# Representation of Integers and Reals: Section 2

By **misof**– *TopCoder Member*

...read Section 1

***Rumor:*** *Floating point variables can store not only numbers but also some strange values.*

***Validity:*** *True.*

As stated in the previous answer, the standard reserves both the smallest and the largest possible value of the exponent to store special numbers. (Note that in memory these values of the exponent are stored as "all zeroes" and "all ones", respectively.)

Zero

When talking about the sign-mantissa-exponent representation we noted that any **non-zero** number can be represented in this way. Zero is not directly representable in this way. To represent zero we will use a special value denoted with both the exponent field and the mantissa containing all zeroes. Note that -0 and +0 are distinct values, though they both compare as equal.

It is worth noting that if `memset()` is used to fill an array of floating point variables with zero bytes, the value of the stored numbers will be zero. Also, global variables in C++ are initialized to a zero bit pattern, thus global floating point variables will be initialized to zero.

Also, note that negative zero is sometimes printed as "-0" or "-0.0". In some programming challenges (with inexperienced problemsetters) this may cause your otherwise correct solution to fail.

There are quite a few subtle pitfalls concerning the negative zero. For example, the expressions "`0.0 - x`" and "`-x`" are not equivalent – if $x = 0.0$, the value of the first expression is 0.0, the second one evaluates to -0.0.

My favorite quote on this topic: *Negative zeros can "create the opportunity for an educational experience" when they are printed as they are often printed as "-0" or "-0.0" (the "educational experience" is the time and effort that you spend learning why you're getting these strange values).*

Infinities

The values +infinity and -infinity correspond to an exponent of all ones and a mantissa

of all zeroes. The sign bit distinguishes between negative infinity and positive infinity. Being able to denote infinity as a specific value is useful because it allows operations to continue past overflow situations.

Not a Number

The value NaN (Not a Number) is used to represent a value that does not represent a real number. NaNs are represented by a bit pattern with an exponent of all ones and a non-zero mantissa. There are two categories of NaN: QNaN (Quiet NaN) and SNaN (Signaling NaN).

A QNaN is a NaN with the most significant bit of the mantissa set. QNaNs propagate freely through most arithmetic operations. These values pop out of an operation when the result is not mathematically defined. (For example, `3*sqrt(-1.0)` is a QNaN.)

An SNaN is a NaN with the most significant bit of the mantissa clear. It is used to signal an exception when used in operations. SNaNs can be handy to assign to uninitialized variables to trap premature usage.

If a return value is a QNaN, it means that it is impossible to determine the result of the operation, a SNaN means that the operation is invalid.

Subnormal numbers

We still didn't use the case when the exponent is all zeroes and the mantissa is non-zero. We will use these values to store numbers very close to zero.

These numbers are called *subnormal*, as they are smaller than the normally representable values. Here we don't assume we have a leading 1 before the binary point. If the sign bit is *s*, the exponent is all zeroes and the mantissa is *m*, the value of the stored number is $(-1)^s \times 0.m \times 2^{-q}$, where *q* is 126 for single and 1022 for double precision.

(Note that zero is just a special case of a subnormal number. Still, we wanted to present it separately.)

Summary of all possible values

In the following table, *b* is the bias used when storing the exponent, i.e., 127 for single and 1023 for double precision.

| **sign** *s* | **exponent** *e* | **mantissa** *m* | represented number |
|---|---|---|---|
| 0 | 00...00 | 00...00 | +0.0 |
| | | | |

| | | | |
|---|---|---|---|
| 0 | 00...00 | 00...01 to 11...11 | $0.m \times 2^{-b+1}$ |
| 0 | 00...01 to 11...10 | anything | $1.m \times 2^{e-b}$ |
| 0 | 11...11 | 00...00 | +Infinity |
| 0 | 11...11 | 00...01 to 01...11 | SNaN |
| 0 | 11...11 | 10...00 to 11...11 | QNaN |
| 1 | 00...00 | 00...00 | -0.0 |
| 1 | 00...00 | 00...01 to 11...11 | $-0.m \times 2^{-b+1}$ |
| 1 | 00...01 to 11...10 | anything | $-1.m \times 2^{e-b}$ |
| 1 | 11...11 | 00...00 | -Infinity |
| 1 | 11...11 | 00...01 to 01...11 | SNaN |
| 1 | 11...11 | 10...00 to 11.11 | QNaN |

Operations with all the special numbers

All operations with the special numbers presented above are well-defined. This means that your program won't crash just because one of the computed values exceeded the representable range. Still, this is usually an unwanted situation and if it may occur, you should check it in your program and handle the cases when it occurs.

The operations are defined in the probably most intuitive way. Any operation with a NaN yields a NaN as a result. Some other operations are presented in the table below. (In the table, *r* is a positive representable number, ? is Infinity, ÷ is normal floating point division.) A complete list can be found in the standard or in your compiler's documentation. Note that even comparison operators are defined for these values. This topic exceeds the scope of this article, if interested, browse through the references presented at the end of the article.

| operation | result |
|---|---|
| 0 ÷ ±? | 0 |
| ±*r* ÷ ±? | 0 |
| $(-1)^{s}? \times (-1)^{t}?$ | $(-1)^{st}?$ |
| ? + ? | ? |
| ±*r* ÷ 0 | ±? |
| 0 ÷ 0 | NaN |
| ? – ? | NaN |
| ±? ÷ ±? | NaN |

| | |
|---|---|
| ±? x 0 | NaN |

***Rumor:*** *Floating point numbers can be compared by comparing the bit patterns in memory.*

***Validity:*** *True.*

Note that we have to handle sign comparison separately. If one of the numbers is negative and the other is positive, the result is clear. If both numbers are negative, we may compare them by flipping their signs, comparing and returning the opposite answer. From now on consider non-negative numbers only.

When comparing the two bit patterns, the first few bits form the exponent. The larger the exponent is, the further is the bit pattern in lexicographic order. Similarly, patterns with the same exponent are compared according to their mantissa.

Another way of looking at the same thing: when comparing two non-negative real numbers stored in the form described above, the result of the comparison is always the same as when comparing integers with the same bit pattern. (Note that this makes the comparison pretty fast.)

***Rumor:*** *Comparing floating point numbers for equality is usually a bad idea.*

***Validity:*** *True.*

Consider the following code:

```
for (double r=0.0; r!=1.0; r+=0.1) printf("*");
```

How many stars is it going to print? Ten? Run it and be surprised. The code just keeps on printing the stars until we break it.

Where's the problem? As we already know, `double`s are not infinitely precise. The problem we encountered here is the following: In binary, the representation of 0.1 is not finite (as it is in base 10). Decimal 0.1 is equivalent to binary 0.0(0011), where the part in the parentheses is repeated forever. When 0.1 is stored in a `double` variable, it gets rounded to the closest representable value. Thus if we add it 10 times the result is not exactly equal to one.

The most common advice is to use some tolerance (usually denoted ?) when comparing two `double`s. E.g., you may sometimes hear the following hint: consider the `double`s `a` and `b` equal, if `fabs(a-b)<1e-7`. Note that while this is an improvement, it is

not the best possible way. We will show a better way later on.

***Rumor:*** *Floating point numbers are not exact, they are rounded.*

***Validity:*** *Partially true.*

Yes, if a number can't be represented exactly, it has to be rounded. But sometimes an even more important fact is that lots of important numbers (like zero, the powers of two, etc.) can be stored exactly. And it gets even better. Note that the mantissa of `double`s contains more than 32 bits. Thus all the binary digits of an `int` fit into the mantissa and the stored value is exact.

This can still be improved. If we note that ? when comparing floating point numbers.

***Validity:*** *False.*

Often if you visit the Round Tables after a SRM that involved a floating point task you can see people posting messages like "after I changed the precision from `1e-12` to `1e-7` it passed all systests in the practice room"

Examples of such discussions: here, here, here, here and here. (They are worth reading, it is always less painful to learn on the mistakes of other people made than to learn on your own mistakes.)

We will start our answer by presenting another simple example.

```
for (double r=0.0; r<1e22; r+=1.0) printf(".");
```

How many dots will this program print? This time it's clear, isn't it? The terminating condition doesn't use equality testing. The cycle has to stop after $10^{22}$ iterations. Or... has it?

Bad luck, this is again an infinite cycle. Why is it so? Because when the value of *r* becomes large, the precision of the variable isn't large enough to store all decimal digits of *r*. The last ones become lost. And when we add 1 to such a large number, the result is simply rounded back to the original number.

Exercise: Try to estimate the largest value of *r* our cycle will reach. Verify your answer. If your estimate was wrong, find out why.

After making this observation, we will show why the expression `fabs(a-b)<epsilon` (with a fixed value of `epsilon`, usually recommended between `1e-7` and `1e-9`) is not ideal for comparing `double`s.

Consider the values 123456123456.1234588623046875 and 123456123456.1234741210937500. There's nothing that special about them. These are just two values that can be stored in a `double` without rounding. Their difference is approximately `2e-5`.

Now take a look at the bit patterns of these two values:

```
 first: 01000010 00111100 10111110 10001110 11110010 01000000 00011111 10011011
second: 01000010 00111100 10111110 10001110 11110010 01000000 00011111 10011100
```

Yes, right. These are two consecutive values that can be stored in a `double`. Almost any rounding error can change one of them onto the other one (or even further). And still, they are quite far apart, thus our original test for "equality" fails.

What we really want is to tolerate small precision errors. As we already saw, `double`s are able to store approximately 15 most significant decimal digits. By accumulating precision errors that arise due to rounding, the last few of these digits may become corrupt. But how exactly shall we implement tolerating such errors?

We won't use a constant value of ?, but a value relative to the magnitude of the compared numbers. More precisely, if *x* is a `double`, then `x*1e-10` is a number that's 10 degrees of magnitude smaller than *x*. Its most significant digit corresponds to *x*'s eleventh most significant digit. This makes it a perfect ? for our needs.

In other words, a better way to compare `double`s `a` and `b` for "equality" is to check whether `a` lies between `b*(1-1e-10)` and `b*(1+1e-10)`. (Be careful, if `b` is negative, the first of these two numbers is larger!)

See any problems with doing the comparison this way? Try comparing `1e-1072` and `-1e-1072`. Both numbers are almost equal to zero and to each other, but our test fails to handle this properly. This is why we have to use **both** the first test (known as testing for an absolute error) and the second test (known as testing for a relative error).

This is the way TC uses to check whether your return value is correct. Now you know why.

There are even better comparison functions (see one of the references), but it is important to know that in practice you can often get away with using only the absolute error test. Why? Because the numbers involved in computation come from a limited range. For example, if the largest number you will ever compare is 9947, you know that a `double` will be able to store another 11 digits after the decimal point correctly. Thus if we use `epsilon=1e-8` when doing the absolute error test, we allow the last three

significant digits to become corrupt.

The advantage this approach gives you is clear: checking for an absolute error is much simpler than the advanced tests presented above.

- Elections (a Div2 easy with a success rate of only 57.58%)
- Archimedes
- SortEstimate (the binary search is quite tricky to get right if you don't understand precision issues)
- PerforatedSheet (beware, huge rounding errors possible)
- WatchTower
- PackingShapes

***Rumor:*** *Computations using floating point variables are as exact as possible.*

***Validity:*** *True.*

Most of the standards require this. To be even more exact: For any arithmetical operation the returned value has to be that representable value that's closest to the exact result. Moreover, in C++ the default rounding mode says that if two values are tied for being the closest, the one that's more even (i.e., its least significant bit of the mantissa is 0) is returned. (Other standards may have different rules for this tie breaking.)

As a useful example, note that if an integer $n$ is a square (i.e., $n = k^2$ for some integer $k$), then `sqrt(double(n))` will return the exact value $k$. And as we know that $k$ can be stored in a variable of the same type as $n$, the code `int k = int(sqrt(double(n)))` is safe, there will be no rounding errors.

***Rumor:*** *If I do the same computation twice, the results I get will be equal.*

***Validity:*** *Partially true.*

Wait, only partially true? Doesn't this contradict the previous answer? Well, it doesn't.

In C++ this rumor isn't always true. The problem is that according to the standard a C++ compiler can sometimes do the internal calculations using a larger data type. And indeed, g++ sometimes internally uses `long double`s instead of `double`s to achieve larger precision. The value stored is only typecast to `double` when necessary. If the compiler decides that in one instance of your computation `long double`s will be used

and in the other just `double`s are used internally, the different roundings will influence the results and thus the final results may differ.

This is one of THE bugs that are almost impossible to find and also one of the most confusing ones. Imagine that you add debug outputs after each step of the computations. What you unintentionally cause is that after each step each of the intermediate results is cast to `double` and output. In other words, you just pushed the compiler to only use `double`s internally and suddenly everything works. Needless to say, after you remove the debug outputs, the program will start to misbehave again.

A workaround is to write your code using `long double`s only.

Sadly, this only cures one of the possible problems. The other is that when optimizing your code the compiler is allowed to rearrange the order in which operations are computed. On different occasions, it may rewrite two identical pieces of C++ code into two different sets of instructions. And all the precision problems are back.

As an example, the expression $x + y - z$ may once be evaluated as $x + (y - z)$ and the other time as $(x + y) - z$. Try substituting the values $x = 1.0$ and $y = z = 10^{30}$.

Thus even if you have two identical pieces of code, you can't be sure that they will produce exactly the same result. If you want this guarantee, wrap the code into a function and call the same function on both occasions.

Further reading

- Comparing floating point numbers (a detailed article by Bruce Dawson)
- Floating-point representation
- IEEE Standard 754
- Integer Types In C and C++ (an article by Jack Klein)
- Java Floating-Point Number Intricacies (an article by Thomas Wang)
- Lecture notes on IEEE-754 (by William Kahan)
- Lots of referrences about IEEE-754
- Revision of IEEE-754 (note the definition of the operators min and max)
- What Every Computer Scientist Should Know About Floating-Point Arithmetic (a pretty long article by David Goldberg)