Representation of Integers and Reals: Section 1

By **misof** — *TopCoder Member*
Discuss this article in the forums

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 â^^ {0, 1} represents the sign, m â^^ [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 `single`s and `double`s.

	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.

...continue to Section 2