

Reprezentacja symboli w komputerze. Liczby całkowite i zmiennoprzecinkowe.

- **Bit** (*binary digit*) najmniejsza ilość informacji
 $\{0, 1\}$, wysokie/niskie napięcie
- **Kod binarny** - grupa bitów reprezentująca zbiór symboli
 - 1b $\rightarrow$ 2 symbole $\{0, 1\}$
 - 2b $\rightarrow$ 4 symbole $\{00, 01, 10, 11\}$
 - 8b $\rightarrow$ $256 = 2^8$ symboli
 - n bitów $\rightarrow$ 2^n symboli
- Do zakodowania n symboli potrzeba co najmniej $\lceil \log_2 n \rceil$ bitów
- Ile bitów potrzeba do zakodowania alfabetu polskiego?

- **Bajt** (*byte*) $1\text{B} = 8\text{b}$ (zazwyczaj ...)
Wg. standardu C: najmniejsza ilość adresowalnej pamięci
Stała `CHAR_BIT` z pliku `limits.h`
Najmniejsza porcja danych, którą może „ugryźć” komputer.
Utożsamiany ze znakiem (typ `char`).
- Ile pamięci można zaadresować?
 $16\text{b} \rightarrow 2^{16}\text{B} = 64\text{KB}$
 $32\text{b} \rightarrow 2^{32}\text{B} = 4294967296\text{B} \sim 4\text{GB}$
 $64\text{b} \rightarrow 2^{64}\text{B} \sim 16\text{EB}$

- układ SI: $1k = 10^3 = 1000 \neq 1024 = 2^{10} = 1K$
- 1997 EIC, przedrostki dwójkowe - raczej nieużywane

IEC		podstawa						SI	
nazwa	symbol	2	16	różnica	10			nazwa	symbol
kibi	Ki	2^{10}	$16^{2.5}$	400_{16}	2,40%	1 024	$> 10^3$	kilo	k
mebi	Mi	2^{20}	16^5	$10\ 0000_{16}$	4,86%	1 048 576	$> 10^6$	mega	M
gibi	Gi	2^{30}	$16^{7.5}$	$4000\ 0000_{16}$	7,37%	1 073 741 824	$> 10^9$	giga	G
tebi	Ti	2^{40}	16^{10}	$100\ 0000\ 0000_{16}$	9,95%	1 099 511 627 776	$> 10^{12}$	tera	T
pebi	Pi	2^{50}	$16^{12.5}$	$4\ 0000\ 0000\ 0000_{16}$	12,59%	1 125 899 906 842 624	$> 10^{15}$	peta	P
eksbi	Ei	2^{60}	16^{15}	$1000\ 0000\ 0000\ 0000_{16}$	15,29%	1 152 921 504 606 846 976	$> 10^{18}$	eksa	E
zebi	Zi	2^{70}	$16^{17.5}$	$40\ 0000\ 0000\ 0000\ 0000_{16}$	18,06%	1 180 591 620 717 411 303 424	$> 10^{21}$	zetta	Z
jobi	Yi	2^{80}	16^{20}	$1\ 0000\ 0000\ 0000\ 0000\ 0000_{16}$	20,89%	1 208 925 819 614 629 174 706 176	$> 10^{24}$	jotta	Y

<http://www.wikipedia.org>

$$x_{n-1}x_n \dots x_0 = \sum_{i=0}^{n-1} x_i a^i$$

a - baza (podstawa) systemu

DZIESIĘTNY

$$46532_{(10)} = 4 \cdot 10^4 + 6 \cdot 10^3 + 5 \cdot 10^2 + 3 \cdot 10^1 + 2 \cdot 10^0$$

DWÓJKOWY (BINARNY)

$$10011_{(2)} = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 19_{(10)}$$

ÓSEMKOWY (OKTALNY)

$$351_{(8)} = 3 \cdot 8^2 + 5 \cdot 8^1 + 1 \cdot 8^0 = 233_{(10)}$$

SZESNASTKOWY (HEKSADECYMALNY)

$$3FC_{(16)} = 3 \cdot 16^2 + 15 \cdot 16^1 + 12 \cdot 16^0 = 1020_{(10)}$$

A=10, B=11, C=12, D=13, E=14, F=15

$$1111110101111110_{(2)} = \text{FD7E}_{(16)} = 176576_{(8)} = 64894_{(10)}$$

BINARNY NA SZESNASTKOWY

1 cyfrze odpowiadają 4 bity

1111	1101	0111	1110
F	D	7	E

BINARNY NA ÓSEMKOWY

1 cyfrze odpowiadają 3 bity

1	111	110	101	111	110
1	7	6	5	7	6

LICZBY ÓSEMKOWE I SZESNASTKOWE W C

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int a = 10;          /* dec */
6     int b = 010;         /* oct */
7     int c = 0x10;        /* hex */
8
9     printf("dec: %d %d %d\n", a, b, c);
10    printf("oct: %o %o %o\n", a, b, c);
11    printf("hex: %x %x %x\n", a, b, c);
12
13    return 0;
14 }
```

PRZELICZANIE NA INNY SYSTEM POZYCYJNY

Problem: wyrazić liczbę $x_{(10)}$ w dowolnym systemie pozycyjnym o podstawie $p \in \{2, \dots, 10\}$.

$$\begin{array}{r|l} 139 \% 2 & 1 \\ 69 \% 2 & 1 \\ 34 \% 2 & 0 \\ 17 \% 2 & 1 \\ 8 \% 2 & 0 \\ 4 \% 2 & 0 \\ 2 \% 2 & 0 \\ 1 \% 2 & 1 \end{array}$$

$$\begin{array}{r|l} 139 \% 8 & 3 \\ 17 \% 8 & 1 \\ 2 \% 8 & 2 \end{array}$$

$$\begin{array}{r|ll} 139 \% 16 & 11 & B \\ 8 \% 16 & 8 & \end{array}$$

$$139_{(10)} = 10001011_{(2)} = 213_{(8)} = 8B_{(16)}$$

Algorytm 1 Zapis liczby dziesiętnej w innej podstawie

Dane wejściowe: liczba całkowita przeliczana $x \geq 0$ oraz p podstawa docelowego systemu

Wynik: ciąg $\{t_{n-1}, \dots, t_1, t_0\}$ reprezentujący zapis w nowym systemie

1: $i \leftarrow 0$

2: **dopóki** $x \neq 0$ **wykonuj**

3: $t_i \leftarrow x \bmod p$

4: $x \leftarrow \lfloor \frac{x}{p} \rfloor$

5: $i \leftarrow i + 1$

6: **zwróć** $\{t_{i-1}, \dots, t_1, t_0\}$

▷ w odwrotnej kolejności

```
1  int zmien_podstawe(int x, int *t, int p)
2  {
3      int i=0;
4
5      while( x != 0 )
6      {
7          t[i] = x % p;
8          x = x / p;
9          i = i + 1;
10     }
11     return i;
12 }
```

 dec2all.c

Operator / dla liczb całkowitych dzieli bez reszty!

NATURALNY KOD BINARNY (NKB)

$$b_{n-1} \dots b_1 b_0 = \sum_{i=0}^{n-1} b_i \cdot 2^i$$

- liczby bez znaku
- zakres $[0, 2^n - 1]$
- unsigned int
- unsigned char
- operacje arytmetyczne przeprowadza się podobnie jak w systemie dziesiętnym
- John Napier, XVI w.

LICZBA CAŁKOWITA ZE ZNAKIEM W KOMPETERZE

U2, KOD UZUPEŁNIEŃ DO DWÓCH

$$b_{n-1} \dots b_1 b_0 = -b_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} b_i \cdot 2^i$$

- liczby ze znakiem
- najstarszy bit b_{n-1} odpowiada za znak
- zakres $[-2^{n-1}, 2^{n-1} - 1]$
- `int`, `char`
- operacje arytmetyczne przeprowadza się podobnie jak w NKB
- Pascal, uzupełnienie do 10, XV w.

LICZBA CAŁKOWITA W KOMPUTERZE

bity	NKB	U2
00000000	0	0
00000001	1	1
⋮		
01111110	126	126
01111111	127	127
10000000	128	-128
10000001	129	-127
⋮		
11111110	254	-2
11111111	255	-1

NADMIAR ARYTMETYCZNY

$$\begin{array}{r} 127 \\ \begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \hline \end{array} \\ + \begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ \hline \end{array} \\ \hline \end{array} = \begin{array}{|c|c|c|c|c|c|c|c|} \hline 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline \end{array} \quad -128$$

- **Nadmiar** (*overflow*) - przekroczenie zakresu wynikające ze skończonej liczby bitów reprezentujących liczbę
- Zazwyczaj błąd nie jest sygnalizowany.
- Dodawanie dużych liczb dodatnich (lub odejmowanie liczb ujemnych). W U2 widoczna zmiana znaku.
- Próba zapisu liczby typu bardziej pojemnego do typu mniej pojemnego (np. float do char)
- Dodawanie liczby ujemnej i dodatniej nie powoduje nadmiaru
- 4 czerwiec 1996, pierwszy lot rakiety Ariane 5 zakończony zniszczeniem w skutek nadmiaru, 370 mln \$

KOD STAŁOPRZECINKOWY

$$3,14_{(10)} = 3 \cdot 10^0 + 1 \cdot 10^{-1} + 4 \cdot 10^{-2}$$

$$11,001_{(2)} = 1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} = 3,125_{(10)}$$

ZAPIS ZMIENNOPRZECINKOWY

$$L = m \cdot p^c$$

c - cecha

p - podstawa (baza)

m - mantysa

$$3,14 \cdot 10^0 = 0,0314 \cdot 10^2 = 314 \cdot 10^{-2}$$

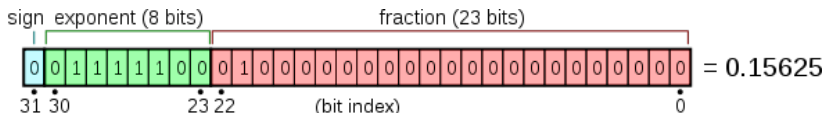
Normalizacja

$$1 \leq |m| < p$$

LICZBY ZMIENNOPOZYCYJNE IEEE-754

pojedyncza precyzja (*single precision*), float

$$x = (-1)^s \cdot 1.m \cdot 2^{(c-127)}$$



$$s = 0$$

$$1.m = \left(1 + \sum_{i=1}^{23} b_{23-i} \cdot 2^{-i}\right) = 1 + 2^{-2} = 1.25$$

$$2^{(c-127)} = 2^{(124-127)} = 2^{-3} = 0.125$$

Podobne rozwiązanie stosował Konrad Zuse, 1936 r.

http://en.wikipedia.org/wiki/Single-precision_floating-point_format

POJEDYNCZA PRECYZJA

- zakres $\pm 3.4 \cdot 10^{38}$
- precyzja 7 cyfr dziesiętnych

PODWÓJNA PRECYZJA (*double precision*)

- double
cecha 11 bitów, mantysa 52 bity, bias 1024
- zakres $\pm 1.8 \cdot 10^{308}$
- precyzja 15 cyfr znaczących

ZNACZENIE SPECJALNE BITÓW IEEE-754

cecha	mantysa	znaczenie
$\neq 0 \dots 0$	dowolne	znormalizowana
$\neq 1 \dots 1$	dowolne	znormalizowana
$0 \dots 0$	$\neq 0 \dots 0$	zdenormalizowana $(-1)^s \cdot 0.m \cdot 2^{-127}$
$0 \dots 0$	$0 \dots 0$	± 0
$1 \dots 1$	$\neq 1 \dots 1$	nieliczby NaN
$1 \dots 1$	$0 \dots 0$	nieskończoność Inf, -Inf

WYNIKI OPERACJI Z WARTOŚCIAMI SPECJALNYMI

Operacja	Wynik
$\pm x / \pm \infty$	0
$\pm \infty \cdot \pm \infty$	$\pm \infty$
$\pm x / 0$	$\pm \infty$
$\infty + \infty$	∞
$\pm 0 / \pm 0$	NaN
$\infty - \infty$	NaN
$\pm \infty / \pm \infty$	NaN
$\pm \infty \cdot 0$	NaN

NIEDOKŁADNOŚĆ REPREZENTACJI

```
1  #include <stdio.h>
2
3  int main()
4  {
5      float x = 0.1;
6
7      if(x == 0.1 ) printf ("OK, %f jest rowne 0.1\n", x);
8      else printf("Nie OK, %f nie jest rowne 0.1\n", x);
9
10     return 0;
11 }
```

 prec.c

NIEDOKŁADNOŚĆ REPREZENTACJI

```
1  #include <stdio.h>
2
3  int main()
4  {
5      float x = 0.1;
6
7      if(x == 0.1 ) printf ("OK, %f jest rowne 0.1\n", x);
8      else printf("Nie OK, %f nie jest rowne 0.1\n", x);
9
10     return 0;
11 }
```

 prec.c

- Niektóre liczby nie będą dokładnie reprezentowane
- Liczb zmiennoprzecinkowych nie należy przyrównywać do dokładnych wartości. Lepiej $|a - b| < \epsilon$

NIEDOKŁADNOŚĆ REPREZENTACJI

```
1  #include <stdio.h>
2  #include <math.h>
3  #define EPS 0.000001
4
5  int main()
6  {
7      float x = 0.1;
8
9      if( fabs(x-0.1) < EPS )
10         printf ("OK, %f jest rowne 0.1\n", x);
11     else
12         printf("Nie OK, %f nie jest rowne 0.1\n", x);
13
14     return 0;
15 }
```

- **Epsilon maszynowy** określa precyzję obliczeń numerycznych na liczbach zmiennopozycyjnych
- Najmniejsza liczba nieujemna, której dodanie do jedności daje wynik nierówny 1.
Najmniejsze $\epsilon > 0$ takie, że $1 + \epsilon \neq 1$
- Im mniejsza wartość epsilon maszynowego, tym większa jest względna precyzja obliczeń.
- Nie mylić z najmniejszą liczbą większą od zera

EPSILON MASZYNOWY

```
1  #include <stdio.h>
2  #include <float.h>
3
4  int main()
5  {
6      float epsilon = 1.0;
7      float p_epsilon = 1.0;
8      float x = 1.0;
9
10     printf( "epsilon          1.0 + epsilon\n" );
11
12     while (x + epsilon != 1.0)
13     {
14         printf( "%e\t%.20f\n", epsilon, (x + epsilon) );
15         p_epsilon = epsilon;
16         epsilon = epsilon / 2;
17     }
18     printf( "\nObliczony epsilon: %e\n", p_epsilon );
19     printf( "FLT_EPSILON          : %e\n", FLT_EPSILON );
20     return 0;
21 }
```

 epsilon.c

LIMITY DLA OBLICZEŃ ZMIENNOPOZYCYJNYCH W C

```
1 #include <stdio.h>
2 #include <float.h>
3
4 int main()
5 {
6     printf("Limity typu float\n");
7     printf("Najwieksza wartosc      : %e\n", FLT_MAX );
8     printf("Najmniejsza dodatnia   : %e\n", FLT_MIN );
9     printf("Epsilon maszynowy       : %e\n", FLT_EPSILON );
10
11     return 0;
12 }
```

 float.c

```
Najwieksza wartosc      : 3.402823e+38
Najmniejsza dodatnia   : 1.175494e-38
Epsilon maszynowy       : 1.192093e-07
```

- **nadmiar** (*overflow*) - przekroczenie zakresu (+Inf, -Inf)
- **niedomiar** (*underflow*) - zaokrąglenie bardzo małej liczby do 0
- redukcja cyfr przy odejmowaniu bardzo bliskich sobie liczb
- dodawanie (lub odejmowanie) dużej i małej liczby
- kolejność operacji może mieć wpływ na wynik
 $(a + b) + c \neq a + (b + c)$
- zaokrąglenia, np. liczby 0.1 nie można dokładnie reprezentować w systemie binarnym
- 25 luty 1991 r., wojna w zatoce perskiej, awaria systemu antyrakietowego Patriot (zegar rakiety tykał co 0.1 s.), zginęło 28 amerykańskich żołnierzy a 100 zostało rannych.

PRZYKŁAD: SZEREG HARMONICZNY

Problem: wyznaczyć sumę pierwszych n elementów szeregu harmonicznego

$$\sum_{i=1}^n \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \dots \xrightarrow{n \rightarrow \infty} \infty$$

```

1  #include <stdio.h>
2
3  int main()
4  {
5      float x=0.0, y=0.0;
6      int i=1, n;
7
8      printf("n="); scanf("%d", &n);
9
10     while(i <= n)
11     {
12         x = x + 1.0/i;
13         y = y + 1.0/(n-i+1);
14         i = i + 1;
15     }
16
17     printf("x=%f\ny=%f\n", x, y);
18     return 0;
19 }

```

n=150
 x=5.591181
 y=5.591181

 n=600
 x=6.974980
 y=6.974978

 n=2000
 x=8.178369
 y=8.178370

 n=500000
 x=13.690692
 y=13.699607

PRZYKŁAD: MIEJSCA ZEROWE PARABOLI

Problem: wyznaczyć miejsca zerowe wielomianu stopnia 2

$$y(x) = ax^2 + bx + c, \quad a \neq 0$$

$$\Delta = b^2 - 4ac$$

Dla $\Delta > 0$ istnieją 2 miejsca zerowe

$$x_1 = -\frac{b + \sqrt{\Delta}}{2a} \quad x_2 = -\frac{b - \sqrt{\Delta}}{2a}$$

Dla $\Delta = 0$ istnieje 1 miejsca zerowe

$$x_1 = -\frac{b}{2a}$$

Dla $\Delta < 0$ brak rzeczywistych miejsc zerowych

```

1  int miejsca_zerowe(float a, float b, float c, float *x1, float *x2)
2  {
3      float delta;
4
5      delta = b*b - 4*a*c;
6
7      if( fabs(delta) < EPS )
8      {
9          *x1=-b/a/2;
10         *x2=*x1;
11         return 1;
12     }
13
14     if(delta >= EPS)
15     {
16         delta=sqrt(delta);
17         *x1 = -(b-delta)/(2*a);
18         *x2 = -(b+delta)/(2*a);
19         return 2;
20     }
21     return 0;
22 }

```

 [miejscazerowe.c](#)

PROBLEMY Z ODEJMOWANIEM

- możliwe błędy, gdy jeden z pierwiastków bliski 0, tzn $b \approx \sqrt{\Delta}$

$$x_1 = -\frac{b + \sqrt{\Delta}}{2a} \quad x_2 = -\frac{b - \sqrt{\Delta}}{2a}$$

- np.: $a = 1, b = -100, c = 1 \implies \sqrt{\Delta} \approx 99.979997999$
- $x_1=99.99000e+01, x_2=1.000214e-02$
- $w(x_1)=0.000000e+00, w(x_2)=-1.136065e-04$

PROBLEMY Z ODEJMOWANIEM

- możliwe błędy, gdy jeden z pierwiastków bliski 0, tzn $b \approx \sqrt{\Delta}$

$$x_1 = -\frac{b + \sqrt{\Delta}}{2a} \quad x_2 = -\frac{b - \sqrt{\Delta}}{2a}$$

- Jeśli problem jest trudny - najlepiej zmienić problem
- wzory Viete'a

$$\begin{cases} x_1 + x_2 = -\frac{b}{a} \\ x_1 x_2 = \frac{c}{a} \end{cases}$$

- $x_1=9.999000e+01$, $x_2=1.000100e-02$
- $w(x_1)=0.000000e+00$, $w(x_2)=0.000000e+00$

```

1 int miejsca_zerowe2(float a, float b, float c, float *x1, float *x2)
2 {
3     float delta;
4
5     delta = b*b - 4*a*c;
6
7     if( fabs(delta) < EPS )
8     {
9         *x1=-b/a/2;
10        *x2=*x1;
11        return 1;
12    }
13
14    if( delta >= EPS )
15    {
16        delta=sqrt(delta);
17        if(b < 0) *x1 = (delta-b)/(2*a);
18        else     *x1 = -(b+delta)/(2*a);
19        *x2 = (c/a)/ *x1;
20        return 2;
21    }
22    return 0;
23 }

```

 miejscazerowe.c

- nadmiar - przekroczenie zakresu typu
- niedomiar - zaokrąglenie do 0 w arytmetyce zmiennopozycyjnej
- $3/2$ wynosi 1, zaś $3/2.0$ wynosi 1.5
- przybliżenia arytmetyki zmiennopozycyjnej
- porównywanie wartości zmiennopozycyjnych $f \text{ abs}(a-b) < \text{EPS}$

-  Jerzy Wałaszek, „*Binarne kodowanie liczb*”,
http://edu.i-lo.tarnow.pl/inf/alg/006_bin/index.php
-  Thomas Huckle, „*Collection of Software Bugs*”,
<http://www5.in.tum.de/~huckle/bugse.html>
-  Piotr Krzyżanowski, Leszek Plaskota, Materiały do wykładu
„*Metody numeryczne*”, Uniwersytet Warszawski, WMliM,
<http://wazniak.mimuw.edu.pl/>
-  WikiBooks, „*Kursie programowania w języku C*”,
Zaawansowane operacje matematyczne,
http://pl.wikibooks.org/wiki/C/Zaawansowane_operacje_matematyczne/