Karol Sobiesiak

Base types

Things you have to know!

Last update: 07/05/2016
Original release: 25/08/2015

int

Integer numbers encoding variations:

bits	unsigned	signed two's complement	signed one's complement
0111 1111	127	127	127
0111 1110	126	126	126
0000 0010	2	2	2
0000 0001	1	1	1
0000 0000	0	0	0
1111 1111	255	−1	-0
1111 1110	254	−2	-1
1000 0010	130	−126	-125
1000 0001	129	−127	-126
1000 0000	128	−128	-127
		Common encoding for signed integer	Rarely used (0 duplicated)

int

Two's complement

0000 0010	2
0000 0001	1
0000 0000	0
1111 1111	−1
1111 1110	−2

Negative values are made by negating bits of positive values and adding 1.
You can use that method to go from negative to positive values as well.
(watch out for minimum value, in 8 bit there is -128 but no 128)

from 5 to -5:

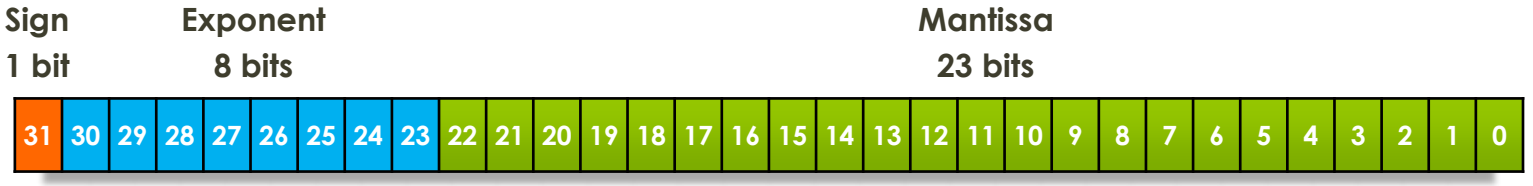
5 ➔	**0000 0101**
1. Negate	1111 1010
2. Add 1	1111 1011 **(-5)**

from -5 to 5:

-5 ➔	**1111 1011**
1. Negate	0000 0100
2. Add 1	0000 0101 **(5)**

For one's complement it's the same, except that you just have to negate the bits.

float (IEEE 754)



Sign – codes sign of the whole number (0 – pos, 1 – neg)
Exponent – codes exponent value with 2 as a base
Mantissa – codes fractional number (still, decimal separator is controled by exponent)

Special values:

Sign	Exponent	Mantissa	Meaning
0	all 0	all 0	+0
1			-0
0		has 1	+denormal
1			-denormal
0	all 1	all 0	+infinity
1			-infinity
0		has 1	NaN
1			NaN

float (IEEE 754)

General concept

(sign) mantissa × base^{exponent}

10 base – human way	2 base – float way
Mantissa = 3.14159265	**Mantissa** = 1.00101111
Exponent = 5 $3.14159265 \times 10^{5} = 314159.265$	**Exponent** = 5 $1.0010101 \times 2^{5} = 100101.111$
Exponent = -3 $3.14159265 \times 10^{-3} = 0.00314159265$	**Exponent** = -3 $1.0010101 \times 2^{-3} = 0.00100101111$

note: different numbers used

float (IEEE 754)

Exponent

- Uses base of two: $2^{exponent}$
- Can be coded as:
  - 8 bit unsigned integer from 0 to 255 – **usual case** **(that's what we use here)**
    - Shifted by a bias, exponent = 127 represents actual zero (see below)
    - i.e. for $2^{(exp-127)}$ to be one, exponent bits must represent 127 ($2^{(127-127)} = 1$)
  - 8 bit signed integer from −128 to 127 (2's complement)

To find out what is the **actual value of the binary exponent** substract 127 from it:

keep in mind that all 0's and all 1's hold special meanings

00011101_b == 29 ➔ 29 - 127 == -98
00000001_b == 1 ➔ 1 - 127 == -126 **(min)**
11111110_b == 254 ➔ 254 - 127 == 127 **(max)**
01111111_b == 127 ➔ 127 - 127 == 0

float (IEEE 754)

Mantissa

- Codes fractional number #.**###**
- There is also one additional (integer side) bit: **#**.###, sometimes described as 24th mantissa implicit bit which is derived from the exponent
  - It's 1 in usual case
    - As 24th bit it increases precision which will become apparent later
  - It's 0 when number is denormal or 0 (all exponent bits are 0). When number is denormal (really small) calculations are more expensive

float (IEEE 754)

Formula – bits to value

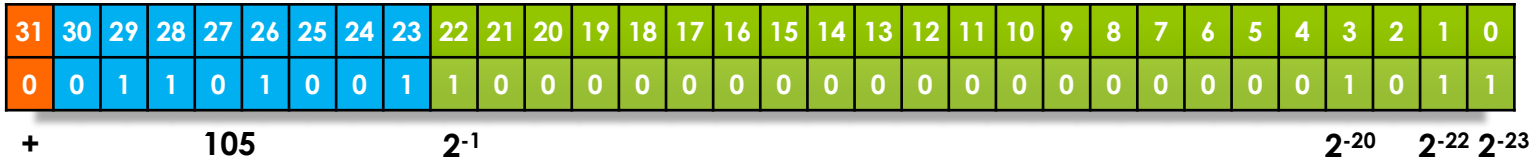
$$(-1)^{sign}(b_{lead}.\, b_{22}b_{21}b_{20}\ldots b_0)_2 \cdot 2^{(exp-127)}$$

or

$$(-1)^{sign}\left(b_{lead} + \sum_{i=1}^{23} b_{23-i}2^{-i}\right) \cdot 2^{(exp-127)}$$

***b*_{lead}** – codes additional integer bit 1.#### or 0.#### (see mantissa slide)

Example



$b_{lead} = 1$ (normal number, exponent bits are not all 0's)

$(-1)^0 \times (1.10000000000000000001011)_b \times 2^{(105-127)} =$ // $2^{-22} = 0.00000023841857910156 25$

$(-1)^0 \times (1 + 2^{-1} + 2^{-20} + 2^{-22} + 2^{-23}) \times 2^{(105-127)} =$ // $1 + 2^{-1} + 2^{-20} + 2^{-22} + 2^{-23} = 1.50000131130218505859375$

$1 \times 1.50000131130218505859375 \times 0.00000023841857910156 25 =$

$3.57628181291147484444081783294677734375 \times 10^{-7}$

float (IEEE 754)

Formula – value to bits

Decimal number has to be converted to this form: **1.### × 2^{exp}**

Example: 13.7

13.7 / 2 = 13.7 / 2^1 = 6.85
6.85 / 2 = 13.7 / 2^2 = 3.425
3.425 / 2 = 13.7 / 2^3 = **1.7125**

Formula for abs(number) > 1:
Repeatedly divide resulting numbers by 2 till you reach base form

// shortcut: exponent = $\lfloor log_2(13.7) \rfloor$ (only for values ≥ 1)

We've got our base form: **1.7125 × 2^3**

Coding Exponent

As we said before 127 $(01111111)_2$ is biased 0
127 + **3** = 130

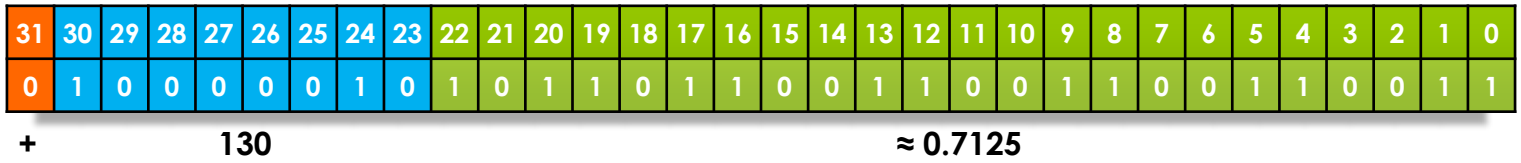
Exponent:
$130_{10} = 10000010_2$
Mantissa:
$0.7125_{10} = 1011(0110)_2$

Coding Mantissa

0.7125 // we don't code implied bit (1.###) into mantissa

0.7125 × 2	= 1.425	// got 1
0.425 × 2	= 0.85	// got 0
0.85 × 2	= 1.7	// got 1
0.7 × 2	= 1.4	// got 1
0.4 × 2	**= 0.8**	// got 0
0.8 × 2	= 1.6	// got 1
0.6 × 2	= 1.2	// got 1
0.2 × 2	= 0.4	// got 0
0.4 × 2	**= 0.8**	// got 0 and **pattern**

Formula:
Repeatedly multiply last calculated fractional number by 2 and successively store resulting integer numbers (0 or 1) in mantissa bits array (from most significant bit to less).



float (IEEE 754)

Formula – value to bits

Decimal number has to be converted to this form: **1.### × 2^{exp}**

Example: -0.21

0.21 × 2 = 0.21 / 2^{-1} = 0.42
0.42 × 2 = 0.21 / 2^{-2} = 0.84
0.84 × 2 = 0.21 / 2^{-3} = **1.68**

We've got our base form: **1.68 × 2^{-3}**

Formula for abs(number) < 1:
Repeatedly multiply resulting numbers by 2 till you reach base form

Coding Exponent

As we said before 127 $(01111111)_2$ is biased 0
127 - **3** = 124

Exponent:
$124_{10} = 01111100_2$
Mantissa:
$0.68_{10} \approx 10101110000101000111101_2$

Coding Mantissa

0.68 // we don't code implied bit (1.###) into mantissa

0.68 × 2	= 1.36	// got 1
0.36 × 2	= 0.72	// got 0
0.72 × 2	= 1.44	// got 1
0.44 × 2	= 0.88	// got 0
0.88 × 2	**= 1.76**	// got 1
0.76 × 2	= 1.52	// got 1
0.52 × 2	= 1.04	// got 1
0.04 × 2	= 0.08	// got 0
…		

Formula:
Repeatedly multiply last calculated fractional number by 2 and successively store resulting integer numbers (0 or 1) in mantissa bits array (from most significant bit to less).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	1	0	1	0	1	1	1	0	0	0	0	1	0	1	0	0	0	1	1	1	1	0	1

- **124** **≈ 0.68**

float (IEEE 754)

Formula – value to bits (other approach)

Calculate integer part and fractional part separately, combine and move separator to reach base form of 1.####. Nr of moves left (+) or right (-) is actual exponent.

Example: 13.7

integer.fractional (integer = 13, fractional = 0.7)

integer (13) ➔ binary

13 / 2 = 6.5 // 1 (LSB)
6 / 2 = 3 // 0
3 / 2 = 1.5 // 1
1 / 2 = 0.5 // 1 (MSB)

$13_{10} = 1101_2$

fractional (0.7) ➔ binary

0.7 × 2 = 1.4 // 1 (MSB)
0.4 × 2 = 0.8 // 0
0.8 × 2 = 1.6 // 1
0.6 × 2 = 1.2 // 1
0.2 × 2 = **0.4** // 0 (**pattern**)

$0.7_{10} = .1(0110)$

$1101.1(0110) \times 2^{3}$ ➔ $1.1011(0110)$ // move separator **3** places left (right to the separator is mantissa)

Exponent = 127 + 3 = 130 = 10000010_2
Mantissa = $1011(0110)_2$

float (IEEE 754)

Accuracy

Why is there additional 24th implied (by exponent) bit in mantissa?

- To get additional bit of accuracy
- There will always be 1 at some point looking at the bits from most significant side, so why not add one at the top
  - Exponent moves it (with whole mantissa) left or right

1.00101 * 2^{5} = **1**00101

1.00101 * 2^{-5} = 0.0000**1**00101

Accuracy is around 7-8 most significant digits in decimal base

- That's why even though in „bits to value" example we calculated $3.576281812911474844440817832946777343 75 \times 10^{-7}$ as a value from bit representation, IDEs will usually show only the significant part of that number.

float (IEEE 754)

Accuracy

Precision for floating point numbers (IEEE 754) is the same when the same exponent is applied. Meaning, if we keep increasing mantissa by a single bit, the difference between two adjacent values will remain the same. For example:

A = 1.0
B = 1.0000001**1920928955078125**
B - A = 0.0000001**1920928955078125** // step for this exponent

8 digits of precision

A = 1.**9999997**615814208984375
B = 1.**9999998**8079071044921875
B - A = 0.0000001**1920928955078125** // the same

float (IEEE 754)

Accuracy

Rule from previous slide applies only within the same exponent. When exponent changes, the step changes as well. This change has radius of approx. single digit point so that precision is always within 7-8 most significant decimal digits.

Function below, written in JavaScript, visualizes accuracy for each exponent (like in prev. slide for exp=0), additionally reducing results to form #.### and removing big exponents (e.g. #.####e10 – removing e10) to make it easier to see number of precision digits. Javascript by default uses double precision, that's why we can keep calculations simple.

Hint: use the result from previous slide (exp=0) as a reference to understand resulting data.

```
function floatStepsForEachExponent()
{
  for(var i = -126; i <= 127; i++)
  {
    var A = Math.pow(2, i);                          // mantissa = 0b
    var B = (1.0 + Math.pow(2,-23)) * Math.pow(2, i)  // mantissa = 1b (marked only the LSB)
    var res;
    while(true)
    {
      if(B >= 10)
      { A /= 10; B /= 10; }
      else if(B < 1)
      { A *= 10; B *= 10; }
      else
      {
        res = B - A;
        console.log("exp = " + i + ": " + res.toFixed(10));
        break;
      }
    }
  }
}
```

popular floating point types

long double – 80bit (10 bytes)

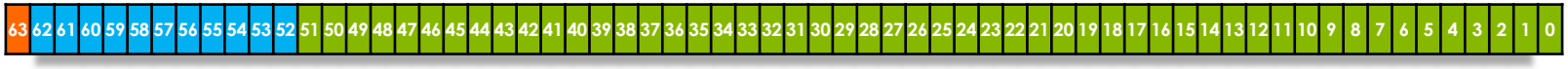
coding is little diffrent from the others

Sign
1 bit

Exponent
15 bits

Mantissa
64 bits

double – 64bit (8 bytes)



Sign
1 bit

Exponent
11 bits

Mantissa
52 bits

float – 32bit (4 bytes)

Sign
1 bit

Exponent
8 bits

Mantissa
23 bits

half – 16bit (2bytes)

Sign
1 bit

Exponent
5 bits

Mantissa
10 bits

e scientific notation

$$a\mathbf{e}b \rightarrow a \times 10^{b}$$

$3.1415\mathbf{e}\text{-}2 = 3.1415 \times 10^{-2} = 0.031415$

$3.1415\mathbf{e}2 = 3.1415 \times 10^{2} = 314.15$