Representation of Integers and Reals

By **misof**
TopCoder Member

Choosing the correct data type for your variables can often be the only difference between a faulty solution and a correct one. Especially when there's some geometry around, precision problems often cause solutions to fail. To make matters even worse, there are many (often incorrect) rumors about the reasons of these problems and ways how to solve them.

To be able to avoid these problems, one has to know a bit about how things work inside the computer. In this article we will take a look at the necessary facts and disprove some false rumors. After reading and understanding it, you should be able to avoid the problems mentioned above.

This article is in **no way** intended to be a complete reference, nor to be 100% accurate. Several times, presented things will be a bit simplified. As the readers of this article are TopCoder (TC) members, we will concentrate on the x86 architecture used by the machines TC uses to evaluate solutions. For example, we will assume that on our computers a byte consists of 8 bits and that the machines use 32-bit integer registers.

While most of this article is general and can be applied on all programming languages used at TC, the article is slightly biased towards C++ and on some occasions special notes on g++ are included.

We will start by presenting a (somewhat simplified) table of integer data types available in the g++ compiler. You can find this table in any g++ reference. All of the other compilers used at TC have similar data types and similar tables in their references, look one up if you don't know it by heart yet. Below we will explain that all we need to know is the storage size of each of the types, the range of integers it is able to store can be derived easily.

Table 1: Integer data types in g++.

name	size in bits	representable range
`char`	8	-2^7 to $2^7 - 1$
`unsigned char`	8	0 to $2^8 - 1$
`short`	16	-2^{15} to $2^{15} - 1$
`unsigned short`	16	0 to $2^{16} - 1$
`int`	32	-2^{31} to $2^{31} - 1$
`unsigned int`	32	0 to $2^{32} - 1$
`long`	32	-2^{31} to $2^{31} - 1$
`unsigned long`	32	0 to $2^{32} - 1$
`long long`	64	-2^{63} to $2^{63} - 1$
`unsigned long long`	64	0 to $2^{64} - 1$

Notes:

- The storage size of an `int` and an `unsigned int` is platform dependent. E.g., on machines using 64-bit registers, `int`s in g++ will have 64 bits. The old Borland C compiler used 16-bit `int`s. It is guaranteed that an `int` will always have at least 16 bits. Similarly, it is guaranteed that on any system a `long` will have at least 32 bits.
- The type `long long` is a g++ extension, it is not a part of any C++ standard (yet?). Many other C++ compilers miss this data type or call it differently. E.g., MSVC++ has `__int64` instead.

Rumor: *Signed integers are stored using a sign bit and "digit" bits.*

Validity: *Only partially true.*

Most of the current computers, including those used at TC, store the integers in a so-called *two's complement form*. It is true that for non-negative integers the most significant bit is zero and for negative integers it is one. But this is not exactly a sign bit, we can't produce a "negative zero" by flipping it. Negative numbers are stored in a somewhat different way. The negative number -n is stored as a bitwise negation of the non-negative number (n-1).

In Table 2 we present the bit patterns that arise when some small integers are stored in a (signed) `char` variable. The rightmost bit is the least significant one.

Table 2: Two's complement bit patterns for some integers.

value	two's complement form
0	00000000
1	00000001
2	00000010
46	00101110
47	00101111
127	01111111
-1	11111111
-2	11111110
-3	11111101
-47	11010001
-127	10000001
-128	10000000

Note that due to the way negative numbers are stored the set of representable numbers is not placed symmetrically around zero. The largest representable integer in b bits is $2^{b-1} - 1$, the smallest (i.e., most negative) one is -2^{b-1}.

A neat way of looking at the two's complement form is that the bits correspond to digits in base 2 with the exception that the largest power of two is negative. E.g., the bit pattern 11010001 corresponds to $1 \times (-128) + 1 \times 64 + 0 \times 32 + 1 \times 16 + 0 \times 8 + 0 \times 4 + 0 \times 2 + 1 \times 1 = -128 + 81 = -47$

Rumor: *Unsigned integers are just stored as binary digits of the number.*

Validity: *True.*

In general, the bit pattern consists of base 2 digits of the represented number. E.g., the bit pattern 11010001 corresponds to $1 \times 128 + 1 \times 64 + 0 \times 32 + 1 \times 16 + 0 \times 8 + 0 \times 4 + 0 \times 2 + 1 \times 1 = 209$.

Thus, in a b-bit unsigned integer variable, the smallest representable number is zero and the largest is $2^b - 1$ (corresponding to an all-ones pattern).

Note that if the leftmost (most significant) bit is zero, the pattern corresponds to the same value regardless of whether the variable is signed or unsigned. If we have a b-bit pattern with the leftmost bit set to one, and the represented unsigned integer is x, the same pattern in a signed variable represents the value $x - 2^b$.

In our previous examples, the pattern 11010001 can represent either 209 (in an unsigned variable) or -47 (in a signed variable).

Rumor: *In C++, the code "*`int A[1000]; memset(A,x,sizeof(A));`*" stores 1000 copies of x into A.*

Validity: *False.*

The `memset()` function fills a part of the memory with `char`s, not `int`s. Thus for most values of x you would get unexpected results.

However, this does work (and is often used) for two special values of x: 0 and -1. The first case is straightforward. By filling the entire array with zeroes, all the bits in each of the`int`s will be zero, thus representing the number 0. Actually, the second case is the same story: -1 stored in a `char` is 1111111, thus we fill the entire array with ones, getting an array containing -1s.

(Note that most processors have a special set of instructions to fill a part of memory with a given value. Thus the `memset()` operation is usually much faster than filling the array in a cycle.)

When you know what you are doing, `memset()` can be used to fill the array A with sufficiently large/small values, you just have to supply a suitable bit pattern as the second argument. E.g., use $x = 63$ to get really large values ($1,061,109,567$) in A.

Rumor: *Bitwise operations can be useful.*

Validity: *True.*

First, they are fast. Second, many useful tricks can be done using just a few bitwise operations.

As an easy example, x is a power of 2 if and only if `(x & (x-1) == 0)`. (Why? Think how does the bit pattern of a power of 2 look like.) Note that `x=x & (x-1)` clears the least significant set bit. By repeatedly doing this operation (until we get zero) we can easily count the number of ones in the binary representation of x.

If you are interested in many more such tricks, download the free second chapter of the book Hacker's Delight and read The Aggregate Magic Algorithms.

One important trick: `unsigned int`s can be used to encode subsets of $\{0, 1,..., 31\}$ in a straightforward way - the i-th bit of a variable will be one if and only if the represented set contains the number i. For example, the number 18 (binary 10010 = $2^4 + 2^1$) represents the set $\{1, 4\}$.

When manipulating the sets, bitwise "and" corresponds to their intersection, bitwise "or" gives their union.

In C++, we may explicitly set the i-th bit of x using the command `x |= (1<<i)`, clear it using `x &= ~(1<<i)` and check whether it is set using `((x & (1<<i)) != 0)`. Note that `bitset` and `vector<bool>` offer a similar functionality with arbitrarily large sets.

This trick can be used when your program has to compute the answer for all subsets of a given set of things. This concept is quite often used in SRM problems. We won't go into more details here, the best way of getting it right is looking at an actual implementation (try looking at the best solutions for the problems below) and then trying to solve a few such problems on your own.

- BorelSets (a simple exercise in set manipulation, generate sets until no new sets appear)
- TableSeating
- CompanyMessages
- ChessMatch (for each subset of your players find the best assignment)
- RevolvingDoors (encode your position and the states of all the doors into one integer)

Rumor: *Real numbers are represented using a floating point representation.*

Validity: *True.*

The most common way to represent "real" numbers in computers is the *floating point* representation defined by the IEEE Standard 754. We will give a brief overview of this representation.

Basically, the words "floating point" mean that the position of the decimal (or more exactly, binary) point is not fixed. This will allow us to store a large range of numbers than fixed point formats allow.

The numbers will be represented in scientific notation, using a normalized number and an exponent. For example, in base 10 the number 123.456 could be represented as 1.23456×10^2. As a shorthand, we sometimes use the letter E to denote the phrase "times 10 to the power of". E.g., the previous expression can be rewritten as `1.23456e2`.

Of course, in computers we use binary numbers, thus the number 5.125 (binary 101.001) will be represented as 1.01001×2^2, and the number -0.125 (binary -0.001) will be represented as -1×2^{-3}.

Note that any (non-zero) real number x can be written in the form $(-1)^s \times m \times 2^e$, where $s \in \{0, 1\}$ represents the sign, $m \in [1, 2)$ is the normalized number and e is the (integer) exponent. This is the general form we are going to use to store real numbers.

What exactly do we need to store? The base is fixed, so the three things to store are the sign bit s, the normalized number (known as the *mantissa*) m and the exponent e.

The IEEE Standard 754 defines four types of precision when storing floating point numbers. The two most commonly used are *single* and *double precision*. In most programming languages these are also the names of corresponding data types. You may encounter other data types (such as *float*) that are platform dependent and usually map to one of these types. If not sure, stick to these two types.

Single precision floating point numbers use 32 bits (4 bytes) of storage, double precision numbers use 64 bits (8 bytes). These bits are used as shown in Table 3:

Table 3: Organization of memory in `singles` and `doubles`.

	sign	exponent	mantissa
single precision	1	8	23
double precision	1	11	52

(The bits are given in order. I.e., the sign bit is the most significant bit, 8 or 11 exponent bits and then 23 or 52 mantissa bits follow.)

The sign bit
The sign bit is as simple as it gets. 0 denotes a positive number; 1 denotes a negative number. Inverting this bit changes the sign of the number.

The exponent
The exponent field needs to represent both positive and negative exponents. To be able to do this, a *bias* is added to the actual exponent e. This bias is 127 for single precision and 1023 for double precision. The result is stored as an unsigned integer. (E.g., if $e = -13$ and we use single precision, the actual value stored in memory will be $-13 + 127 = 114$.)

This would imply that the range of available exponents is -127 to 128 for single and -1023 to 1024 for double precision. This is almost true. For reasons discussed later, both boundaries are reserved for special numbers. The actual range is then -126 to 127, and -1022 to 1023, respectively.

The mantissa
The mantissa represents the precision bits of the number. If we write the number in binary, these will be the first few digits, regardless of the position of the binary point. (Note that the position of the binary point is specified by the exponent.)

The fact that we use base 2 allows us to do a simple optimization: We know that for any (non-zero) number the first digit is surely 1. Thus we don't have to store this digit. As a result, a b-bit mantissa can actually store the $b + 1$ most significant bits of a number.

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 contests (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, $\div$ 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

$0 \div \pm\infty$	0
$\pm r \div \pm\infty$	0
$(-1)^s\infty \times (-1)^t\infty$	$(-1)^{st}\infty$
$\infty + \infty$	∞
$\pm r \div 0$	$\pm\infty$
$0 \div 0$	NaN
$\infty - \infty$	NaN
$\pm\infty \div \pm\infty$	NaN
$\pm\infty \times 0$	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 ε) 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 `doubles` 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 $2^{54} > 10^{16}$, it should be clear that any integer with up to 15 decimal digits has at most 54 binary digits, and thus it can be stored in a`double` without rounding. This observation can even be extended to non-integer values: `double` is able to store 15 most significant decimal digits of any number (in the normally representable range).

A similar computation for `singles` shows that they are able to store only 7 most significant decimal digits. This is way too little for almost any practical applications, and what's even more important, it is less than the precision required by TC when a floating point value shall be returned. The moral of this story is pretty clear: **Never use `singles`!**Seriously. Don't even think about it. There's plenty of available memory nowadays.

As a side note, once rounding errors occur in your computation, they are propagated into further computations. Thus even if the final result shall be an integer, its representation in a floating point variable may not be exact. As an example consider the star-printing cycle above.

***Rumor:** I've heard that `long doubles` will give me more precision.*

***Validity:** Platform dependent.*

One less common precision type defined in the IEEE-754 standard is *extended double precision*, which requires at least 79 bits of storage space. Some compilers have a data type with this precision, some don't. E.g., in g++ on the x86 architecture we have a data type `long double` that uses 10 bytes (80 bits) of memory. (In MSVC++ the type`long double` is present, but it is mapped to a `double`.)

The 80-bit extended double precision format is used internally by the Intel 80x87 floating-point math co-processor in order to be able to shift operands back and forth without any loss of precision in the IEEE-754 64-bit (and 32-bit) format. When optimization in g++ is set to a non-zero value, g++ may even generate code that uses `long doubles` internally instead of `doubles` and `singles`. This format is able to store 19 most significant decimal digits.

If even more precision is required, you may either implement your own arithmetic, or use the BigInteger and BigDecimal classes from Java's math library.

***Rumor:** In practice, there's no real difference between using different values of ε 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 `doubles`.

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, `doubles` 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 ε, 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 ε for our needs.

In other words, a better way to compare `doubles` `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 doubles` instead of `doubles` 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)