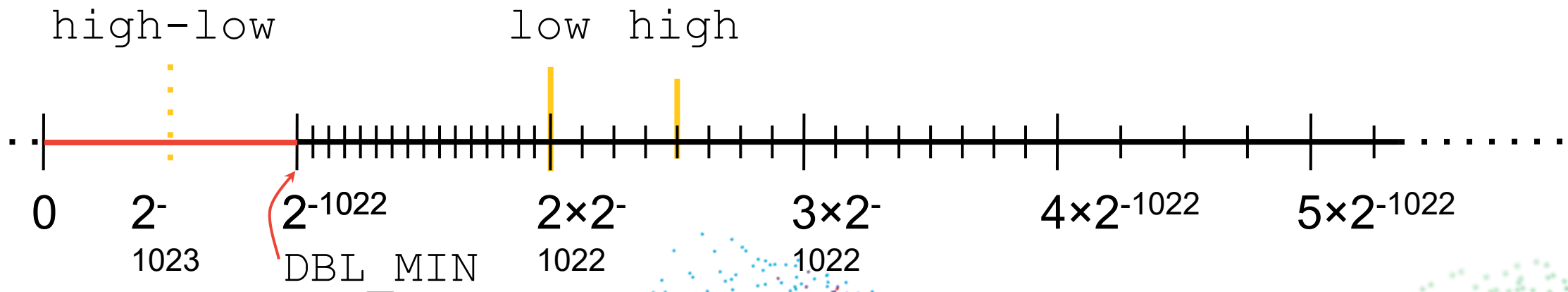
- We focus on `double` as an example
- *Relative* precision is mostly the same
- Bigger numbers \Rightarrow less *absolute* precision
- Gamers have fun: https://minecraft.fandom.com/wiki/Distance_effects



Underflow

- Exponent is in $[-1022; 1023]$
- Minimal possible number is $1.0 \times 2^{-1022} = \text{DBL_MIN}$
- Get division by zero in $(\text{mid} - \text{low}) / (\text{high} - \text{low})$

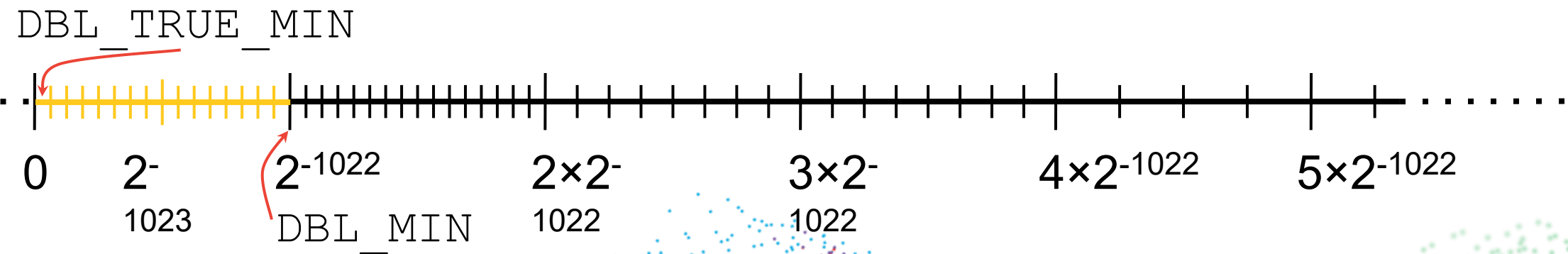
$$\begin{aligned} & 0b1.010\dots00000e-1021 \\ - & 0b1.000\dots00000e-1021 \\ = & 0b0.010000000e-1021 = 2^{-1023} \approx 0 \end{aligned}$$



“Subnormal” numbers (gradual underflow)

- Invariant: $(x \neq y) \Leftrightarrow (x - y) \neq 0$

	Sign	Exponent	Mantissa	Name
Normal (smallest)	0	000 0000 0001 1-1023= -1022	(1.) 0000...0000	DBL_MIN/FLT_MIN numeric_limits::min
Subnormal (largest)	0	000 0000 0000	(0.) 1111...1111	
Subnormal (smallest)	0	000 0000 0000	(0.) 0000...0001	DBL_TRUE_MIN/FLT_TRUE_MIN numeric_limits::denorm_min




Special Values: Infinities (overflow)

	A C++ name	Sign	Exponent	Mantissa
$+\infty$	INFINITY	0	111 1111 1111	0000...0000
$-\infty$	-INFINITY	1	111 1111 1111	0000...0000

- Checks like `a * b > c` work even if `a * b` overflows
 - `(b == 0 && 0 > c) || (a * sign(b) > c / fabs(b))`
- Allegedly: simpler code, less branching, more vectorization


```
cout << format("{} {} \n {} {} \n {} {} \n",
    10, atan(10),          // 10 1.4711276743037347
    1e400, atan(1e400)     // inf 1.5707963267948966
    exp(-1e400), log(0)    // 0 -inf
);
```



Special Values: Signed zero

	A C++ name	Sign	Exponent	Mantissa
+0 (0)	0.0	0	000 0000 0000	0000...0000
-0	-0.0	1	000 0000 0000	0000...0000

```
const float pz = 0.0, nz = -0.0;
cout << format("{} {} {} \n {} \n {} {} \n {} {} \n",
    pz, nz, pz == nz,           // 0 -0 true
    2 - 2,                     // 0
    1 / (1 / 1e500), 1 / pz,   // inf inf
    1 / (-1 / 1e500), 1 / nz  // -inf -inf
);
```



Special Values: NaN (not a number)

	Sign	Exponent	Mantissa
$2^{53}-2$ NaNs	?	111 1111 1111	????...???? (except all 0)

```
cout << format("{} {} \{} {} \n{} {} \n{} {} \n",
    sqrt(-0.1),      sqrt(-0.1) + 0.0,      // -nan -nan
    0.0 * INFINITY,  INFINITY - INFINITY,    // -nan -nan
    0.1 * INFINITY,  INFINITY + INFINITY,    // inf inf
);
```

- Appears after most “invalid” operations
- *Most* operations with NaN yield NaN
- Payload (including sign) is arbitrary



Rounding

Bloomberg

Engineering

Rounding strategies

Number	Toward 0 (trunc)	Toward $-\infty$ (floor)	Toward $+\infty$ (ceil)	To nearest (round)
-1.7	-1.0	-2.0	-1.0	-2.0
-1.5	-1.0	-2.0	-1.0	-2.0 or -1.0
-1.2	-1.0	-2.0	-1.0	-1.0
1.2	1.0	1.0	2.0	1.0
1.5	1.0	1.0	2.0	1.0 or 2.0
1.7	1.0	1.0	2.0	2.0

- **Later on:** rounding to X non-radix digits (0.123 to 0.12 in binary)
- Not included: away from 0
- “To nearest” needs a tie-break strategy
- Tie-break can only happen if radix is even: 2, 10

Rounding strategies: to nearest tie breaks

Number	Half toward 0	Half toward $-\infty$ (down)	Half toward $+\infty$ (up)	Half away from 0 (primary school)	Half to even (bankers')
-2.5	-2.0	-3.0	-2.0	-3.0	-2.0
-1.5	-1.0	-2.0	-1.0	-2.0	-2.0
-0.5	0.0	-1.0	0.0	-1.0	0.0
0.5	0.0	0.0	1.0	1.0	0.0
1.5	1.0	1.0	2.0	2.0	2.0
2.5	2.0	2.0	3.0	3.0	2.0
Sum=0.0	0	-1.0	1.0	0.0	0.0
Sum(>0)=4.5	3.0	3.0	6.0	6.0	4.0

- Not included: half to odd
- Half to even (radix 2) is the default in IEEE-754

Bloomberg

Engineering

Decimal to binary

0.1 only exists in math, not IEEE-754 in C++

$$0.1F ==$$

0.100000001490116119384765625000000000000000000000000000000

```
(double) 0.1 ==
```

[illegible]

- Differentiate “math” 0.1 and “double” 0.1.
- Default rounding: to nearest representable, tie-to-even
- Not necessarily upward:

```
(double) 0.3 ==
```

0.29999999999999998889776975374843459576368331909179687500

- **Bonus points: where is the border between `DBL_MAX` and `INFINITY`?**

Compilation: round to nearest, half to even

“Even” means “last bit of mantissa is zero”:

```
A = 0.1000000001490116119384765625000F = 0|01111011|100110011001100110011001101
B = 0.10000000008940696716308593750000F = 0|01111011|10011001100110011001110
C = 0.10000000016391277313232421875000F = 0|01111011|10011001100110011001111
```

Compiler rounds all literals, possibly losing precision:

```
cout << format("{:.30f}\n{:.30f}\n{:.30f}\n{:.30f}\n{:.30f}\n",
    0.1000000001490116119384765625000F, // A 0.1000000001490116119384765625000
    0.10000000005215406417846679687500F, // x 0.10000000008940696716308593750000
    0.10000000008940696716308593750000F, // B 0.10000000008940696716308593750000
    0.10000000012665987014770507812500F, // y 0.10000000008940696716308593750000
    0.10000000016391277313232421875000F // C 0.10000000016391277313232421875000
);
```



Compilation: round to nearest, half to even (2)

“Even” means “last bit of mantissa is zero”:

```
A = 0.1000000001490116119384765625000F = 0|01111011|100110011001100110011001101
B = 0.10000000008940696716308593750000F = 0|01111011|10011001100110011001110
C = 0.10000000016391277313232421875000F = 0|01111011|10011001100110011001111
```

Compiler rounds all literals, possibly losing precision

```
cout << format("{:.30f}\n{:.30f}\n{:.30f}\n{:.30f}\n{:.30f}\n",
    0.1000000001490116119384765625000F, // A 0.1000000001490116119384765625000
    0.10000000005215406417846679687500F, // x 0.10000000008940696716308593750000
    0.10000000008940696716308593750000F, // B 0.10000000008940696716308593750000
    0.10000000012665987014770507812501F, // y 0.10000000016391277313232421875000
    0.10000000016391277313232421875000F // C 0.10000000016391277313232421875000
);
```

Runtime: default

Perform with infinite precision first, then round:

```
0.100000000000000000000055511151231257827021181583404541015625
+0.2000000000000000000000111022302462515654042363166809082031250
=0.30000000000000000000000166533453693773481063544750213623046875
```

Round to nearest. Options are equidistant, half to even bit (here: up)

```
~0.29999999999999999999999888977697537484345957636833190917968750
~0.30000000000000000000000444089209850062616169452667236328125000
0.0000000000000000000000277555756156289135105907917022705078125
```

Two “errors” when representing 0.1 and 0.2 added up:

```
(double) 0.3 == 0.299999999999999999999998..
```

```
(double) 0.3 != (double) 0.1 + (double) 0.2
```



Reproducible results

Bloomberg

Engineering

Not always reproducible

- https://gcc.gnu.org/bugs/#nonbugs_general

```
auto div = [] (double a, double b) { return a / b; };  
  
double a = 0.5;    // 0.5 exactly  
double b = 0.01;   // 0.010000000000000000000000208...  
  
cout << (int) (a / b) << "\n"           // 49 or 50, rounding?  
      << (int) div(a, b) << "\n";      // same?
```

<https://godbolt.org/z/KTMv7na6v>

Not always reproducible (2)

- https://gcc.gnu.org/bugs/#nonbugs_general

```
auto div = [] (double a, double b) { return a / b; };  
  
double a = 0.5;    // 0.5 exactly  
double b = 0.01;   // 0.010000000000000000000000208...  
  
cout << (int) (a / b) << "\n"           // 49 or 50  
      << (int) div(a, b) << "\n";      // 50
```

- GCC prints 49 50 with `-m32 -std=c++20`, but not `-std=gnu++20`!
- MSVC/Clang print 50 50.
- Affected by simple refactoring, feels weird to debug

Excess precision

(double) 0.01 = 0.0100000000000000000000020816681711721685132...
0.5 / 0.0100000000000000000000020816681711721685132... =
49.99999999999999999895916591... =

1 10001.1111'1111'1111'1111'1111'1111'1111'1111'
1111'1111'1111'1111'1'1011'01...

- Have to round to double with 52 bits of mantissa, either:

110010.0000'0000'0000'0000'0000'0000'0000'0000'
0000'0000'0000'0000'0'0000'00...

110001.1111'1111'1111'1111'1111'1111'1111'1111'
1111'1111'1111'1111'0'0000'00...

- With 52 mantissa bits, 50 is closer.

Excess precision

(double) 0.01 = 0.0100000000000000000000020816681711721685132...
0.5 / 0.010000000000000000000000020816681711721685132... =
49.99999999999999999895916591... =

1 10001.1111'1111'1111'1111'1111'1111'1111'1111'
1111'1111'1111'1111'1011'01...

- Have to round to double with 52 bits of mantissa, either:

110010.0000'0000'0000'0000'0000'0000'0000'0000'
0000'0000'0000'0000'0000'00...

110001.1111'1111'1111'1111'1111'1111'1111'1111'
1111'1111'1111'1111'0000'00...

- With 53 mantissa bits, 50 is closer.

Excess precision

(double) 0.01 = 0.0100000000000000000000020816681711721685132...
0.5 / 0.010000000000000000000000020816681711721685132... =
49.99999999999999999895916591... =

1 10001.1111'1111'1111'1111'1111'1111'1111'1111'
1111'1111'1111'1111'1011'01...

- Have to round to double with 52 bits of mantissa, either:

110010.0000'0000'0000'0000'0000'0000'0000'0000'
0000'0000'0000'0000'0000'00...

110001.1111'1111'1111'1111'1111'1111'1111'1111'
1111'1111'1111'1111'0000'00...

- With 54+ mantissa bits, 49.999... is closer.

Bug 323: optimized code gives strange floating point results

- `(double) a / b` is 50.0
- `(long double) a / b` is 49.9999
- 64-bit GCC uses SSE and 64-bit registers
- 32-bit GCC uses x87 and 80-bit registers
 - No need to truncate 80-bit register to 64 bits after every operation
 - Hence, we get extra precision for free
- Excess precision for intermediate calculations => different decision
- Very old issue, reported 25 years ago
 - Me, 15 years ago: <https://codeforces.com/blog/entry/1059>
 - Ptilouk, “[How I joined the bug 323 community](#)”
- `-fexcess-precision=standard` should help, but it does not?

Other issues

Cross-Platform Floating-Point Determinism, Sherry Ignatchenko, CppCon 2024

- Bad optimizations, see <https://gcc.gnu.org/wiki/FloatingPointMath>
 - $-(1/3) == 1/(-3)$ is incorrect when rounding toward infinities
 - Disabled by `-frounding-math`
- See `-ffast-math` that incorrectly assumes
 - $(a * c) + (b * c) == (a + b) * c$
- Excess precision
- Fused Multiply-Add (FMA)
 - $a + (b * c)$ as a single opcode, faster, infinite intermediate precision
- Hence, trouble implementing, e.g., Kahan summation algorithm

Precision guarantees

- Max precision and correct rounding should be guaranteed
 - `+` `-` `/` `*` `sqrt`
- `sin(1e100)`
 - How to calculate fast?
 - Do the result even mean anything?

Most standard libraries agree
- `pow(a, b)`
 - Mathematically equal to `exp(ln(a) * b)`
 - Ten years ago: `(int)pow(10, 2) == 99`
 - <https://codeforces.com/blog/entry/21844>

Is pow(int, int) precise?

```
auto mypow = [](int a, int b) { double result = 1;
                                while (b --> 0) result *= a;
                                return result; };

int bad = 0, total = 0;
for (int a = 1; a < 100; ++a)
    for (int b = 0; b < 100 && mypow(a, b) < 1e18; ++b) {
        ++total;
        if (pow(a, b) != mypow(a, b))
            bad++, cout << format("{} {}: {} {}\n", a, b, pow(a, b), mypow(a, b));
    }

cout << format("{:.2f}% failed\n", bad * 100.0 / total);
```

pow(int, int) is not precise

```
auto mypow = [](int a, int b) { double result = 1;
                                while (b --> 0) result *= a;
                                return result; };
```

Example output (MSVC, 32-bit):

7 19: 11398895185373142 11398895185373144
7 21: 558545864083284032 558545864083283968

Compiler	32-bit	64-bit
GCC/libstdc++	0.79%	1.15%
MSVC/Microsoft STL	0.94%	0.87%

Printing numbers

Bloomberg

Engineering

Default output

```
cout << 1.23 << "\n"; // like %g: 1.23 , no unnecessary zeros
printf("%g\n%g\n%g\n%g\n%g\n%g\n%g\n%d\n",
12.34567,           // 12.3457           default precision is 6
123456.7,           // 123457           keep the default precision
0.0001234567,       // 0.000123457
0.00001234567,      // 1.23457e-05      scientific for exponent <= -5
1234567.0,           // 1.23457e+06      scientific for exponent >= precision
1000000.0,           // 1e+06            do not print unnecessary zeros
0.1 + 0.2,           // 0.3              despite not being exactly 0.3D
0.1 + 0.2 == 0.3     // 0                hence: not great for debugging
);
```

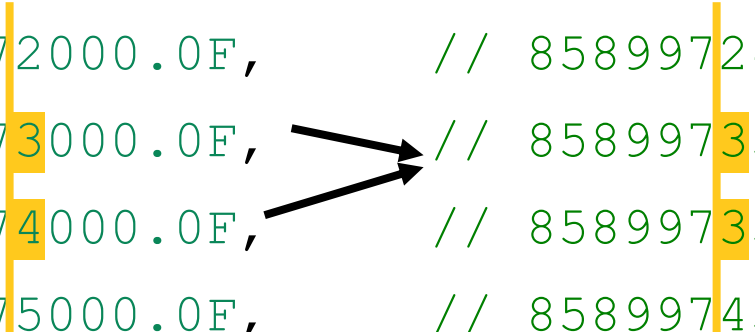
std::numeric_limits<T>::max_digits10

```
static_assert(std::numeric_limits<float>::max_digits10 == 9);  
printf("%.10f\n%.8g\n%.10f\n%.8g\n",  
123456.156F,    // 123456.1562500000000000000000000000000000  
123456.156F,    // 123456.16  
123456.164F,    // 123456.1640625000000000000000000000000000  
123456.164F     // 123456.16  
);
```

- 9 significant digits are always enough to uniquely identify a float
- 8 digits are sometimes not enough
- Not obvious: 23 mantissa bits specify 8'388'608 values (7 digits)
 - There are 128 floats in [123456; 123457) - some will round to the same 2 digits
- `>= max_digits10` is the “lower bound” for float-string-float roundtrips

std::numeric_limits<T>::digits10

```
static_assert(std::numeric_limits<float>::digits10 == 6);  
printf("%.30f\n%.2f\n%.2f\n%.2f\n%.2f\n%.10e\n%.10e\n",  
123.126F,           // 123.12599945068359375000000000000000  
8589972000.0F,      // 8589972480.00  
8589973000.0F,      // 8589973504.00  
8589974000.0F,      // 8589973504.00  
8589975000.0F,      // 8589974528.00  
FLT_MIN,            // 1.1754943508e-38  
1.23456E-43F);      // 1.2331426486e-43 - underflow, subnormal number
```



- 6 significant digits can always be stored exactly, 7 not always
 - Fail among 10-digit integers (0.3%) and 13-digit integers (0.5%)
- `<= digits10` is the “upper bound” for string-float-string roundtrips

Best way to print

```
cout << format("{}\n{}\n{}\n{}\n",  
0.1,           // 0.1  
0.2,           // 0.2  
0.299999999999999998, // 0.3, prints the shortest representation  
0.1 + 0.2      // 0.3000000000000000004 - all 17 digits needed  
               // numeric_limits<double>::max_digits10 == 17  
);
```

Special values

- Case-insensitive
- `INF`, `+INF`, `-INF`, `INFINITY`, `+INFINITY`, `-INFINITY`
- `NAN`, `+NAN`, `-NAN`, `NAN(str1)`, `+NAN(234)`, `-NAN(x)`
- My GCC just prints `inf`, `-inf`, `nan` or `-nan`
- My Visual Studio (may add a sign):
 - `nan` — “quiet NaN”
 - `nan(snan)` — “signaling nan”
 - `nan(ind)` — “indefinite” value
- Before Visual Studio 2015 (may add a sign):
 - `1.#INF123456`
 - `0.#IND123456` — “indefinite” value, quiet NaN
 - `0.#NAN123456` — NaNa

Special values — portability

```
cout << sqrt(-1) << "\n";
```

- Build with 32-bit Visual Studio for x86
- Executed on a usual x86: `-nan(ind)`
- Executed on Windows ARM (macOS+Parallels): `nan`
- Probably because x86 and ARM use different NaN payload



NaNs are special

Bloomberg

Engineering

NaN comparisons

```
cout << ( 0 == nan1) << (0 != nan1) // 01
      << ( 0 < nan1) << (0 > nan1) // 00
      << (nan1 == nan2) << (nan1 != nan2) // 01
      << (nan1 < nan2) << (nan1 > nan2); // 00
```

- NaN is not equal to, greater than, or less than anything.
- Not equal to itself – easier to adopt before `isnan` appears

```
cout << (nan1 == nan1) << (nan1 != nan1) // 01
      << (nan1 < nan1) << (nan1 > nan1); // 00
```

```
bool isnan(double x) { return x != x; }
```



NaN comparisons

```
double x[] {3, 1, NAN, 4, 2};  sort(begin(x), end(x));  
for (double v : x) cout << v << " ";  // 1 3 nan 2 4
```

- [alg.sorting.general]/3: “comp shall induce a strict weak ordering”
- [alg.sorting.general]/4.5: “it can be shown that ... is a strict total ordering”
- UB for `std::sort`, for `std::set...`, for in-house algorithms as well
 - <https://stackoverflow.com/questions/18291620/why-will-stdsort-crash-if-the-comparison>
- IEEE-754’s “totalOrder”: $-\text{NaN} < -0.0 < +0.0 < +\infty < +\text{NaN}$
 - Comparison of NaNs (except sign) is implementation-defined



NaN payload

- We have 1 sign bit and 52 mantissa bits
- Initial payload is not specified:


```
cout << format("{:x}\n", bit_cast<uint64_t>(sqrt(-0.1)));  
// fff8000000000000 (x86)  
// 7ff8000000000000 (ARM64)
```

- Historical idea: debug information about the origin of NaN
- “Quiet NaN” and “Signaling NaN”
 - Please share if you use them!
- Can put a x86-64 (48 bits) pointer and/or a string: `bld_datum`



NaN payload propagation

```
double nan1 = bit_cast<double>(0x7ff800000000000010);  
double nan2 = bit_cast<double>(0x7ff800000000000020);  
#define bits bit_cast<uint64_t>  
cout << format("{:x}\n{:x}\n{:x}\n{:x}\n{:x}\n",  
    bits(nan1 + nan2),           // 7ff800000000000010 (either)  
    bits(sqrt(nan1)),           // 7ff800000000000010  
    bits(std::nan("2050")),     // 7ff8000000000000802 (GCC)  
                                // 7ff8000000000000000 (MSVC)  
);
```



Special Values: NaN (not a number) — Corner cases

```
double x = INFINITY, y = nan("");  
cout << sqrt(x * x + y * y)    // nan  
      << hypot(x, y);         // inf
```

- $\text{hypot}(+\infty, y) = +\infty$ for all $y \Rightarrow$ for NaN too.
- Might be useful.
- Naive implementation may be incorrect.
- Non-naive implementation may be slower.



Math caveats

Bloomberg

Engineering

Even simple operations might lose precision

```
double a = 14468371.32764871;  
double b = 2858973.58793243;  
double c = 3938402.12874453;  
  
cout << format("{}\n{}\n",  
    (2*a+2*b+2*c) / (a+b+c),    // 2  
    (3*a+3*b+3*c) / (a+b+c)    // 2.999999999999999996  
);  
  
assert((3*a+3*b+3*c) / (a+b+c) == 3);    // FAILS
```

Naive approach: assume close numbers are equal

- Slightly incorrect intuition: x is actually $(x - \text{eps}, x + \text{eps})$
- Not `DBL_EPSILON` for previous slide!

<code>a == b</code>	<code>fabs(a - b) < eps</code>
<code>a <= b</code>	<code>a <= b + eps</code> (or <code><</code>)
<code>a < b</code>	<code>a < b - eps</code> (or <code><=</code>)
<code>floor(a)</code>	<code>floor(a+eps)</code>
<code>round(a)</code>	N/A , needs tie-break
<code>sqrt(x)</code>	sqrt (<code>max(x, 0)</code>)
<code>set<double, less<>></code>	<code>set<double, less<>></code>

Avoid unstable functions

Before	After
<code>acos (x/sqrt (x*x+y*y)) *sign (y)</code>	<code>atan2 (y, x)</code>
<code>pow (a, 3)</code>	<code>a * a * a</code>
<code>a * (1.0 / b)</code>	<code>a / b</code>
Heron's formula for triangle area <code>sqrt (s* (s-a) * (s-b) * (s-c))</code>	Vector product <code>fabs (x1*y2-x2*y1) / 2</code>

<https://randomascii.wordpress.com/2012/02/13/dont-store-that-in-a-float/>



Rounding to decimal places

- `ceil()/floor()` can be expressed via `round()`
- Do we care about ties? Banking – yes, simulations – no.
- The default is “round to nearest, tie to even”, but inputs are not precise:

```
cout << format("{0:.2f} {0:.30f}\n...",
    1.105, // 1.10 1.10499999999999999982236431605997 down
    1.115, // 1.11 1.11499999999999999991118215802999 down
    1.125, // 1.12 1.125000000000000000000000000000000000000000000000000 down (tie)
    1.135, // 1.14 1.135000000000000000000000000000000000000000000000000 up
    1.145, // 1.15 1.145000000000000000000000000000000000000000000000000 up
    1.155, // 1.16 1.155000000000000000000000000000000000000000000000000 up
);
```

Rounding to decimal places — still incorrect

```
auto rnd = [] (double x) { cout << format("{} {:.20f}\n",  
    round(x * 100) / 100, x * 100); // away from 0  
};
```

```
rnd(1.015); // 1.01 101.499999999999998578915 down
```

```
rnd(1.025); // 1.02 102.499999999999998578915 down
```

```
rnd(1.035); // 1.03 103.499999999999998578915 down
```

```
rnd(1.045); // 1.05 104.500000000000000000000000 up
```

```
rnd(1.055); // 1.06 105.500000000000000000000000 up
```

- Only rounds down ~7 times for 1.005-1.995
- Works for 1.0005-1.9995



Rounding to decimal places — round away from 0

```
auto rnd = [] (double x) {  
    long long y = llround(x * 1000); // TODO: overflow  
    y += -(y % 10) + (y % 10 >= 5 ? 10 : 0);  
    // TODO: are there enough digits in `double`?  
    cout << format("{} {:.20f}\n", y / 1000.0, y / 1000.0);  
};  
  
rnd(1.105); // 1.11 1.110000000000000000009770  
rnd(1.115); // 1.12 1.1200000000000000000010658  
rnd(1.125); // 1.13 1.129999999999999999989342  
rnd(1.135); // 1.14 1.139999999999999999990230
```



C++ caveats

TechAtBloomberg.com

© 2025 Bloomberg Finance L.P. All rights reserved.

abs - Why?

```
#include <cmath>
#include <iostream>
#include <vector>

int main() {
    std::vector<double> vals{-0.5, 1.5, -2.5};
    for (double x : vals)
        if (abs(x) >= 0.1)
            std::cout << x << " ";
}
```

- Visual Studio: -0.5 1.5 -2.5
- GCC/Clang: 1.5 -2.5

abs - because

```
#include <cmath>
#include <iostream>
#include <vector>

int main() {
    std::vector<double> vals{-0.5, 1.5, -2.5};
    for (double x : vals)
        if (abs(x) >= 0.1)
            std::cout << x << " ";
}

int abs(int j); // (int)-0.5 == 0
namespace std { float abs(float j); }
```

abs - Solution (1)

```
#include <cmath>
#include <iostream>
#include <vector>

int main() {
    std::vector<double> vals{-0.5, 1.5, -2.5};
    for (double x : vals)
        if (std::abs(x) >= 0.1)
            std::cout << x << " ";
}

int abs(int j); // (int)-0.5 == 0
namespace std { float abs(float j); }
```

abs - Solution (2)

```
#include <math.h>
#include <iostream>
#include <vector>

int main() {
    std::vector<double> vals{-0.5, 1.5, -2.5};
    for (double x : vals)
        if (abs(x) >= 0.1)
            std::cout << x << " ";
}

int abs(int j);
float abs(float j);
```

abs - Solution (3)

```
#include <cmath>
#include <iostream>
#include <vector>

int main() {
    std::vector<double> vals{-0.5, 1.5, -2.5};
    for (double x : vals)
        if (fabs(x) >= 0.1)
            std::cout << x << " ";
}

float fabs(float j);
```

Float-to-integer-conversion

[conv.fpoint]/1: The conversion truncates; ... The behavior is undefined if the truncated value cannot be represented...

```
cout << (int)123.6 << "\n";
```

```
cout << (int)-123.6 << "\n";
```

```
cout << (int)1e10 << "\n";    //  2147483647  (GCC 14)
                                // -2147483648  (GCC 15)
                                //  1410065408  (MSVC)
                                //  0x2'540B'E400      0x540B'E400  (MSVC)
                                //  2085281808  random (Clang)
```

Integer-to-float conversion


```
long long orig          = 0x1234'5678'1234'5F7F;  
double dbl = orig;  
long long rt = dbl;  
cout << format("{:a}\n", dbl); // 1.234567812345fp+60  
cout << format("{:x}\n", rt);  // 1234567812345f00
```

[conv.fprint]/2: The result is exact if possible. If ... can be represented but ... cannot be represented exactly, it is an **implementation-defined choice** of either the next lower or higher representable value.

- [GCC](#), [MSVC](#) claim to round to the nearest, no tiebreaker
- [Clang](#) does not say

Subnormals can be magnitude(s) slower

```
static void Normal(benchmark::State& state) {  
    for (auto _ : state) {  
        double s = 1e-100; s /= 1.1;  
        benchmark::DoNotOptimize(s);  
    }  
}  
BENCHMARK(Normal); // 2.01ns on an Intel Xeon VM with -O0  
  
static void Subnormal(benchmark::State& state) {  
    for (auto _ : state) {  
        double s = 1e-100; s /= 1e-310;  
        benchmark::DoNotOptimize(s);  
    }  
}  
BENCHMARK(Subnormal); // 50.4ns (x25 slower)
```



Subnormals can be magnitude(s) slower (2)

MATLAB tells about a typical x5 slowdown (up to x50):

<https://www.mathworks.com/help/rtw/ug/subnormal-number-performance.html>

“On Subnormal Floating Point and Abnormal Timing” (2015)

- JavaScript timing attack: extract <iframe> ~16.4 pixels/second

Alternative: flush-to-zero (FTZ) + denormals-are-zero (DAZ)

- `if (abs(x) < DBL_MIN) x = 0;`
- Opt-in for some CPUs (x86, ARM), GPUs and compilers
- Do not blindly disable; get rid of subnormals

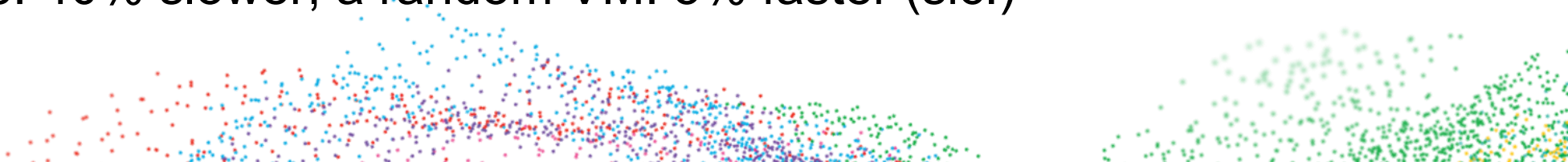
<https://randomascii.wordpress.com/2012/05/20/thats-not-normalthe-performance-of-odd-floats/>



Complex numbers

- C99: `float _Complex`, has its own “inf” and “NaN”
- C++: `std::complex<float>`, should be compatible
- Might be 50% slower with Clang/libstdc++/-O3 than naive rewrite:

```
static void Standard(benchmark::State& state) {  
    for (auto _ : state) {  
        std::complex<double> x{1, 2};  
        benchmark::DoNotOptimize(x);  
        x = x * std::complex<double>{3, 4};  
        benchmark::DoNotOptimize(x);  
    }  
}
```

- Might call C functions instead of inlining (libc++, 300% slower)
 - Local machine: 40% slower; a random VM: 5% faster (sic!)
- 

Case study

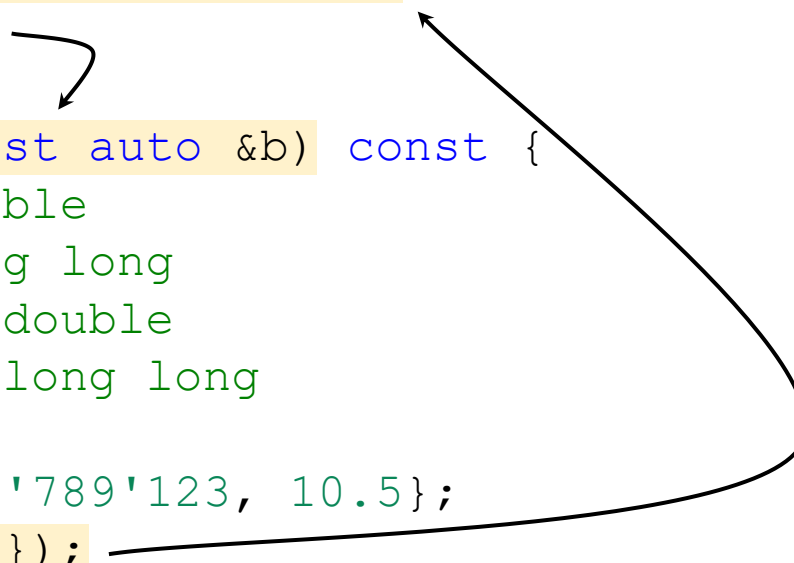
Bloomberg

Engineering

Sorting a heterogenous list

```
using Data = variant<long long, double>; // bools, strings, arrays...
struct DataLess {
    bool operator()(const Data &lhs, const Data &rhs) const {
        return visit(*this, lhs, rhs);
    }
    bool operator()(const auto &a, const auto &b) const {
        return a < b; // double < double
                       // double < long long
                       // long long < double
                       // long long < long long
    };
};

int main() {
    vector<Data> v{1e23, -3.5, 123'456'789'123, 10.5};
    sort(v.begin(), v.end(), DataLess{});
    for (const auto &d : v) {
        visit([](const auto &x) {
            cout << fixed << typeid(x).name() << " " << x << "\n";
        }, d);
    }
}
```



The diagram consists of two curved arrows. One arrow starts from the `visit(*this, lhs, rhs);` line in the `operator()` of `DataLess` and points to the `const auto &a, const auto &b` parameters in the second `operator()`. The other arrow starts from the `DataLess{}` argument in the `sort` function call and points to the `DataLess` struct definition.

Sorting a heterogenous list

```
using Data = variant<long long, double>; // bools, strings, arrays...
struct DataLess {
    bool operator()(const Data &lhs, const Data &rhs) const {
        return visit(*this, lhs, rhs);
    }
    bool operator()(const auto &a, const auto &b) const {
        return a < b; // double < double
                       // double < long long
                       // long long < double
                       // long long < long long
    }
};

int main() {
    vector<Data> v{1e23, -3.5, 123'456'789'123, 10.5};
    sort(v.begin(), v.end(), DataLess{});
    for (const auto &d : v) {
        visit([](const auto &x) {
            cout << fixed << typeid(x).name() << " " << x << "\n";
        }, d);
    }
}
```

Output:

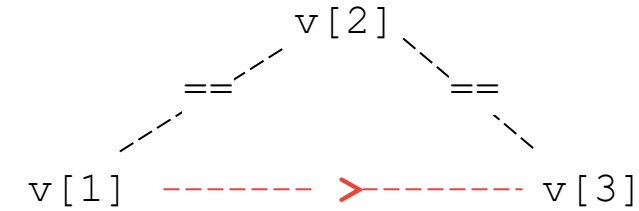
```
d -3.500000
d 10.500000
x 123456789123
d 9999999999999999
```

Output

```
d -3.500000
d 10.500000
x 123456789123
d 99999999999999991611392.000000
```

The bug

```
vector<Data> v{
    123'456'789'123'456'781,    // v[0]
    // <
    123'456'789'123'456'785,    // v[1]
    // ==
    double{123'456'789'123'456'784}, // v[2]
    // ==
    123'456'789'123'456'783,    // v[3]
};
assert(is_sorted(v.begin(), v.end(), DataLess{})); // passes
assert(DataLess{}(v[3], v[1])); // passes too?!
```



- [alg.sorting.general]/3: “comp shall induce a strict weak ordering on the values.”
- [alg.sorting.general]/4.5: “it can be shown that ... is a strict total ordering”
- Undefined behavior for `std::sort`, for `std::set...`, for in-house algorithms as well
- <https://stackoverflow.com/questions/18291620/why-will-stdsort-crash-if-the-comparison>

C++ can only compare within a type

There is no `operator<(long long, double)`.
We need two values of **the same type**.

```
// Original
bool operator()(auto a, auto b) const
{
    ...
    return a < b;
}

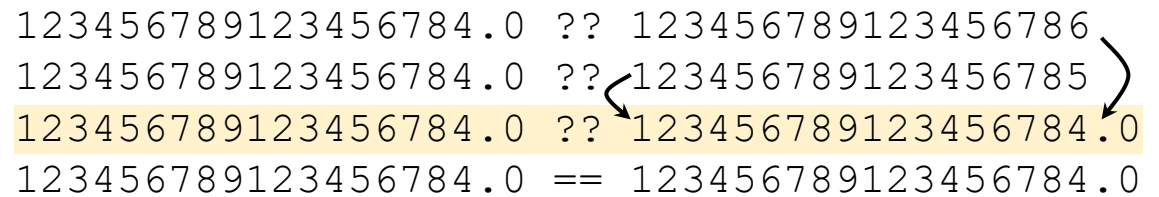
// Substitute types
bool operator()(double a, long long b)
{
    ...
    return a < b;
}

// There are implicit conversions
bool operator()(double a, long long b)
{
    ...
    return a < static_cast<double>(b);
}
```

Floating-integral conversions [conv.fpint]/2:

- ... an integer type ... can be converted to ... a floating point
- The result is exact, if possible

```
123456789123456784.0 ?? 123456789123456786
123456789123456784.0 ?? 123456789123456785
123456789123456784.0 ?? 123456789123456784.0
123456789123456784.0 == 123456789123456784.0
```

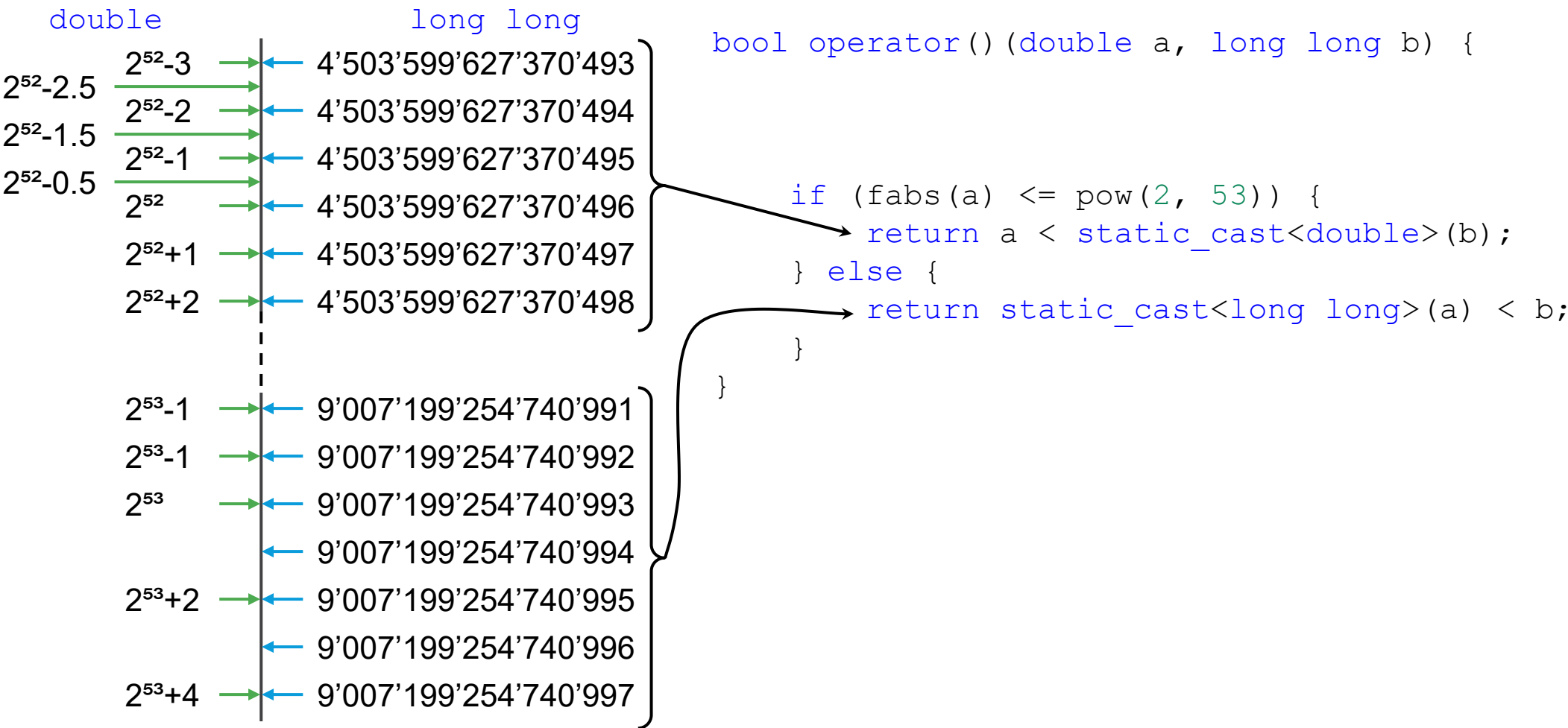


Hence

```
123456789123456784.0 == 123456789123456785
```

- ... it's an implementation-defined choice of either the next lower or higher representable value
 - [GCC](#), [MSVC](#) claim to round to the nearest, no tiebreaker
 - [Clang](#) does not say

The main fix (almost)



Double-to-long long is UB

double		long long
$2^{52}-2.5$	$2^{52}-3$	4'503'599'627'370'493
$2^{52}-1.5$	$2^{52}-2$	4'503'599'627'370'494
$2^{52}-0.5$	$2^{52}-1$	4'503'599'627'370'495
	2^{52}	4'503'599'627'370'496
	$2^{52}+1$	4'503'599'627'370'497
	$2^{52}+2$	4'503'599'627'370'498
<hr/>		
	$2^{53}-1$	9'007'199'254'740'991
	$2^{53}-1$	9'007'199'254'740'992
	2^{53}	9'007'199'254'740'993
		9'007'199'254'740'994
	$2^{53}+2$	9'007'199'254'740'995
		9'007'199'254'740'996
	$2^{53}+4$	9'007'199'254'740'997

```
bool operator() (double a, long long b) {  
  
    if (fabs(a) <= pow(2, 53)) {  
        return a < static_cast<double>(b);  
    } else {  
        return static_cast<long long>(a) < b;  
    }  
}
```

Floating-integral conversions [conf.fpoint]/1:

- ... a floating point type can be converted to ... an integer type
- The conversion truncates; that is, the fractional part is discarded.
- **The behavior is undefined** if the truncated value cannot be represented in the destination type (nan, 1e100, -∞...).
- <https://stackoverflow.com/questions/77292701/how-does-visual-studio-2022-handle-double-to-long-long-out-of-range>

Still incorrect?

double		long long
$2^{52}-2.5$	$2^{52}-3$	4'503'599'627'370'493
$2^{52}-1.5$	$2^{52}-2$	4'503'599'627'370'494
$2^{52}-0.5$	$2^{52}-1$	4'503'599'627'370'495
	2^{52}	4'503'599'627'370'496
	$2^{52}+1$	4'503'599'627'370'497
	$2^{52}+2$	4'503'599'627'370'498

	$2^{53}-1$	9'007'199'254'740'991
	$2^{53}-1$	9'007'199'254'740'992
	2^{53}	9'007'199'254'740'993
		9'007'199'254'740'994
	$2^{53}+2$	9'007'199'254'740'995
		9'007'199'254'740'996
	$2^{53}+4$	9'007'199'254'740'997

```
bool operator()(double a, long long b) {  
    if (a < LLONG_MIN) return false;  
    if (a > LLONG_MAX) return true;  
    if (fabs(a) <= pow(2, 53)) {  
        return a < static_cast<double>(b);  
    } else {  
        return static_cast<long long>(a) < b;  
    }  
}
```

Still incorrect.

double		long long
$2^{52}-2.5$	$2^{52}-3$	4'503'599'627'370'493
$2^{52}-1.5$	$2^{52}-2$	4'503'599'627'370'494
$2^{52}-0.5$	$2^{52}-1$	4'503'599'627'370'495
	2^{52}	4'503'599'627'370'496
	$2^{52}+1$	4'503'599'627'370'497
	$2^{52}+2$	4'503'599'627'370'498

	$2^{53}-1$	9'007'199'254'740'991
	$2^{53}-1$	9'007'199'254'740'992
	2^{53}	9'007'199'254'740'993
		9'007'199'254'740'994
	$2^{53}+2$	9'007'199'254'740'995
		9'007'199'254'740'996
	$2^{53}+4$	9'007'199'254'740'997

```
bool operator()(double a, long long b) {  
    if (a < static_cast<double>(LLONG_MIN)) return false;  
    if (a > static_cast<double>(LLONG_MAX)) return true;  
    if (fabs(a) <= pow(2, 53)) {  
        return a < static_cast<double>(b);  
    } else {  
        return static_cast<long long>(a) < b;  
    }  
}
```

Still incorrect (2)

double		long long
$2^{52}-2.5$	$2^{52}-3$	4'503'599'627'370'493
$2^{52}-1.5$	$2^{52}-2$	4'503'599'627'370'494
$2^{52}-0.5$	$2^{52}-1$	4'503'599'627'370'495
	2^{52}	4'503'599'627'370'496
	$2^{52}+1$	4'503'599'627'370'497
	$2^{52}+2$	4'503'599'627'370'498
<hr/>		
	$2^{53}-1$	9'007'199'254'740'991
	$2^{53}-1$	9'007'199'254'740'992
	2^{53}	9'007'199'254'740'993
		9'007'199'254'740'994
	$2^{53}+2$	9'007'199'254'740'995
		9'007'199'254'740'996
	$2^{53}+4$	9'007'199'254'740'997

```
bool operator()(double a, long long b) {  
    if (a < -9'223'372'036'854'775'808.0 ) return false;  
    if (a > 9'223'372'036'854'775'807.0 ) return true;  
    if (fabs(a) <= pow(2, 53)) {  
        return a < static_cast<double>(b);  
    } else {  
        return static_cast<long long>(a) < b;  
    }  
}
```

$\text{LLONG_MAX} == 9'223'372'036'854'775'807$

$a == \text{double}\{9'223'372'036'854'775'808\}$

`assert(!(a > LLONG_MAX));` // passes

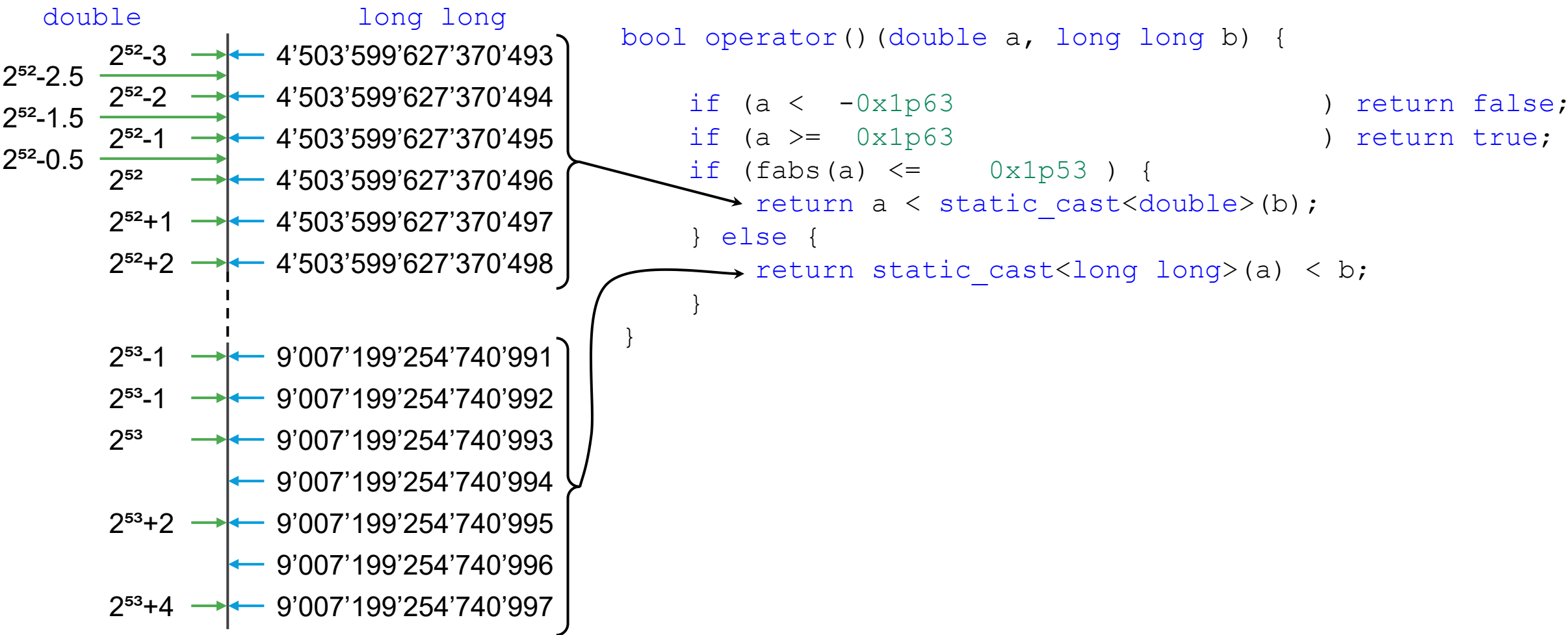
`static_cast<long long>(a)` // undefined behavior

Works*, but not pretty

double		long long
$2^{52}-2.5$	$2^{52}-3$	4'503'599'627'370'493
$2^{52}-1.5$	$2^{52}-2$	4'503'599'627'370'494
$2^{52}-0.5$	$2^{52}-1$	4'503'599'627'370'495
	2^{52}	4'503'599'627'370'496
	$2^{52}+1$	4'503'599'627'370'497
	$2^{52}+2$	4'503'599'627'370'498
<hr/>		
	$2^{53}-1$	9'007'199'254'740'991
	$2^{53}-1$	9'007'199'254'740'992
	2^{53}	9'007'199'254'740'993
		9'007'199'254'740'994
	$2^{53}+2$	9'007'199'254'740'995
		9'007'199'254'740'996
	$2^{53}+4$	9'007'199'254'740'997

```
bool operator()(double a, long long b) {  
    if (a < -9'223'372'036'854'775'808.0 ) return false;  
    if (a >= 9'223'372'036'854'775'808.0 ) return true;  
    if (fabs(a) <= pow(2, 53)) {  
        return a < static_cast<double>(b);  
    } else {  
        return static_cast<long long>(a) < b;  
    }  
}  
  
LLONG_MAX == 9'223'372'036'854'775'807  
a == double{9'223'372'036'854'775'808}  
assert(!(a > LLONG_MAX)); // passes  
static_cast<long long>(a) // undefined behavior
```

Works*, kind of pretty



Works

double		long long
$2^{52}-2.5$	$2^{52}-3$	4'503'599'627'370'493
$2^{52}-1.5$	$2^{52}-2$	4'503'599'627'370'494
$2^{52}-0.5$	$2^{52}-1$	4'503'599'627'370'495
	2^{52}	4'503'599'627'370'496
	$2^{52}+1$	4'503'599'627'370'497
	$2^{52}+2$	4'503'599'627'370'498
<hr/>		
	$2^{53}-1$	9'007'199'254'740'991
	$2^{53}-1$	9'007'199'254'740'992
	2^{53}	9'007'199'254'740'993
		9'007'199'254'740'994
	$2^{53}+2$	9'007'199'254'740'995
		9'007'199'254'740'996
	$2^{53}+4$	9'007'199'254'740'997

```
bool operator()(double a, long long b) {  
    if (isnan(a)) return true;  
    if (a < -0x1p63) return false;  
    if (a >= 0x1p63) return true;  
    if (fabs(a) <= 0x1p53) {  
        return a < static_cast<double>(b);  
    } else {  
        return static_cast<long long>(a) < b;  
    }  
}
```

```
!(nan < 5) && !(5 < nan) && !(5 == nan) // ?!
```

Bonus points:

- Get rid of the `isnan(a)` check by using `!(x < nan)`
- Use `std::numeric_limits<double>`
- <https://stackoverflow.com/questions/24347670/how-to-compare-a-long>

Conclusion

Bloomberg

Engineering

Know your inputs!

123.6500000000000000568434188608080148696899414062500000

1. Exact number slightly bigger than 123.65?
2. 123.65 stored in `double`?
3. 123.65000000000000006 stored in `double`?

Depends on how we obtain the input:

- `double` variable
- Direct user input
- Exact string coming from JSON/XML; # of decimal places known
- `double` coming from a library that parsed JSON/XML

Know your outputs!

123.6500000000000000568434188608080148696899414062500000

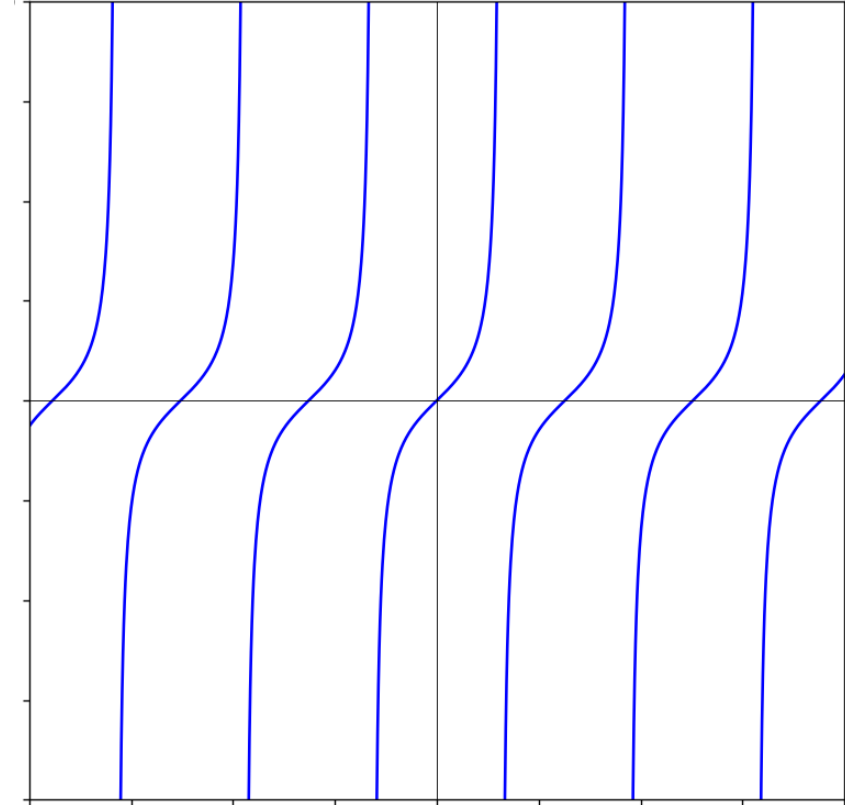
1. Exact number slightly bigger than 123.65?
2. 123.65 stored in `double`?
3. 123.65000000000000006 stored in `double`?

Depends on who will consume the output

- `double` variable that is an approximation
- `double` variable that, when rounded, should have 2 decimal places
- Storing in Excel/Word
- Exact string going to JSON/XML

How stable is your $f(x)$?

- Stable: $\text{sqrt}(a \times 1.001) \approx \text{sqrt}(a)$
- $\tan(a)$ jumps from $+\infty$ to $-\infty$
- Any discreteness is bad
- Other bad functions:
 - `if`
 - `round`
 - `cout` (because it rounds)
- Do you make **any binary decisions**?
- Numerical Analysis?



Possible requirements

Context	Inputs	Outputs	Stable	Needs
Progress bar	Integers	Pixels to render	Yes	Does not show 100% unless done
Financial report	Decimals with 2 places	Decimals with 2 places	Not really	Exact results
Game	Player inputs	Display	No: collisions	Fast
Game with saves	Players inputs	Display, save files	No: collisions	Cross-platform
Simulation	Numbers	Numbers	Hopefully?	



Use integers with integer inputs!

- Progress bar in integers: `progress * 100 / max`
- $(a / b == c / d) == (a * d == c * b)$
- Two vectors are parallel:
 - Polar angles are the same
 - Dot product is zero
- Number of solutions for $ax^2+bx+c=0$
 - Check the sign of `sqrt(b2-4ac)`
 - Check the sign of b^2-4ac
- Average is greater than X
 - Sum is greater than $N * X$



Testing

- Test like any other component
- There are corner cases: infinities, nan, subnormals
 - Both for correctness and performance
- Test smart algorithms for precision at least
 - E.g.: use Python's arbitrary integers to generate golden test
 - Ensure the algorithm is better than the naive one.
 - My first several implementations did not work well.
 - Maybe do not implement yourself?
- There are 2^{32} floats, try them all
- Just random tests are not enough:

<https://scicomp.stackexchange.com/questions/40541/what-are-some-good-strategies-to-test-a-floating-point-arithmetic-implementation>



Rules of thumb

1. Use integers instead
2. Understand your inputs and outputs
3. Assume approximate, unless you are ready to prove
4. Use better formulas before micro-optimizing precision
 - a. There are libraries for numerical methods
 - b. Do not calculate Taylor series manually
5. When micro-optimizing, be wary of the compiler

<https://randomascii.wordpress.com/category/floating-point/>

<https://floating-point-gui.de/references/>

“What Every Computer Scientist Should Know About Floating-Point Arithmetic” - David Goldberg

TechAtBloomberg.com

© 2025 Bloomberg Finance L.P. All rights reserved.

Bloomberg

Engineering

Thank you!

Learn more:

www.TechAtBloomberg.com/cplusplus

We are hiring: bloomberg.com/engineering

Bloomberg
Engineering

TechAtBloomberg.com