Real Arithmetic

Richard Šusta, Pavel Píša

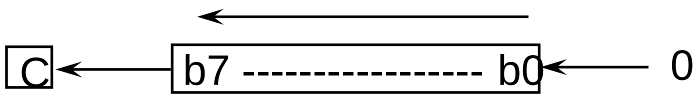
Czech Technical University in Prague, Faculty of Electrical Engineering

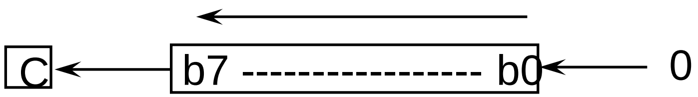
Speed of operations

Operation	Language C
Bit negation	`~x`
Multiplying or dividing by 2^n	`x<<n , x>>n`
Increment, decrement	`++x, x++, --x, x--`
Minus number <- bit negation+increment	`-x`
Adding	`x+y`
Subtracting <- minus + addition	`x-y`
Multiplying by hardware multiplyer	`x*y`
Multiplying by sequence multiplyer	
Divisiov	`x/y`

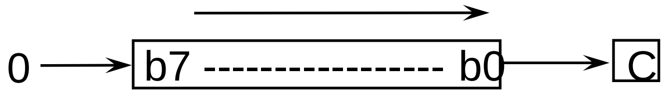
Logical Shift

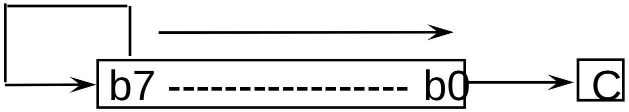
Arithmetic Shift





multiply by 2





divide by 2 unsigned

divide by 2 signed

Sign Extension Example in C

```
short int x =  15213;
int       ix = (int) x;
short int y = -15213;
int       iy = (int) y;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 C4 92	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

Non-restoring division

1 1 1 : 0 1 1

```
        0 0 0 1 1 1   :   0 0 1 1
[-]     1 1 0 0 : :   negate
              1 : :   hot one
      ---------
      0 1 1 1 0 : :   −  ⇒ 0
          ↓ ↓ ↓ ↓ :
          1 1 0 1 :
[+]       0 0 1 1 :
        ---------
        1 0 0 0 0 1      +  ⇒ 1
            ↓ ↓ ↓ ↓
            0 0 0 1
[-]         1 1 0 0
                  1
          ---------
          0 1 1 1 0          −  ⇒ 0
[N]         0 0 1 1   return
          ---------
          1 0 0 0 1

              0 0 1 — reminder        0 1 0 — quotient
```

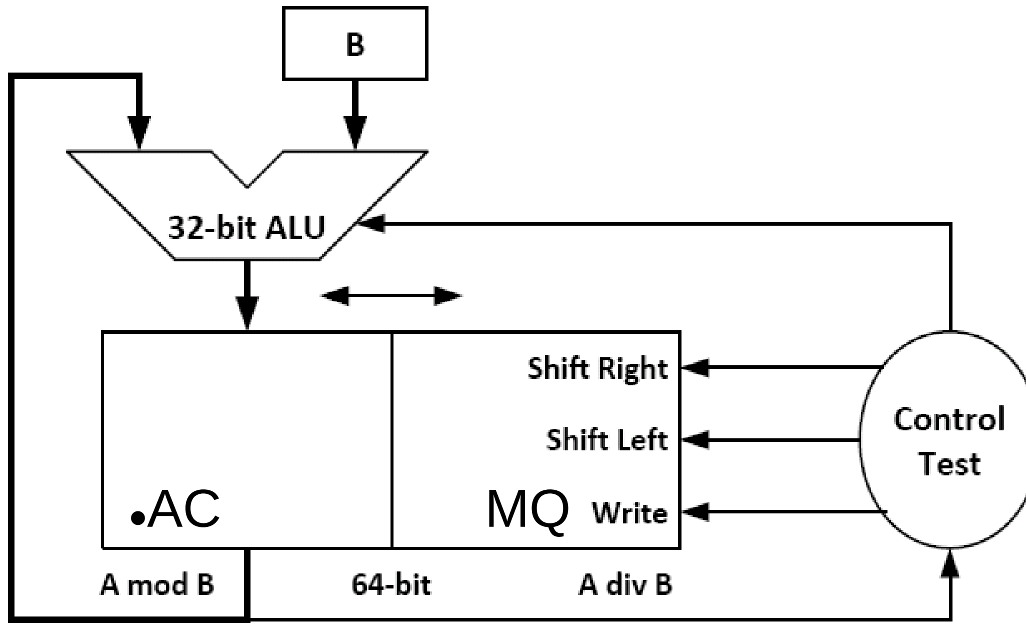
Hardware divider logic (32b case)

1 1 1 : 0 1 1

- divident = quotient × divisor + reminder

```
   0 0 0 1 1 1   :  0 0 1 1
⊟    1 1 0 0 : :    negate
           1 : :    hot one
 0 1 1 1 0   : :    −  ⇒  0
     ↓ ↓ ↓ ↓ :
     1 1 0 1 :
⊞    0 0 1 1 :
   1 0 0 0 0 1      +  ⇒  1
       ↓ ↓ ↓ ↓
       0 0 0 1
⊟      1 1 0 0
             1
     0 1 1 1 0      −  ⇒  0
Ⓝ      0 0 1 1      return
     1 0 0 0 1

       0 0 1 — : reminder      0 1 0 —  quotient
```



Non-restoring division

Algorithm of the sequential division

Restoring division

```
MQ = dividend;
B = divisor; (Condition: divisor is not 0!)
AC = 0;


for( int i=1; i <= n; i++){
   SL (shift AC MQ by one bit to the left, the LSB bit is kept on zero)

   if(AC >= B)   {
      AC = AC – B;
      MQ0 = 1;    // the LSB of the MQ register is set to 1
    }
}
```

→ Value of MQ register represents quotient and AC remainder

Example of X/Y division

Restoring division

- Dividend x=1010 and divisor y=0011

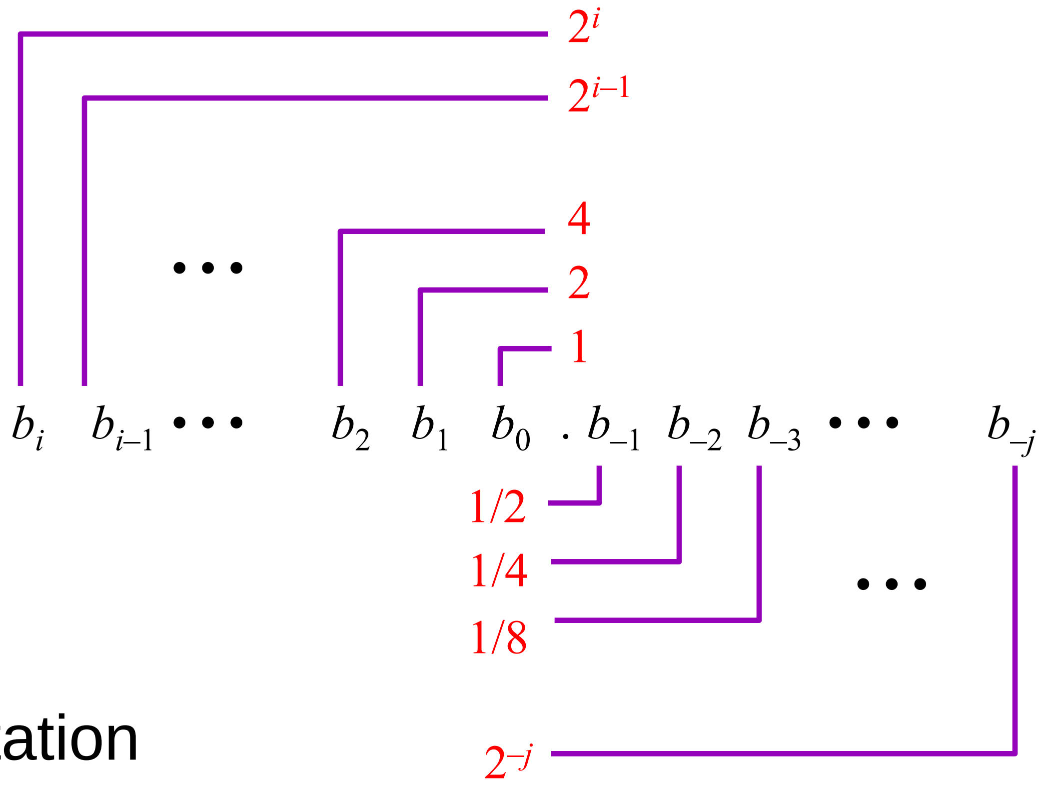
i	operation	AC	MQ	B	comment
		0000	1010	0011	initial setup
1	**SL**	0001	0100		
	nothing	0001	0100		the if condition not true
2	**SL**	0010	1000		
		0010	1000		the if condition not true
3	**SL**	0101	0000		r ≥ y
	AC = AC – B; MQ_0 = 1;	0010	0001		
4	**SL**	0100	0010		r ≥ y
	AC = AC – B; MQ_0 = 1;	0001	0011		end of the cycle

- **x : y = 1010 : 0011 = 0011 reminder 0001, (10 : 3 = 3 reminder 1)**

*Real numbers

a their starage in computers

Fractional Binary Numbers



Reprezentation

righth bits are fractions 2

$$\sum_{k=-j}^{i} b_k \cdot 2^k$$

Fractional numbers

Value *Representation*

5-3/4	101.11_2
2-7/8	10.111_2
63/64	0.111111_2

Operation

Dividing by 2 - shift right

Multiplying by 2 - shift left

Numbers below $0.111111\ldots_2$ are less than 1.0

$$1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$$

Binary → Decadic

$23.47 = 2\times10^{1} + 3\times10^{0} + 4\times10^{-1} + 7\times10^{-2}$

↑ decimal point

$10.01_{two} = 1\times2^{1} + 0\times2^{0} + 0\times2^{-1} + 1\times2^{-2}$

↑ binary point

$= 1\times2 + 0\times1 + 0\times\frac{1}{2} + 1\times\frac{1}{4}$

$= 2 + 0.25 = 2.25$

Scientific notation

Decadic:

-123,000,000,000,000 → -1.23×10^{14}

0.000 000 000 000 000 123 → $+1.23 \times 10^{-16}$

Binary:

110 1100 0000 0000 → $1.1011 \times 2^{14} = 29696_{10}$

-0.0000 0000 0000 0001 1011 → -1.1101×2^{-16}

$= -2.765655517578125 \times 10^{-5}$

Beware

Finite decadic number ➔ infinity binary number

Example:

0.1_{ten} ➔ 0.2 ➔ 0.4 ➔ 0.8 ➔ 1.6 ➔ 1.2 ➔ 3.2 ➔ 6.4 ➔ 12.6 ➔ …

$0.1_{10} = 0.00011001100110011\ldots_2$

Example 0.1_{10} to real

$0.1_{10} = 0.0\underline{00110011....}_2 =$

```
0.00011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011
0011...
```

Real numbers

Limits

exact representation only $x/2^k$

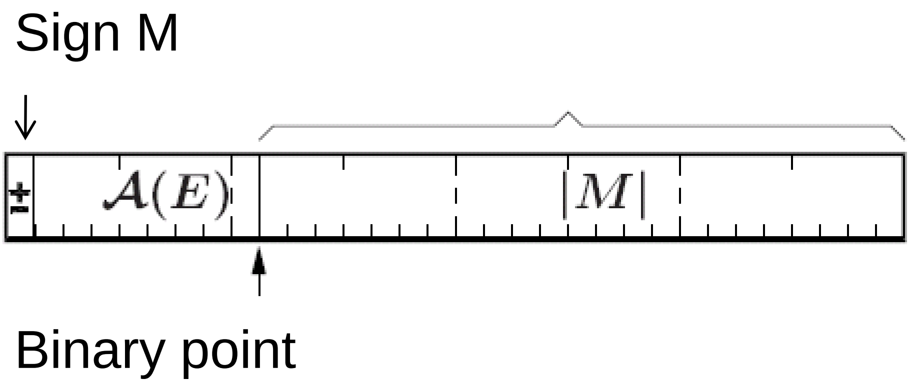
Other numbers are inexact

Value	*Binary float*
1/3	$0.0101010101[01]..._2$
1/5	$0.001100110011[0011]..._2$
1/10	$0.0001100110011[0011]..._2$

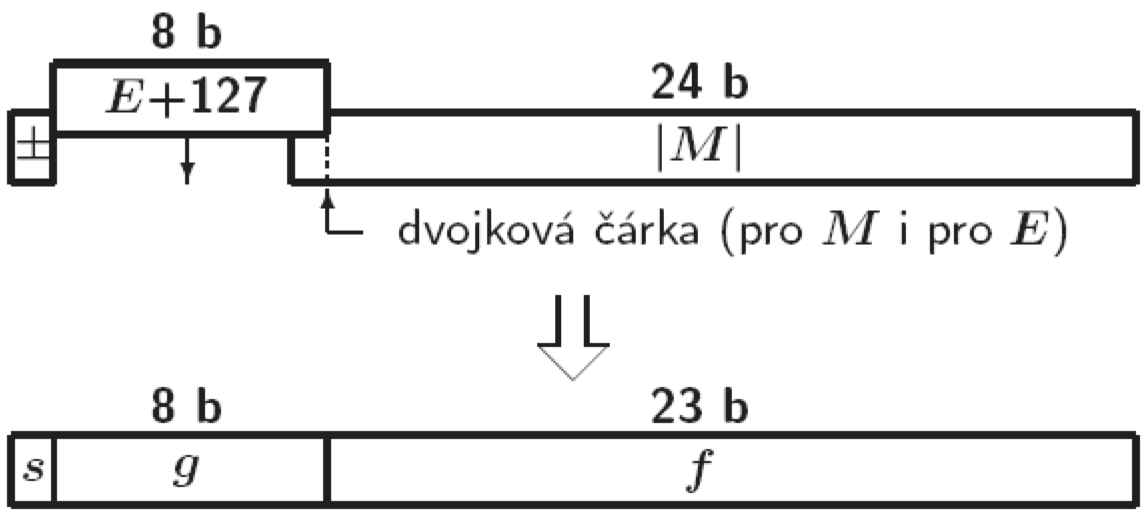
Type Float

Mantissa: direct code — sign and absolute value

Exponent: additive code
(K-excess for float +127, for double +1023).



ANSI/IEEE Std 754-1985 — simple — 32b



ANSI/IEEE Std 754-1985 — double— 64b

g . . . 11b *f* . . . 52b

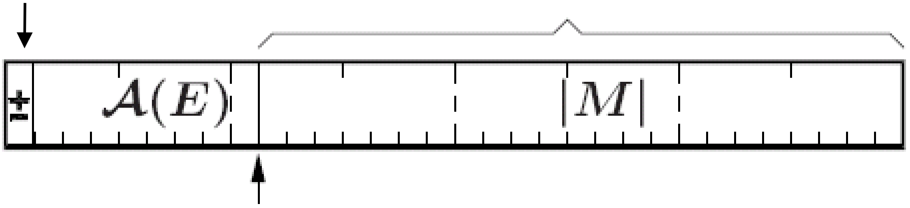
The representation/encoding of floating point number

- Mantissa encoded as the sign and absolute value (magnitude) – equivalent to the direct representation
- Exponent encoded in biased representation (K=127 for single precision)
- The implicit leading one can be omitted due to normalization of m ∈ ⟨1, 2) – 23+1 implicit bit for single

$X = -1^{s}\, 2^{A(E)-127}\, m$ where $m \in \langle 1, 2)$

$m = 1 + 2^{-23} M$

- Sign of M



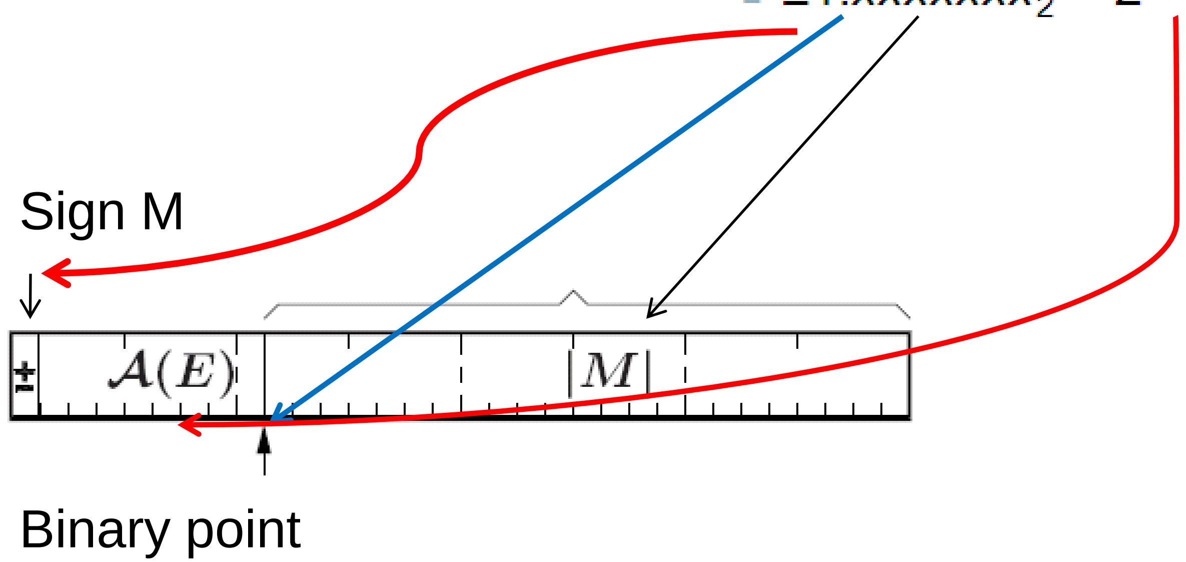
- Radix point position for E and M

Examples

- -2.34×10^{56} ← normalized
- $+0.002 \times 10^{-4}$ ← not normalized
- $+987.02 \times 10^{9}$ ← not normalized

binary

- $\pm 1.xxxxxxx_2 \times 2^{yyyy}$



IEEE-754 conversion float

- Convert -12.625_{10} IEEE-754 float format.
- Step #1: Convert $-12.625_{10} = -1100.101_2$ = 101 / 8
- Step #2: Normalize $-1100.101_2 = -1.100101_2 * 2^3$
- Step #3:

Fill sign -> +/- 0/1.

Expoment + 127 -> 130 -> 1000 0010 .

Leading bit of mantissa is hidden ->

1 1000 0010 . 1001 0100 0000 0000 0000 0000 000

Example: 0.75

$0.75_{10} = 0.11_2 = 1.1 \times 2^{-1} = 3/4$

$1.1 = 1.F \rightarrow F = 1$

$E - 127 = -1 \rightarrow E = 127 - 1 = 126 = 01111110_2$

$S = 0$

00111111010000000000000000000000 =
0x3F400000

Example 0.1_{10} to float

$0.1_{10} = 0.000110011\ldots_2$

$= 1.10011_2 \times 2^{-4} = 1.F \times 2^{E-127}$

$F = 10011$ $-4 = E - 127$

$E = 127 - 4 = 123 = 01111011_2$

0011 1101 1100 1100 1100 1100 1100 1100 *1100 11..*

0x3DCCCCCD, proč je D ?

Special numbers NaN, +Inf a -Inf

- If the result of the mathematical operation for a given input is not defined (log -1), or the result is ambiguous as 0/0, + Inf -Inf, then the NaN (Not-a-Number) value is stored, the exponent is set to all ones, and mantissa is non zero.
- The result of only overflow from the range is epresented by infinity (+ Inf or -Inf), the exponent is all ones and mantissa contain zero.

NaN	`0 11111111` *mantissa !=0*	**NaN**

Infinity

+	`0 11111111 00000000000000000000000`	**+Inf**
-	`1 11111111 00000000000000000000000`	**-Inf**

Normalized and denormalized numbers

If the exponent is between 1 and 254, a normal real number is represented.

If the exponent is 0:

- if fraction is 0, then value = 0.
- if fraction is not zero, it represents a denormalized number.

$b_1\ b_2\ \ldots\ b_{23}$ represents $0.\ b_1\ b_2\ \ldots\ b_{23}$ rather than $1.b_1 b_2\ \ldots\ b_{23}$

Why? To reduce the chance of underflow.

Denormals?

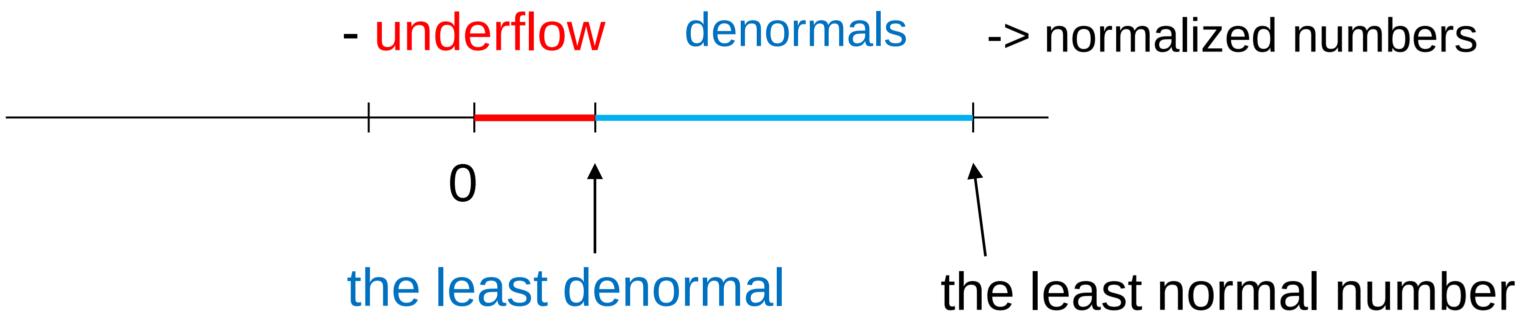
- The purpose of introducing denormalized numbers is to extend the representation of numbers that are closer to zero, ie numbers of very small (in the figure below the area is labeled blue).

- Denormalized numbers have a zero exponent, and the hidden bit before the command line is implicitly zero.

- The price is the necessity of special treatment of the case zero exponent, nonzero mantisa -> denormalized numbers support only some implementations. (Intel co-processors have)

Hidden 1

- For each standard number, the most important mantissa bit is 1, thus, it does not need to be stored.
- If the value is exponent field is 0, then the number is "denormalized", hidden bit is 0.
- Denormals allow you to maintain a resolution ranging from the smallest normalized number to zero



Denormals - to be, or to be not ?

Denormal computations use hardware and/or operating system resources to handle denormals; these can cost hundreds of clock cycles.
Denormal computations take much longer to calculate than normal computations.

There are several ways **to avoid denormals** and increase the performance of your application:

- Scale the values into the normalized range.
- Use a higher precision data type with a larger range.
- Flush denormals to zero.

[Source: https://software.intel.com/en-us/node/523326]

Overview

Type	sign (31)	exponent(8) (30–23)	fraction (23-bit) (22–0)	Hex	Watch in Windows®	Valule
Zero	0	00000000	00000000000000000000000	0 0 0 0 0 0 0 0	0.00000000	0
One	0	01111111	00000000000000000000000	3 F 8 0 0 0 0 0	1.00000000	1
Minus One	1	01111111	00000000000000000000000	B F 8 0 0 0 0 0	-1.0000000	−1.0
Smallest denormalized number	*	00000000	00000000000000000000001	0 0 0 0 0 0 0 1	1.401e-045#DEN	$\pm 2^{-23} \times 2^{-126} = \pm 2^{-149} \approx \pm 1.4 \times 10^{-45}$
"Middle" denormalized number	*	00000000	10000000000000000000000	0 0 4 0 0 0 0 0	5.877e-039#DEN	$\pm 2^{-1} \times 2^{-126} = \pm 2^{-127} \approx \pm 5.88 \times 10^{-39}$
Largest denormalized number	*	00000000	11111111111111111111111	0 0 7 F F F F F	1.175e-038#DEN	$\pm (1 - 2^{-23}) \times 2^{-126} \approx \pm 1.18 \times 10^{-38}$
Smallest normalized number	*	00000001	00000000000000000000000	0 0 8 0 0 0 0 0	1.1754944e-038	$\pm 2^{-126} \approx \pm 1.18 \times 10^{-38}$
Largest normalized number	*	11111110	11111111111111111111111	7 F 7 F F F F F	3.4028235e+038	$\pm (2 - 2^{-23}) \times 2^{127} \approx \pm 3.4 \times 10^{38}$
Positive infinity	0	11111111	00000000000000000000000	7 F 8 0 0 0 0 0	1.#INF000	$+\infty$
Negative infinity	1	11111111	00000000000000000000000	F F 8 0 0 0 0 0	-1.#INF000	$-\infty$
Not a number	*	11111111	non-zero	7 F 8 0 0 0 0 1	1.#QNAN00	NaN

* Sign bit can be either 0 or 1 .

Figure: Floating-point Binary

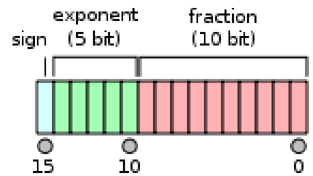
Short and Long IEEE 754 Formats: Features

Some features of ANSI/IEEE standard floating-point formats

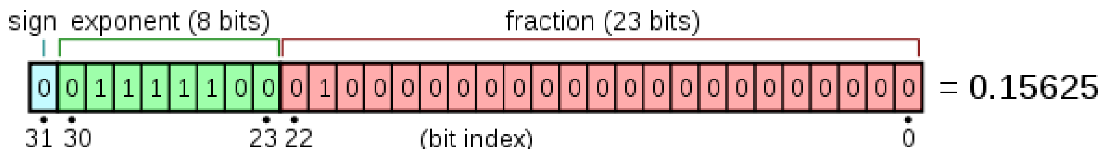
Feature	**Single/Short**	**Double/Long**
Word width in bits	32	64
Significand in bits	23 + 1 hidden	52 + 1 hidden
Significand range	$[1, 2 - 2^{-23}]$	$[1, 2 - 2^{-52}]$
Exponent bits	8	11
Exponent bias	127	1023
Zero (±0)	e + bias = 0, $f = 0$	e + bias = 0, $f = 0$
Denormal	e + bias = 0, $f \neq 0$ represents $\pm 0.f \times 2^{-126}$	e + bias = 0, $f \neq 0$ represents $\pm 0.f \times 2^{-1022}$
Infinity (±∞)	e + bias = 255, $f = 0$	e + bias = 2047, $f = 0$
Not-a-number (NaN)	e + bias = 255, $f \neq 0$	e + bias = 2047, $f \neq 0$
Ordinary number	e + bias $\in [1, 254]$ $e \in [-126, 127]$ represents $1.f \times 2^e$	e + bias $\in [1, 2046]$ $e \in [-1022, 1023]$ represents $1.f \times 2^e$
min	$2^{-126} \cong 1.2 \times 10^{-38}$	$2^{-1022} \cong 2.2 \times 10^{-308}$
max	$\cong 2^{128} \cong 3.4 \times 10^{38}$	$\cong 2^{1024} \cong 1.8 \times 10^{308}$

IEEE 754 Formats

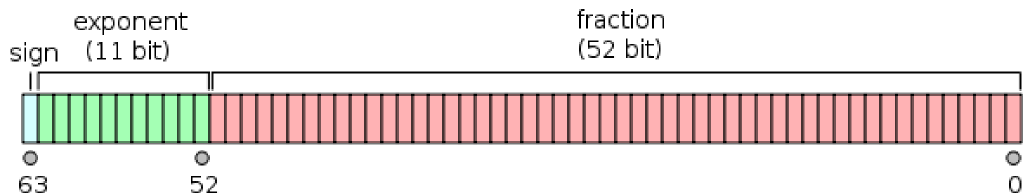
Half precision (binary16)



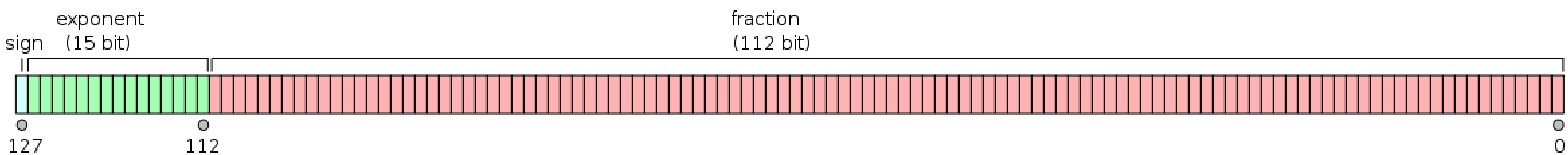
Single precision (binary32)



Double precision (binary64)

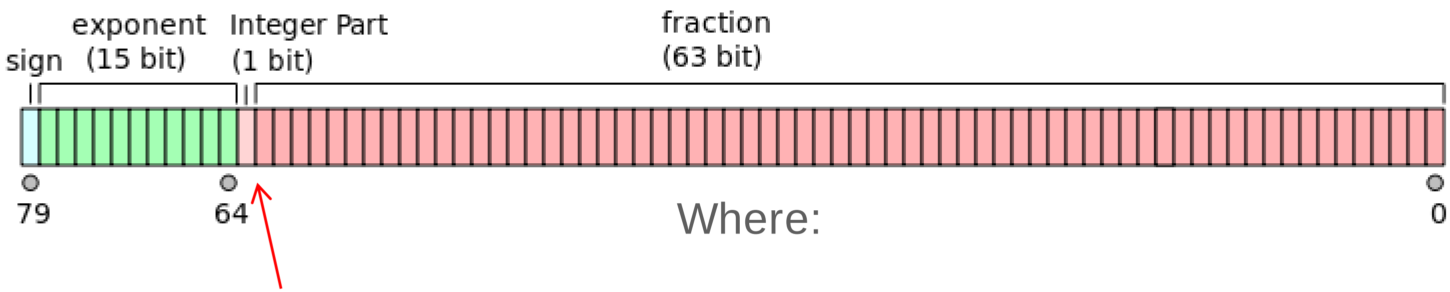


Quadruple precision (binary128)



Source: Herbert G. Mayer, PSU

X86 Extended precision (80 bits)



Bit 1. není skrytý!

Where:

- b = the bias
- n = the number of bits in the exponent

More simply, the biases are shown in the table below:

$$b = \frac{2^n}{2} - 1$$

Or, equivalently:

$$b = (2^{n-1}) - 1$$

Type	Bits	Bias
Half	5	15
Single	8	127
Double	11	1023
Extended	15	16383
Quad	15	16383

Source: Herbert G. Mayer, PSU

*Real number

and their storage in computers

Storage of numbers in memory

32bit hex number: 1234567

Big Endian - downto

address in memory `0x100 0x101 0x102 0x103`

		0x100	0x101	0x102	0x103		
		01	23	45	67		

Little Endian - to

address in memory `0x100 0x101 0x102 0x103`

		0x100	0x101	0x102	0x103		
		67	45	23	01		

Check storage type

- *when numbers are transferred between computers*
- *when single bytes of numbers are picked up*

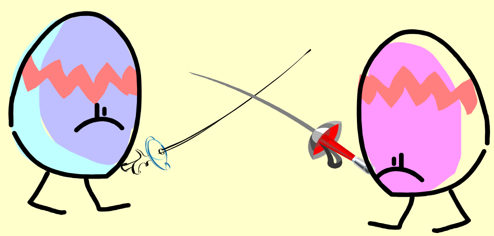
Storage number in memory

Big Endian - downto

		0x100	0x101	0x102	0x103		
		01	23	45	67		

Little Endian - to

		0x100	0x101	0x102	0x103		
		67	45	23	01		



Little Endien comes from the book Gulliver's Travels , Jonathon Swift 1726, in which denote one of the two feuding factions of Lilliputs. Her followers ate eggs from the narrower end to a wider, while the **Big Endien** proceeded in reverse. A war could not be long in coming ...

Remember, how war had ended?

1st seminaries

```c
/* Simple program to examine how are different data types encoded in memory */
#include <stdio.h>
/** The macro determines size of given variable and then
* prints individual bytes of the value representation */
#define PRINT_MEM(a) print_mem((unsigned char*)&(a), sizeof(a))

void print_mem(unsigned char *ptr, int size)
{   int i;
    printf("address = 0x%08lx\n", (long unsigned int)ptr);
    for (i = 0; i < size; i++)
    {  printf("0x%02x ", *(ptr + i));  }
    printf("\n");
}
```

1st seminaries

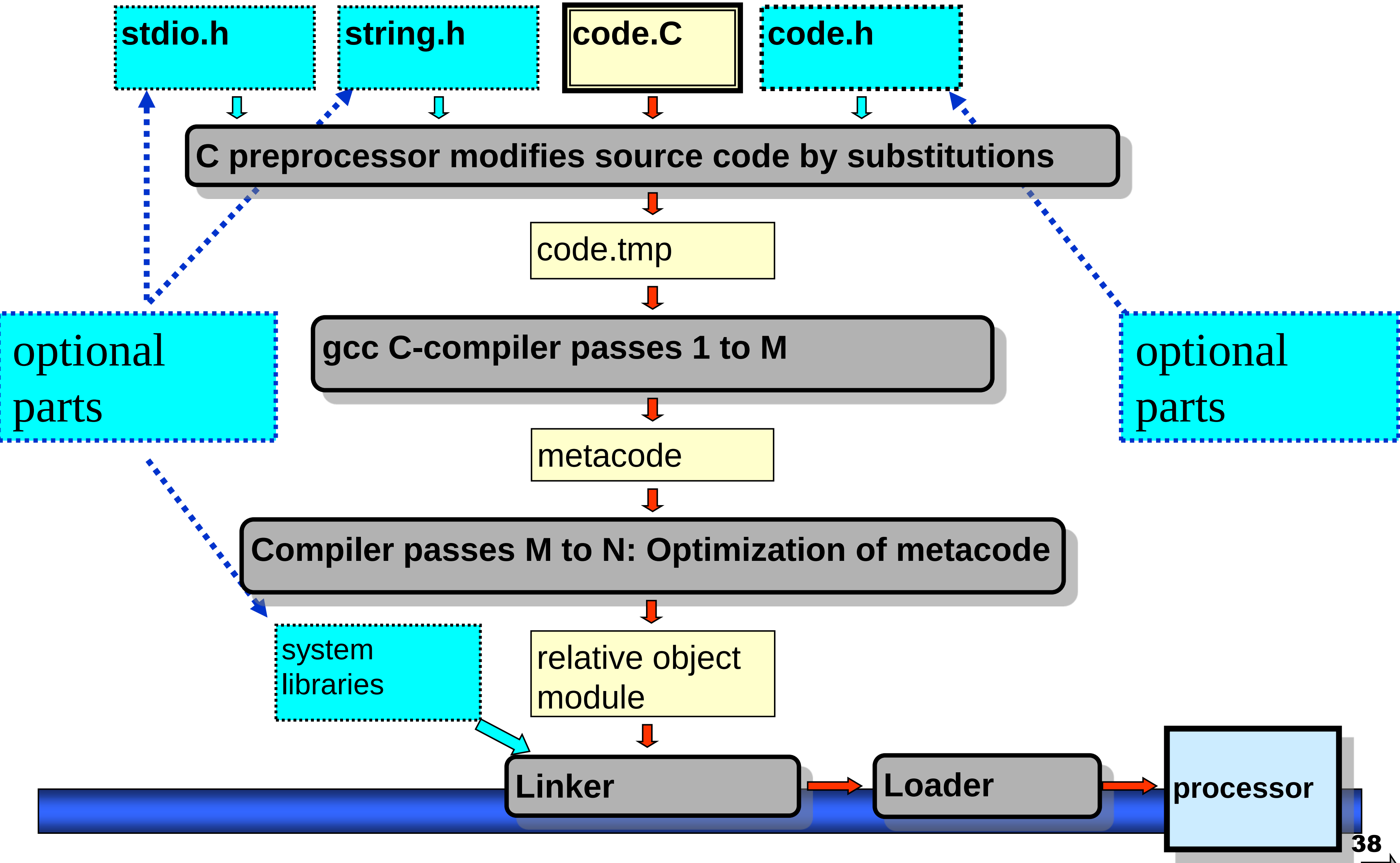
```c
int main()
{ /* try for more types: long, float, double, pointer */
  unsigned int unsig = 5;
  int sig = -5;

/* Read GNU C Library manual for conversion syntax for other types */
/* https://www.gnu.org/software/libc/manual/html_node/Formatted-Output.html */
  printf("value = %d\n", unsig);
  PRINT_MEM(unsig);

  printf("\nvalue = %d\n", sig);
  PRINT_MEM(sig);

return 0;
}
```

Basic Steps of C Compiler



C primitive types

Size	Java	C	C alternative	Range
1	boolean	any integer, true if !=0	BOOL[1]	0 to !=0
8	byte	char[2]	signed char	–128 to +127
8		unsigned char	BYTE[1]	0 to 255
16	short	int	signed short	–32768 to +32767
16		unsigned short		0 to + 65535
32	int	int	signed int	-2^31 to 2^31-1
32		unsigned int	DWORD[1]	0 to 2^32-1
64	long	long	long int	-2^63 to 2^63-1
64		unsigned long	LWORD[1]	0 to 2^64-1

1) In many implementations, it is not a standard C datatype, but only common custom for user's "#define" macro definitions, see next slides

2) Default is signed, but the best way is to specify.

Definition of BYTE and BOOL

// by substitution rule no ; and no type check

```
#define BYTE unsigned char
#define BOOL int
```

// by introducing new type, ending ; is required

- typedef unsigned char BYTE;
- typedef int BOOL;

C language has no strict type checking #define ~ typedef,
but typedef is usually better integrated into compiler.

Defining a Parameterized Macro

#define PRINT_MEM(a) print_mem((unsigned char*)&(a), sizeof(a))

Similar to a C function, preprocessor macros can be defined with a parameter list; parameters are without data types.

Syntax:

```
#define MACRONAME(parameter_list) text
```

No white space before (.

Examples:

```
#define MAXVAL(A,B) ((A) > (B)) ? (A) : (B)
```

```
#define PRINT(e1,e2)
printf("%c\t%d\n",(e1),(e2));
```

```
#define putchar(x) putc(x, stdout)
```

#define PRINT_MEM(a) print_mem((unsigned char*)&(a), sizeof(a))

Side-effects!!!

Example:

```
#define PROD1(A,B) A * B
```

Wrong result:

```
PROD1(1+3,2) → 1+3 * 2
```

Improved example with ()

```
#define PROD2(A,B) (A) * (B)

PROD2(1+3,2) → (1+3) * (2)
```

Pointer Operators

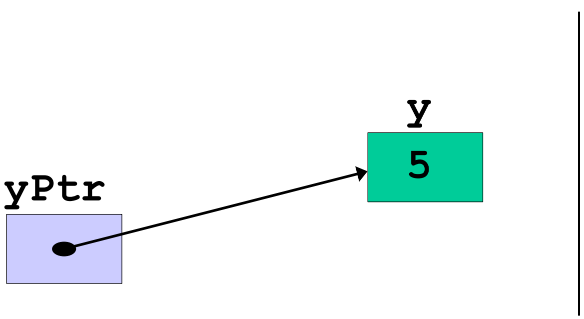
& (address operator)

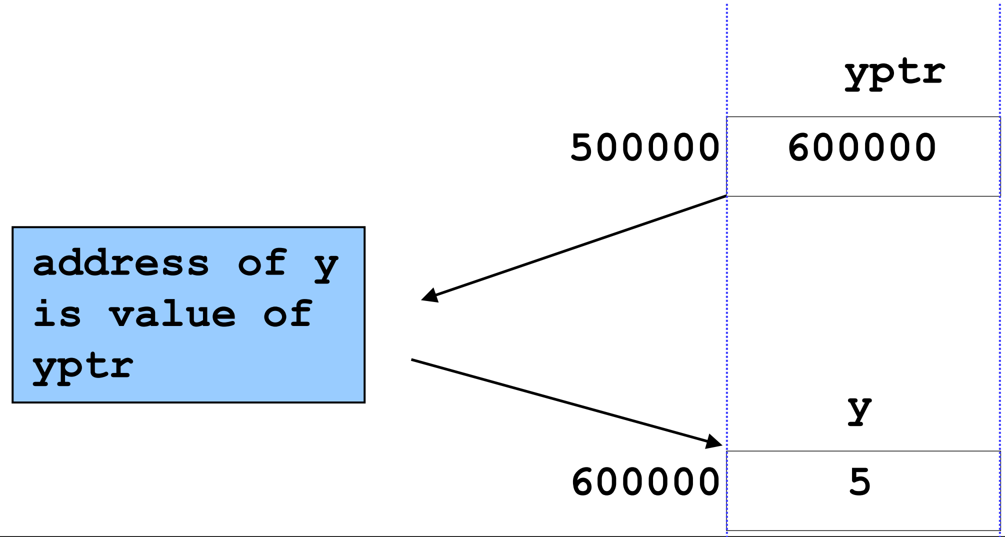
Returns the address of its operand

Example

```
int y = 5;
int *yPtr;
yPtr = &y;    // yPtr gets address of y
```

yPtr "points to" **y**





Pointer Operators

& (address operator)

Returns the address of its operand

* dereference address

Get operand stored in address location

***** and **&** are inverses
(*though not always applicable*)

Cancel each other out

```
*&myVar == myVar
```

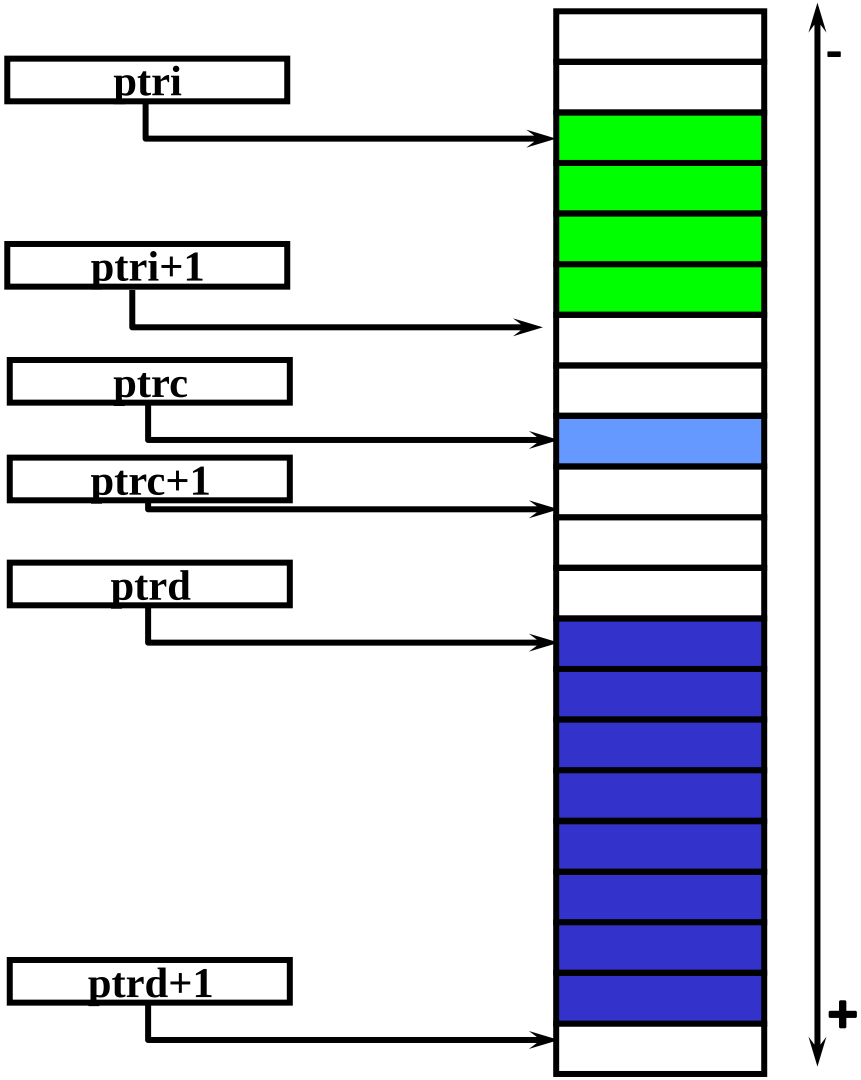
and

```
&*yPtr == yPtr
```

Size of Pointer in C-kod

```
int * ptri;
char * ptrc;
double * ptrd;
```

```
*ptrx  ≡ ptrx[0]
*(ptrx+1) ≡ ptrx[1]
*(ptrx+n) ≡ ptrx[n]
*(ptrx-n) ≡ ptrx[-n]
```




```
nr1 = sizeof (double);
nr2 = sizeof (double*);
```

nr1 != nr2

Surprise or not ???

```
int main()  { float x; double d;
x = 116777215.0;
   printf("%.3f\n", x);        // 116777216.000
   printf("%.3lf\n", x);       // 116777216.000 - it has not significance for float/double nemá l
      význam
   printf("%.3g\n", x);        // 1.17e+08
   printf("%.3e\n", x);        // 1.168e+08
   printf("%lx %f\n", x, x);  // 0 0.00000   -  Sometime I need not specify 64 bit.
   printf("%llx %f\n", x, x); // 419bd78400000000 116777216.000000
   printf("%lx %f\n", *(long *)&x, x); // 4cdebc20 116777216.00000
x = 116777216.3;  printf("%.3f\n", x); // 116777216.000  - float cut end of mantissa
d = 116777216.3;  printf("%.3f\n", d); // 116777216.300
x = 116777217.0;  printf("%.3f\n", x); // 116777216.000
x = 116777218.0;  printf("%.3f\n", x); // 116777216.000
x = 116777219.0;  printf("%.3f\n", x); // 116777216.000
x = 116777220.0;  printf("%.3f\n", x); // 116777216.000
x = 116777221.0;  printf("%.3f\n", x); // 116777224.00
return 0;
}
```