Type definition

FRINT representation: 61.21672 => 0.6121672 * 10^2
FRINT is composed two main parts and one auxiliary part:
- Main part: significand - signed long integer that represents the digits after `0`
- Main part: exponent - signed long integer that represents degree of ten
- Auxiliary: nd - unsigned short integer that represents number of digits in the significand. It's a sort of cache that allows reducing the number of repetitive operations.

The notion of FRINT number:
$X = \{s, e\}$, where s - significand, e - exponent. Nd is not shown as it can be computed from significand.

$nd(X) = floor(log_{10}(X_s)) + 1$
We are using signed long ints to represent negative numbers. In case if we used unsigned long ints, we would have to use an additional field - sign that would take 8 bits (C++ doesn't allow any data type less than 1 byte).
The maximum number is $2^{32} - 1 = 2, 147, 483, 648$. We intentionally limit it to 9 digits (until 999,999,999) to make overflow avoidance easier.

Conversions

Float

Converting from float to FRINT comes in 2 steps: finding exponent and calculating significand.
Exponent is calculated as a logarithm of ten from the number.
The significand is calculated in two steps: normalize the number to the form of (0.X) by dividing the number by 10^exp. After that, multiply by maximum integer number and convert to long int.

$f(x) = \{10^9 \times x / 10^{nd(x)}, nd(x)\}$

Integer

Converting from integer to FRINT is pretty straightforward: the number itself can be used as significand and the number of digits used as exponential:
$f(x) = \{x, nd(x)\}$

Short-comings

One of the main threats in dealing with operations are integer overflow when dealing with a multiplication and sometimes addition operations. To mitigate this threat, we should consider using overflow analysis **before** the operation is performed and make sure that overflow is not going to happen. The overflow is a real threat mainly for significand, so we will focus only on it. The number's most significant digits are located in the left while the ones in the right can be safely omitted for avoiding overflow.
Overflow check: for multiplication $overflow(A, B) = nd(A) + nd(B) > 9$
Shift for making sure to avoid overfit:

- Calculate if `overflow(A, B)`. If it's false, nothing should be done.
- Calculate how much the biggest number can be reduced: $dA = nd(A) - nd(B)$ and shift the first number by dA.
- If $nd(A) - dA + nd(B) <= 9$, we can finish.
- Shift the first number by: $10 - nd(B) / 2$ and the second number by: $10 - nd(B) / 2 + mod(nd(B), 2)$

Operations

Comparison

Equality operation: if exponential and significands are equal, the numbers are equal.
$eq(X, Y) = X_e == Y_e \& X_s = X_e$
Lower than operation: there are three cases:

- When exponentials are not equal: $lt_1(X, Y) = X_e < Y_e$
- When exponentials are equal and significands are not equal: $lt_2(X, Y) = normR(X_s, nd(X), nd(Y)) < normR(Y_s, nd(Y), nd(X))$
- When exponentials and significands are equal: $lt_3(X, Y) = nd(X) < nd(Y)$

Significant normalization: equalizes the number of digits in the significand:
$norm(X, nd_x, nd_y) = if\ nd_x > nd_y\ then\ shiftRight(X, nd_x - nd_y)\ else\ shiftLeft(X, nd_y - nd_x)$

Shift operations:
$shiftLeft(number, nd) = number \times 10^{nd}$
$shiftRight(number, nd) = number / 10^{nd}$

Derived operations:
$X > Y = gt(X, Y) = !\,le(X, Y)$

$X >= Y = ge(X, Y) = !lt(X, Y)$
$X != Y = neq(X, Y) = !eq(X, Y)$

Arithmetic

Negation: defined as negation of significand without change of exponential:
$- X = neg(X) = \{- X_s, X_e\}$

Addition: addition of significands with respect to exponents difference. The first operand is guaranteed to not be smaller than the second:

$X + Y = add(X, Y) = \{X_s + norm(Y_s, nd(X), nd(Y)), max(X_e, Y_e) + ov(X, Y)\}$
$ov(X, Y) = if\ nb(X_s + norm(Y_s, nd(X), nd(Y))) > nd(X_s)\ then\ 1\ else\ 0$

Subtraction: defined as addition of the first operand and negated second operand
$X - Y = sub(X, Y) = X + (- Y)$

Inverse: the negation of the exponential and division of maximum degree of 10 by significand
$X^{-1} = inv(X) = \{10^{10}/X_s, - X_e\}$.

Multiplication: the multiplication of significands and addition of exponents
$X \times Y = mul(X, Y) = \{X_s \times Y_x, X_e + Y_e\}$

Division: the multiplication of the first operand and the inverse of the second operand
$X / Y = div(X, Y) = X \times Y^{-1}$

Experiments

Experimentations setup and benchmark

We run two different tests:

- Precision tests: to check how well the numbers are persisted in various operations and during the conversions.
- Performance tests: we run every crucial operation 1,000,000 times and get how much time it took to perform it. We repeat it 20 times and use the mean time spent. For some cases, we compare them with results get for float operation.

Precision tests: initializing using long, double, and floats; conversion into float, string, long, double; comparison operations, addition, subtraction, multiplication, negation, inversion, division.

Performance tests: initialization using long, double, float; conversion into string, float, double, long; equal and lower than with two number types (depending on the number of digits); addition, subtraction, negation, inverse, multiplication, division.

Running environment: 15.4 GB RAM, 11th Gen Intel® Core™ i5-1135G7 @ 2.40GHz × 8, Ubuntu 20.04.5 LTS 64 bit. Using G++ compiler: g++ (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0

Experimentation results

Space for improvements

Better overflow management

This would help increase the numbers' range from 10^10 to 2^31.

Convert signed integers into unsigned

This would help to increase the numbers' range to 2^32 but make some operations more complex and increase memory usage by 1 byte per number.

Normalizing the significand

Right now, we are not making sure that significand doesn't contain any trailing zeros. This makes it more vulnerable to overflow and makes it more expensive. Here we couldn't tailor any non-loop based algorithm to remove these zeros, so it was easier to leave. If there will be a non-expensive way to remove them, it will help improve the performance.