The fundamental data types

Real numbers are represented according to the IEEE 754 standard. Note that a real number x in base 10 in scientific notation can be written as $x = \pm m \times 10^E$, where E is the exponent and $1 \leq m < 10$. This is also referred to as the real number x in its normalized form. For example, 0.00034 is equal to 3.4×10^{-4} in normalized form. We can also represent a real number x in base 2 in normalized form as $x = \pm m \times 2^E$, where $1 \leq m < 2$, i.e.

$$m = (b_0.b_1b_2b_3\cdots)_2,$$

with $b_0 = 1$. This we further write in the form $m = (1.M)_2$, where $M = (b_1b_2b_3\cdots)_2$. Since we know that the leading bit b_0 is 1, we do not store it. Thus, finally, we have

$$x = (-1)^S \ \times \ 2^E \ \times \ (1.M).$$

Here, S is the sign, E is the exponent, and M is the mantissa.

The data type float

We describe the details of the data type `float` here. The other real types `double` and `long double` follow closely that of `float`, the main differences arise due to the extra bytes that are used to store them. The data type `float` is explained assuming that 4 bytes are used to store them (true for my macOS). Note that a nice decimal real number in base 10 may become a repeating decimal in the binary system (base 2). For example,

$$(0.1)_{10} = (0.00011001100110011....)_2 = (1.10011001100\cdots)_2 \times 2^{-4}.$$

Hence, to store it in a binary system using a finite number of bits, it must be truncated, giving approximate values.

S	$E + 127$	M
$\pm$	$a_1a_2a_3\cdots a_8$	$b_1b_2b_3\cdots b_{23}$

In a 4-byte representation, the first bit is reserved for sign: 0 for positive and 1 for negative. The next 8 bits, $a_1a_2a_3\cdots a_8$, form a bit string used for the exponent. However, the exponent is stored using biased representation: the exponent bit-string is simply the binary representation of $E + 127$, and the number 127 is called exponent bias. Now, using 8 bits, we can represent non-negative numbers in the range from 0 (all 8 bits 0) to 255 (all 8 bits 1). Out of them 0 and 255 are reserved for other purposes (to be described shortly). Hence, the exponent bit string for normalized numbers varies from 1 to 254, and the actual exponent ranges from -126 to 127. The last 23 bits are reserved for mantissa M.

Machine precision (or machine epsilon, ϵ) for a `float` is the smallest positive number that, when added to 1.0, results in a value still distinguishable from 1.0 by the computer. It represents

the limit of relative accuracy in floating-point arithmetic. It is effectively the gap between 1.0 and the next representable floating-point number. Now $1.0 = 1.0 \times 2^0$ and hence

$$S = 0 \quad \text{and} \quad E + 127 = 0 + 127 = 1111\,1111.$$

Thus, 1.0 in floating-point representation can be written as

S	$E + 127$	M
0	1111 1111	0000 0000 0000 0000 0000 000

The next real number in floating-point representation is

S	$E + 127$	M
0	1111 1111	0000 0000 0000 0000 0000 001,

which is

$$(1.0000\,0000\,0000\,0000\,0000\,001)_2 = 1 + 2^{-23}.$$

Thus, machine precision for `float` is $2^{-23} \approx 1.2 \times 10^{-7}$ and there are around six-to-seven significant digits (in base 10) in `float` representation. Also, note that any real number between 1 and $1 + 2^{-23}$ is rounded to one of them. This implies that there are gaps in `float` system compared to the continuum real number system.

If exponent bit-string $a_1a_2\cdots a_8$ is	Then numerical value represented is
$(00000000)_2 = (0)_{10}$	$\pm(0.b_1b_2b_3\cdots\cdots b_{23})_2 \times 2^{-126}$
$(00000001)_2 = (1)_{10}$	$\pm(1.b_1b_2b_3\cdots\cdots b_{23})_2 \times 2^{-126}$
$(00000010)_2 = (2)_{10}$	$\pm(1.b_1b_2b_3\cdots\cdots b_{23})_2 \times 2^{-125}$
$(00000011)_2 = (3)_{10}$	$\pm(1.b_1b_2b_3\cdots\cdots b_{23})_2 \times 2^{-124}$
$\vdots$	$\vdots$
$(01111111)_2 = (127)_{10}$	$\pm(1.b_1b_2b_3\cdots\cdots b_{23})_2 \times 2^{0}$
$(10000000)_2 = (128)_{10}$	$\pm(1.b_1b_2b_3\cdots\cdots b_{23})_2 \times 2^{1}$
$\vdots$	$\vdots$
$(11111100)_2 = (252)_{10}$	$\pm(1.b_1b_2b_3\cdots\cdots b_{23})_2 \times 2^{125}$
$(11111101)_2 = (253)_{10}$	$\pm(1.b_1b_2b_3\cdots\cdots b_{23})_2 \times 2^{126}$
$(11111110)_2 = (254)_{10}$	$\pm(1.b_1b_2b_3\cdots\cdots b_{23})_2 \times 2^{127}$
$(11111111)_2 = (255)_{10}$	$\pm\infty$ if $b_1 = b_2 = \cdots = b_{23} = 0$, NaN otherwise

The table above shows the arrangement of real numbers in `float` system. The numbers between the top row and bottom row represent the normalized numbers, i.e., they have a hidden bit $b_0 = 1$. The normalized numbers have 24 significant bits, which consist of $b_0 = 1$ and $b_1, b_2, \cdots, b_{23}$. The smallest magnitude normalized numbers have representation

S	$E + 127$	M
$\pm$	0000 0001	0000 0000 0000 0000 0000 000,

which is $\pm 1.0 \times 2^{1-127} = \pm 2^{-126} \approx \pm 1.2 \times 10^{-38}$. Similarly, the highest magnitude normalized real number has representation

S	$E + 127$	M
$\pm$	1111 1110	1111 1111 1111 1111 1111 11,

which is

$$\pm(1.1111\,1111\,1111\,1111\,1111\,111)_2 \times 2^{254-127} = \pm(2 - 2^{-23})2^{127} \approx \pm 3.4 \times 10^{38}.$$

If a positive number exceeds the highest magnitude positive number, then the output in a "printf()" results in `inf` (stands for ∞). Similarly, if a negative number is less than the highest magnitude negative number, then the output in a "printf()" results in `-inf` (stands for $-\infty$). Note that apart from $\pm$`inf`, sometimes we also get `nan` as output, which means "not a number". It can occur in various scenarios involving floating-point calculations where the result is mathematically undefined or cannot be represented as a finite number. Common examples include division `0.0/0.0`, square root of a negative number, etc. The configurations of $\pm$`inf` and `nan` are shown in the last row of the table.

We have seen earlier that the smallest magnitude normalized floating-point numbers are $\pm 2^{-126} \approx \pm 1.2 \times 10^{-38}$. Any number of less magnitude should have been treated as zero. However, we can still get nonzero numbers of lesser magnitude by sacrificing the number of significant digits. These numbers are non-normalized, they are called subnormal numbers. The configurations for subnormal numbers are shown in the top row of the table. Here, the hidden bit is absent. The highest magnitude subnormal numbers have representation

S	$E + 127$	M
$\pm$	0000 0000	1111 1111 1111 1111 1111 111,

which is equivalent to $\pm(1-2^{-23}) \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$. Note that the above has 23 significant bits. The lowest magnitude subnormal numbers have representation

S	$E + 127$	M
$\pm$	0000 0000	0000 0000 0000 0000 0000 001,

which is equivalent to $\pm 2^{-23}2^{-126} = 2^{-149} \approx \pm 1.4 \times 10^{-45}$. Note that the above has only one significant bit (the last one). Now, any number whose magnitude is less than the lowest magnitude subnormal number is represented as zero.

The code below illustrates these points. Here we use "asin()" function as well as the power function "pow()" in code. Note that `asin(x)` computes $\sin^{-1}(x)$, which is defined for $-1 \leq x \leq 1$. Since $\sin^{-1}(2)$ is not defined, we obtain `nan` in the output. The expression `pow(x,y)` returns an approximate value of x^y. To use these mathematical functions, we need to include the math header file `math.h`. To compile this code, we need to instruct the compiler to link the math library. This is accomplished by `-lm` option in `gcc` command. The output of the code is explained in the comments used in the program.

```
//Program to test range of float
#include<stdio.h>
#include<math.h>
int main()
{
        float a,b;
        a=(2-pow(2,-23))*pow(2,127); // a approx largest +ve real
        b=(2-pow(2,-24))*pow(2,127); // b higher than a => inf
        printf("a=%e    b=%e\n",a,b);
        a=asin(2.0);                 // sin^{-1}(2) not defined=>nan
        printf("a=%e\n",a);
        a=pow(2,-149);              // a approx lowest +ve real
        b=pow(2,-150);             //  b lower than a=> 0
        printf("a=%e    b=%e\n",a,b);
        return 0;
}
```

Listing 12: C program "cflt.c"

Compiling with `$ gcc cflt.c -lm ↩` and running the default executable with `$ ./a.out ↩` produce the following output:

```
a=3.402823e+38    b=inf
a=nan
a=1.401298e-45    b=0.000000e+00
```

The data type double

In a 8-byte representation (true for my macOS), the first bit is reserved for sign: 0 for positive and 1 for negative. The next 11 bits, $a_1a_2a_3\cdots a_{11}$, form a bit string used for the exponent. However, the exponent is stored using biased representation: the exponent bit-string is simply the binary representation of $E + 1023$, and the number 1023 is the exponent bias. The actual exponent ranges from -1022 to 1023. The last 52 bits are reserved for mantissa M.

Machine precision (or machine epsilon, ϵ) for `double` is $2^{-52} \approx 2.2 \times 10^{-16}$ and thus there are around fifteen-to-sixteen significant digits (in base 10) in `double` representation. The lowest magnitude normalized numbers are $\pm 2^{-1022} \approx \pm 2.2 \times 10^{-308}$ and the highest magnitude normalized numbers are $\pm(2 - 2^{-52}) \times 2^{1023} \approx \pm 1.8 \times 10^{308}$. Also, the lowest magnitude subnormal numbers are $\pm 2^{-1074} \approx \pm 4.94 \times 10^{-324}$.

The data type long double

In C, long double refers to a floating-point data type that is often more precise than `double`

though the language standard only requires it to be at least as precise as `double`. We will not discuss this further.

Return to Lecture Topics