

Informatyka 1 (EZ1E2008)

Politechnika Białostocka - Wydział Elektryczny
Elektrotechnika, semestr II, studia niestacjonarne I stopnia
Rok akademicki 2020/2021

Wykład nr 4 (09.04.2021)

dr inż. Jarosław Forenc

Plan wykładu nr 4

- Język C
 - pętla for
 - operatory ++ i --
- Reprezentacja liczb całkowitych
 - liczby bez znaku, liczby ze znakiem (ZM, U1, U2)
- Reprezentacja zmiennoprzecinkowa
 - zapis zmiennoprzecinkowy liczby rzeczywistej
 - postać znormalizowana liczby zmiennoprzecinkowej
 - zakres liczb zmiennoprzecinkowych
- Standard IEEE 754
 - liczby 32-bitowe, liczby 64-bitowe
 - zakres i precyzja liczb
 - wartości specjalne, operacje z wartościami specjalnymi

Przykład: suma kolejnych 10 liczb: $1+2+\dots+10$

```
#include <stdio.h>

int main(void)
{
    int suma;

    suma = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10;

    printf("Suma wynosi: %d\n", suma);

    return 0;
}
```

Suma wynosi: 55

Przykład: suma kolejnych 100 liczb: $1+2+\dots+100$

```
#include <stdio.h>

int main(void)
{
    int suma=0, i;

    for (i=1; i<=100; i=i+1)
        suma = suma + i;

    printf("Suma wynosi: %d\n", suma);

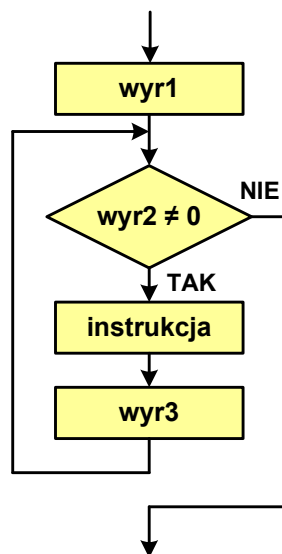
    return 0;
}
```

Suma wynosi: 5050

Język C - pętla for

```
for (wyr1; wyr2; wyr3)  
instrukcja
```

- **wyr1, wyr2, wyr3** - dowolne wyrażenia w języku C
- Instrukcja:
 - **prosta** - jedna instrukcja zakończona średnikiem
 - **złożona** - jedna lub kilka instrukcji objętych nawiasami klamrowymi



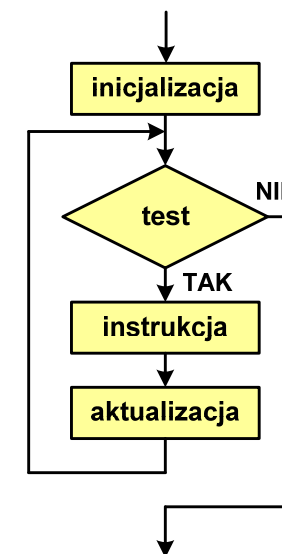
Język C - pętla for

- Najczęściej stosowana postać pętli **for**

```
int i;  
for (i = 0; i < 10; i = i + 1)  
instrukcja
```

- Instrukcja zostanie wykonana 10 razy (dla $i = 0, 1, 2, \dots, 9$)
- Funkcje pełnione przez wyrażenia

```
for (inicjalizacja; test; aktualizacja)  
instrukcja
```



Przykład: wyświetlenie tekstu 5 razy

```
#include <stdio.h>  
  
int main(void)  
{  
    int i;  
  
    for (i=0; i<5; i=i+1)  
        printf("Programowanie nie jest trudne\n");  
  
    return 0;  
}
```

```
Programowanie nie jest trudne  
Programowanie nie jest trudne  
Programowanie nie jest trudne  
Programowanie nie jest trudne  
Programowanie nie jest trudne
```

Przykład - suma liczb: 1 + 2 + ... + N

```
#include <stdio.h>  
#define N 1234  
  
int main(void)  
{  
    int i, suma=0;  
  
    for (i=1; i<=N; i++)  
        suma = suma + i;  
  
    printf("Suma %d liczb to %d\n", N, suma);  
  
    return 0;  
}
```

```
Suma 1234 liczb to 761995
```

Język C - pętla for (przykłady)

```
for (i=0; i<10; i++)  
    printf("%d ", i);
```

0 1 2 3 4 5 6 7 8 9

```
for (i=0; i<10; i++)  
    printf("%d ", i+1);
```

1 2 3 4 5 6 7 8 9 10

```
for (i=1; i<=10; i++)  
    printf("%d ", i);
```

1 2 3 4 5 6 7 8 9 10

Język C - pętla for (przykłady)

```
for (i=1; i<10; i=i+2)  
    printf("%d ", i);
```

1 3 5 7 9

```
for (i=10; i>0; i--)  
    printf("%d ", i);
```

10 9 8 7 6 5 4 3 2 1

```
for (i=-9; i<=9; i=i+3)  
    printf("%d ", i);
```

-9 -6 -3 0 3 6 9

Język C - pętla for (break, continue)

- W pętli `for` można stosować instrukcje skoku: `break` i `continue`

```
int i;  
for (i=1; i<10; i++)  
{  
    if (i%2==0)  
        continue;  
    if (i%7==0)  
        break;  
    printf("%d\n", i);  
}
```

- `continue` przerywa bieżącą iterację i przechodzi do obliczania `wyr3`

- `break` przerywa wykonywanie pętli

1 3 5

Język C - pętla for (najczęstsze błędy)

- Postawienie średnika na końcu pętli `for`

```
int i;  
for (i=0; i<10; i++);  
printf("%d ", i);
```

10

- Przecinki zamiast średników pomiędzy wyrażeniami

```
int i;  
for (i=0, i<10, i++)  
    printf("%d ", i);
```

Błąd kompilacji!

Język C - pętla for (najczęstsze błędy)

- Błędny warunek - brak wykonania instrukcji

```
int i;  
for (i=0; i>10; i++)  
    printf("%d ", i);
```

- Błędny warunek - pętla nieskończona

```
int i;  
for (i=1; i>0; i++)  
    printf("%d ", i);
```

```
1 2 3 4 5 6 7 8 9 ...
```

Język C - pętla nieskończona

```
for (wyr1; wyr2; wyr3)  
    instrukcja
```

- Wszystkie wyrażenia (*wyr1*, *wyr2*, *wyr3*) w pętli for są opcjonalne

```
for ( ; ; )  
    instrukcja
```

- pętla nieskończona

- W przypadku braku *wyr2* przyjmuje się, że jest ono **prawdziwe**

Język C - zagnieżdżanie pętli for

- Jako instrukcja w pętli **for** może występować kolejna pętla **for**

```
int i, j;  
for (i=1; i<=3; i++)           // pętla zewnętrzna  
    for (j=1; j<=2; j++)       // pętla wewnętrzna  
        printf("i: %d  j: %d\n", i, j);
```

```
i: 1  j: 1  
i: 1  j: 2  
i: 2  j: 1  
i: 2  j: 2  
i: 3  j: 1  
i: 3  j: 2
```

Język C - operator inkrementacji (++)

- Jednoargumentowy operator **++** zwiększa wartość zmiennej o 1 (nie wolno stosować go do wyrażeń)
- Operator **++** może występować jako przedrostek lub przyrostek

Zapis	Nazwa	Znaczenie
++x	preinkrementacji	wartość zmiennej jest modyfikowana przed jej użyciem
x++	postinkrementacji	wartość zmiennej jest modyfikowana po użyciu jej poprzedniej wartości

Język C - operator inkrementacji (++)

■ Przykład

```
int x = 1, y;  
y = 2 * ++x;
```

```
int x = 1, y;  
y = 2 * x++;
```

■ Kolejność operacji

```
++x      x = 2  
2 * ++x  2 * 2  
y = 2 * ++x  y = 4
```

```
2 * x      2 * 1  
y = 2 * x  y = 2  
x++       x = 2
```

■ Wartości zmiennych

```
x = 2  y = 4
```

```
x = 2  y = 2
```

Język C - operator dekrementacji (--)

- Jednoargumentowy operator -- zmniejsza wartość zmiennej o 1 (nie wolno stosować go do wyrażeń)
- Operator -- może występować jako przedrostek lub przyrostek

Zapis	Nazwa	Znaczenie
--x	predekrementacji	wartość zmiennej jest modyfikowana przed jej użyciem
x--	postdekrementacji	wartość zmiennej jest modyfikowana po użyciu jej poprzedniej wartości

Język C - operator inkrementacji (++)

- Miejsce umieszczenia operatora ++ nie ma znaczenia w przypadku instrukcji typu:

```
x++;  
++x;
```

równoważne

```
x = x + 1;
```

- Nie należy stosować operatora ++ do zmiennych pojawiających się w wyrażeniu więcej niż jeden raz

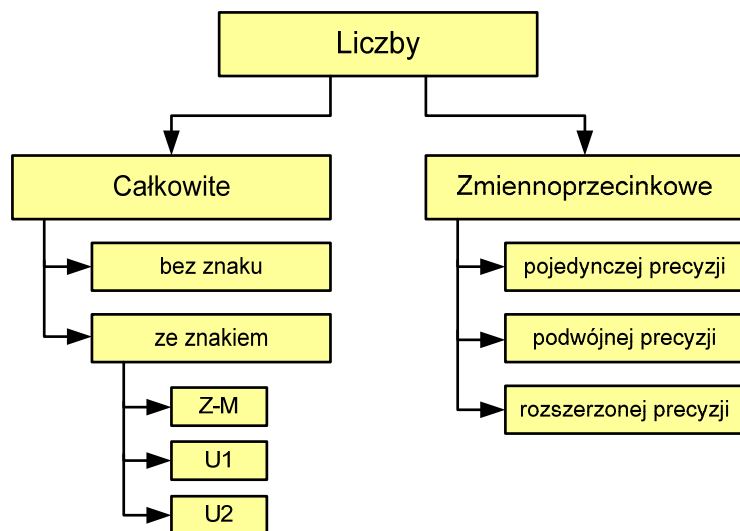
```
x = x++;  
x = ++x;
```

- Zgodnie ze standardem języka C wynik powyższych instrukcji jest **niezdefiniowany**

Język C - priorytet operatorów ++ i --

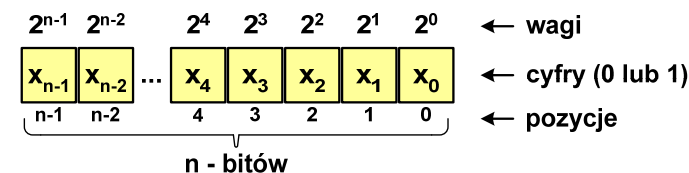
Priorytet	Operator / opis
1	++ -- (przyrostki) () [] . ->
2	++ -- (przedrostki) sizeof (typ) + - ! ~ * & (jednoargumentowe)
3	* / %
4	+ - (dwuargumentowe)
5	<< >>
6	< > <= >=
7	== !=
8	& (bitowy)
9	^

Reprezentacja liczb w systemach komputerowych



Liczby całkowite bez znaku

- Zapis liczby w systemie dwójkowym:



- Używając **n-bitów** można zapisać liczbę z zakresu:

$$X_{(2)} = \langle 0, 2^n - 1 \rangle$$

8-bitów	0 ... 255
16-bitów	0 ... 65 535
32-bity	0 ... 4 294 967 295
64-bity	0 ... 18 446 744 073 709 551 615

18 trylionów 446 miliardów 744 biliony 73 miliardy 709 milionów 551 tysięcy 615

Liczby całkowite bez znaku w języku C

- Typy zmiennych całkowitych bez znaku stosowane w języku C:

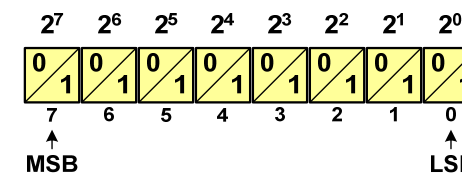
Nazwa typu	Rozmiar (bajty)	Zakres wartości
<code>unsigned char</code>	1 bajt	0 ... 255
<code>unsigned short int</code>	2 bajty	0 ... 65 535
<code>unsigned int</code>	4 bajty	0 ... 4 294 967 295
<code>unsigned long int</code>	4 bajty	0 ... 4 294 967 295
<code>unsigned long long int</code>	8 bajtów	0 ... 18 446 744 073 709 551 615

- W nazwach typów **short** i **long** można pominąć słowo **int**:

<code>unsigned short int</code>	→	<code>unsigned short</code>
<code>unsigned long int</code>	→	<code>unsigned long</code>
<code>unsigned long long int</code>	→	<code>unsigned long long</code>

Liczby całkowite bez znaku w języku C

- Typ **unsigned char** (1 bajt):



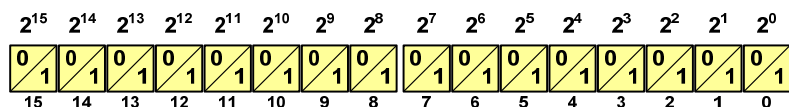
- MSB** (Most Significant Bit) - najbardziej znaczący bit, najstarszy bit, największa waga
- LSB** (Least Significant Bit) - najmniej znaczący bit, najmłodszy bit, najmniejsza waga

- Zakres wartości:

- dolna granica: $0000\ 0000_{(2)} = 00_{(16)} = 0_{(10)}$
- górną granicą: $1111\ 1111_{(2)} = FF_{(16)} = 255_{(10)}$

Liczby całkowite bez znaku w języku C

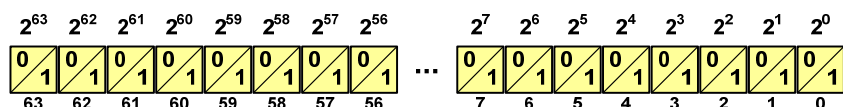
- Typ **unsigned short int** (2 bajty):



- Typ **unsigned int** (4 bajty) i **unsigned long int** (4 bajty):



- Typ **unsigned long long int** (8 bajtów):



Liczby całkowite bez znaku w języku C

```
unsigned short int:    1 0 65535
unsigned int:         1 0 4294967295
unsigned long int:   1 0 4294967295
unsigned long long int: 1 0 18446744073709551615
```

```
#include <stdio.h>

int main() /* przepełnienie zmiennej, ang. integer overflow */
{
    unsigned short int    usi = 1;
    unsigned int          ui = 1;
    unsigned long int     uli = 1;
    unsigned long long int ulli = 1;

    printf("unsigned short int:    %hu %hu %hu\n", usi, usi-1, usi-2);
    printf("unsigned int:         %u %u %u\n", ui, ui-1, ui-2);
    printf("unsigned long int:     %lu %lu %lu\n", uli, uli-1, uli-2);
    printf("unsigned long long int: %llu %llu %llu\n",
           ulli, ulli-1, ulli-2);

    return 0;
}
```

Liczby całkowite bez znaku w języku C

```
unsigned short int:    65535 0 1
unsigned int:         4294967295 0 1
unsigned long int:    4294967295 0 1
unsigned long long int: 18446744073709551615 0 1
```

```
#include <stdio.h>

int main() /* przepełnienie zmiennej, ang. integer overflow */
{
    unsigned short int    usi = 65535;
    unsigned int          ui = 4294967295;
    unsigned long int     uli = 4294967295;
    unsigned long long int ulli = 18446744073709551615;

    printf("unsigned short int:    %hu %hu %hu\n", usi, usi+1, usi+2);
    printf("unsigned int:         %u %u %u\n", ui, ui+1, ui+2);
    printf("unsigned long int:     %lu %lu %lu\n", uli, uli+1, uli+2);
    printf("unsigned long long int: %llu %llu %llu\n",
           ulli, ulli+1, ulli+2);

    return 0;
}
```

Liczby całkowite bez znaku w języku C

```
unsigned short int:    1 0 65535
unsigned int:         1 0 4294967295
unsigned long int:   1 0 4294967295
unsigned long long int: 1 0 18446744073709551615
```

```
#include <stdio.h>

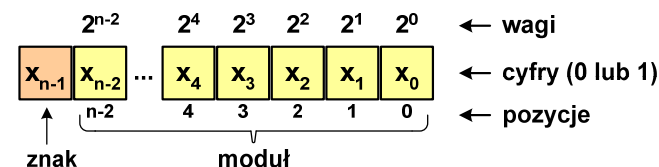
int main() /* przepełnienie zmiennej, ang. integer overflow */
{
    unsigned short int    usi = 1;
    unsigned int          ui = 1;
    unsigned long int     uli = 1;
    unsigned long long int ulli = 1;

    printf("unsigned short int:    %hu %hu %hu\n", usi, usi-1, usi-2);
    printf("unsigned int:         %u %u %u\n", ui, ui-1, ui-2);
    printf("unsigned long int:     %lu %lu %lu\n", uli, uli-1, uli-2);
    printf("unsigned long long int: %llu %llu %llu\n",
           ulli, ulli-1, ulli-2);

    return 0;
}
```

Liczby całkowite ze znakiem - kod znak-moduł

- Inne nazwy: **ZM, Z-M, SM (Signed Magnitude), S+M**
- Najstarszy bit jest bitem znaku liczby: 0 - dodatnia, 1 - ujemna
- Pozostałe bity mają takie same znaczenie jak w **NKB**



- Wartość liczby:

$$X_{(10)} = \underbrace{(x_0 \cdot 2^0 + x_1 \cdot 2^1 + x_2 \cdot 2^2 + \dots + x_{n-2} \cdot 2^{n-2})}_{\text{moduł}} \cdot \underbrace{(-1)^{x_{n-1}}}_{\text{znak}} = (-1)^{x_{n-1}} \cdot \sum_{i=0}^{n-2} x_i \cdot 2^i$$

Liczby całkowite ze znakiem - kod znak-moduł

- Liczby 4-bitowe (1 bit - znak, 3 bity - moduł) w kodzie Z-M:

Z-M	dziesiętnie	Z-M	dziesiętnie
0000	+0	1000	-0
0001	1	1001	-1
0010	2	1010	-2
0011	3	1011	-3
0100	4	1100	-4
0101	5	1101	-5
0110	6	1110	-6
0111	7	1111	-7

- dwie reprezentacje zera

+0 (0000_{ZM})

-0 (1000_{ZM})

- Zakres liczb dla n-bitów:

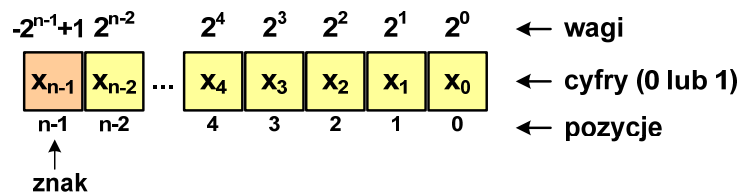
$$X_{(10)} = \langle -2^{n-1} + 1, 2^{n-1} - 1 \rangle$$

dla 8 bitów : $\langle -127 \dots 127 \rangle$

dla 16 bitów : $\langle -32767 \dots 32767 \rangle$

Liczby całkowite ze znakiem - kod U1

- Inne nazwy: U1, ZU1, uzupełnień do jedności
- Najstarszy bit jest bitem znaku liczby: 0 - dodatnia, 1 - ujemna
- Wszystkie bity liczby posiadają takie same wagi jak w NKB, oprócz pierwszego bitu, który ma wagę $-2^{n-1} + 1$



- Wartość liczby:

$$X_{(10)} = x_0 \cdot 2^0 + x_1 \cdot 2^1 + x_2 \cdot 2^2 + \dots + x_{n-2} \cdot 2^{n-2} + x_{n-1} \cdot (-2^{n-1} + 1)$$

Liczby całkowite ze znakiem - kod znak-moduł

- Zamiana liczby dziesiętnej na kod Z-M:

- liczba dodatnia

$$93_{(10)} = ?_{(ZM)}$$

- zamieniamy liczbę na NKB

$$93_{(10)} = 1011101_{(NKB)}$$

- dodajemy bit znaku

$$93_{(10)} = 01011101_{(ZM)}$$

- liczba ujemna

$$-93_{(10)} = ?_{(ZM)}$$

- zamieniamy moduł liczby na NKB

$$|-93_{(10)}| = 93_{(10)} = 1011101_{(NKB)}$$

- dodajemy bit znaku

$$-93_{(10)} = 11011101_{(ZM)}$$

Liczby całkowite ze znakiem - kod U1

- Liczby 4-bitowe (1 bit - znak, 3 bity - moduł) w kodzie U1:

U1	dziesiętnie	U1	dziesiętnie
0000	+0	1111	-0
0001	1	1110	-1
0010	2	1101	-2
0011	3	1100	-3
0100	4	1011	-4
0101	5	1010	-5
0110	6	1001	-6
0111	7	1000	-7

- liczby dodatnie zapisywane są tak samo jak w NKB
- liczby ujemne otrzymywane są poprzez bitową negację
- dwie reprezentacje zera

- Zakres liczb dla n-bitów:

$$X_{(10)} = \langle -2^{n-1} + 1, 2^{n-1} - 1 \rangle$$

dla 8 bitów : $\langle -127 \dots 127 \rangle$

dla 16 bitów : $\langle -32767 \dots 32767 \rangle$

Liczby całkowite ze znakiem - kod U1

Zamiana liczby dziesiętnej na kod U1:

- liczba dodatnia

$$93_{(10)} = ?_{(U1)}$$

- zamieniamy liczbę na NKB

$$93_{(10)} = 1011101_{(NKB)}$$

- dodajemy bit znaku: 0

$$93_{(10)} = 01011101_{(U1)}$$

- liczba ujemna

$$-93_{(10)} = ?_{(U1)}$$

- zamieniamy **moduł** liczby na U1

$$|-93_{(10)}| = 93_{(10)} = 01011101_{(U1)}$$

- negujemy wszystkie bity

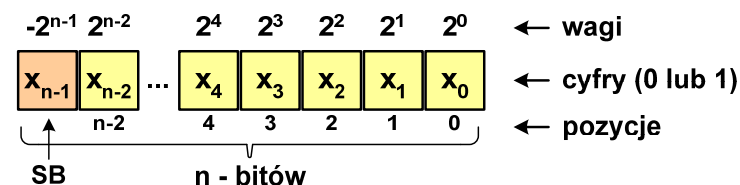
$$-93_{(10)} = 10100010_{(U1)}$$

bit znaku

Liczby całkowite ze znakiem - kod U2

- Inne nazwy: **ZU2**, **uzupełnień do dwóch**, **two's complement**

- Najstarszy bit jest bitem znaku liczby: 0 - dodatnia, 1 - ujemna



- Wartość liczby:

$$X_{(10)} = x_0 \cdot 2^0 + x_1 \cdot 2^1 + x_2 \cdot 2^2 + \dots + x_{n-2} \cdot 2^{n-2} + x_{n-1} \cdot (-2^{n-1})$$

- Kod U2 jest obecnie powszechnie stosowany w informatyce

Liczby całkowite ze znakiem - kod U2

- Liczby **4-bitowe** (1 bit - znak, 3 bity - moduł) w kodzie U2:

U2	dziesiętnie	U2	dziesiętnie
0000	0	1111	-1
0001	1	1110	-2
0010	2	1101	-3
0011	3	1100	-4
0100	4	1011	-5
0101	5	1010	-6
0110	6	1001	-7
0111	7	1000	-8

- brak podwójnej reprezentacji zera

- liczb ujemnych jest o jeden więcej niż dodatnich

- 00...000 zawsze oznacza $0_{(10)}$
11...111 zawsze oznacza $-1_{(10)}$

- Zakres liczb dla **n-bitów**:

$$X_{(10)} = \langle -2^{n-1}, 2^{n-1} - 1 \rangle$$

dla 8 bitów : $\langle -128 \dots 127 \rangle$

dla 16 bitów : $\langle -32768 \dots 32767 \rangle$

Liczby całkowite ze znakiem - kod U2

- Zamiana liczby dziesiętnej na kod U2:

- liczba dodatnia

$$75_{(10)} = ?_{(U2)}$$

- zamieniamy liczbę na NKB

$$75_{(10)} = 1001011_{(NKB)}$$

- dodajemy bit znaku: 0

$$75_{(10)} = 01001011_{(U2)}$$

- liczba ujemna

$$-75_{(10)} = ?_{(U2)}$$

- zamieniamy **moduł** liczby na U2

$$|-75_{(10)}| = 75_{(10)} = 01001011_{(U2)}$$

- negujemy wszystkie bity i dodajemy 1

$$\begin{array}{r} 01001011 \\ \text{negacja: } 10110100 \\ +1: \quad \quad \quad 1 \\ \hline -75_{(10)} = 10110101_{(U2)} \end{array}$$

Liczby całkowite ze znakiem - kod U2 w języku C

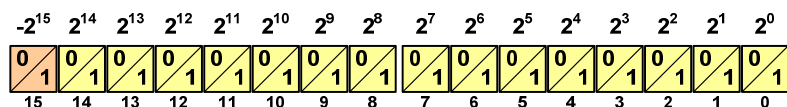
- Typy zmiennych całkowitych ze znakiem stosowane w języku C:

Nazwa typu	Rozmiar (bajty)	Zakres wartości
char	1 bajt	-128 ... 127
short int	2 bajty	-32 768 ... 32 767
int	4 bajty	-2 147 483 648 ... 2 147 483 647
long int	4 bajty	-2 147 483 648 ... 2 147 483 647
long long int	8 bajtów	-9 223 372 036 854 775 808 ... 9 223 372 036 854 775 807

- Przed nazwą każdego z powyższych typów można dodać **signed** signed char, signed short int, signed int ...
- W nazwach typów **short** i **long** można pominąć słowo **int**: short int → short, long int → long, long long int → long long

Liczby całkowite bez znaku w języku C

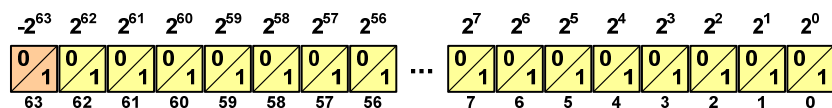
- Typ short / signed short int (2 bajty):



- Typy int / signed int (4 bajty) i long / signed long int (4 bajty):

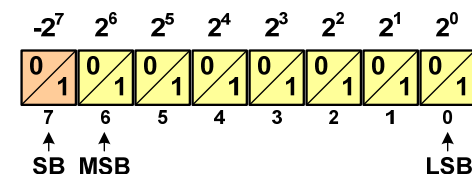


- Typ long long int / signed long long int (8 bajtów):



Liczby całkowite ze znakiem - kod U2 w języku C

- Typ char / signed char (1 bajt):



- Zakres wartości:

- dolna granica: $1000\ 0000_{(2)} = -128_{(10)}$
- górną granicą: $0111\ 1111_{(2)} = 127_{(10)}$
- inne wartości: $1111\ 1111_{(2)} = -1_{(10)}$
 $0000\ 0000_{(2)} = 0_{(10)}$

Liczby całkowite ze znakiem - kod U2 w języku C

```
short int:      32767 -32768 -32767
int:           2147483647 -2147483648 -2147483647
long int:      2147483647 -2147483648 -2147483647
long long int: 9223372036854775807 -9223372036854775808
```

```
#include <stdio.h>
int main() /* przepełnienie zmiennej, ang. integer overflow */
{
    short int    si = 32767;
    int          i  = 2147483647;
    long int     li = 2147483647;
    long long int lli = 9223372036854775807;

    printf("short int:      %hd %hd %hd\n", si, si+1, si+2);
    printf("int:           %d %d %d\n", i, i+1, i+2);
    printf("long int:      %ld %ld %ld\n", li, li+1, li+2);
    printf("long long int: %lld %lld\n", lli, lli+1);

    return 0;
}
```

Zapis zmiennoprzecinkowy liczby rzeczywistej

- Zapis bardzo dużych lub małych liczb wymaga dużej liczby cyfr
- Znacznie prostsze jest przedstawienie liczb w postaci **zmiennoprzecinkowej** (ang. **floating point numbers**)
 - $12\,000\,000\,000\,000 = 1,2 \cdot 10^{13}$
 - $0,000\,000\,000\,001 = 1,0 \cdot 10^{-12}$
- Zapis liczby zmiennoprzecinkowej ma postać:

$$L = M \cdot B^E$$

gdzie:

L - wartość liczby B - podstawa systemu
M - mantysa E - wykładnik, cecha

- notacja naukowa: $1,2e13$ $1,2e+13$ $1,2E13$ $1,2E+13$
- postać wykładnicza: $1,2 \cdot 10^{13}$

Zapis zmiennoprzecinkowy liczby rzeczywistej

$$2,43 \cdot 10^3_{(10)} = 2,43 \cdot 1000 = 2430_{(10)} \quad 6,59 \cdot 10^{-2}_{(10)} = 6,59 \cdot 0,01 = 0,0659_{(10)}$$

$$1,011 \cdot 10^{101}_{(2)} = ?_{(10)}$$

$$M = 1,011_{(2)} = 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} = 1,375_{(10)}$$

$$B = 10_{(2)} = 0 \cdot 2^0 + 1 \cdot 2^1 = 2_{(10)}$$

$$E = 101_{(2)} = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 = 1 + 4 = 5_{(10)}$$

$$1,011 \cdot 10^{101}_{(2)} = 1,375 \cdot 2^5 = 1,375 \cdot 32 = 44_{(10)}$$

$$3,121 \cdot 10^{32}_{(4)} = ?_{(10)}$$

$$M = 3,121_{(4)} = 3 \cdot 4^0 + 1 \cdot 4^{-1} + 2 \cdot 4^{-2} + 1 \cdot 4^{-3} = 3,390625_{(10)}$$

$$B = 10_{(4)} = 0 \cdot 4^0 + 1 \cdot 4^1 = 4_{(10)}$$

$$E = 32_{(4)} = 2 \cdot 4^0 + 3 \cdot 4^1 = 2 + 12 = 14_{(10)}$$

$$3,121 \cdot 10^{32}_{(4)} = 3,390625 \cdot 4^{14} = 910\,163\,968_{(10)}$$

Postać znormalizowana zapisu liczby

- Położenie przecinka w mantysie nie jest ustalone i może się zmieniać
- Poniższe zapisy oznaczają tę samą liczbę (system dziesiętny)

$$243 \cdot 10^1 = 24,3 \cdot 10^2 = 2,43 \cdot 10^3 = 0,243 \cdot 10^4$$
- Dla ujednoczenia zapisu i usunięcia wielokrotnych reprezentacji tej samej liczby, przyjęto tzw. **postać znormalizowaną** zapisu liczby
- W postaci znormalizowanej mantysa spełnia nierówność:

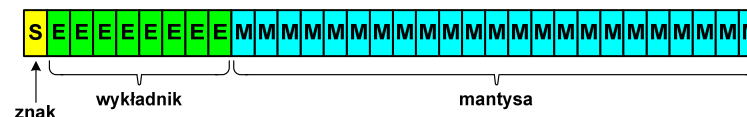
$$B > |M| \geq 1$$

Przykład:

- $2,43 \cdot 10^3$ - to jest postać znormalizowana, gdyż: $10 > |2,43| \geq 1$
- $0,243 \cdot 10^4$ - to nie jest postać znormalizowana
- $24,3 \cdot 10^2$ - to nie jest postać znormalizowana

Liczby zmiennoprzecinkowe w systemie binarnym

- Liczba bitów przeznaczonych na mantysę i wykładnik jest ograniczona



- Wartość liczby L:

$$L = (-1)^S \cdot M \cdot B^E$$

gdzie:

- S - znak liczby (ang. sign), przyjmuje wartość 0 lub 1
- M - znormalizowana mantysa (ang. mantissa), liczba ułamkowa
- B - podstawa systemu liczbowego (ang. base)
- E - wykładnik (ang. exponent), cecha, liczba całkowita

- W systemie binarnym podstawa systemu jest stała: **B = 2**

$$L = (-1)^S \cdot M \cdot 2^E$$

Przesunięcie wykładnika

- Wykładnik zapisywany jest z przesunięciem (ang. **bias**)

$$L = (-1)^S \cdot M \cdot 2^{E-BIAS}$$

gdzie:

L - wartość liczby S - znak liczby M - mantysa
E - wykładnik BIAS - przesunięcie (nadmiar)

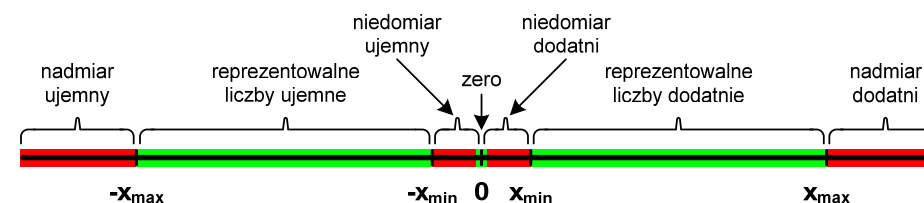
- Typowe wartości przesunięcia (nadmiaru) wynoszą:

- formatu 32-bitowy: $2^7-1 = 127_{(10)} = 7F_{(16)}$
- formatu 64-bitowy: $2^{10}-1 = 1023_{(10)} = 3FF_{(16)}$
- formatu 80-bitowy: $2^{14}-1 = 16383_{(10)} = 3FFF_{(16)}$

Zakres liczb zmiennoprzecinkowych

- Zakres liczb w zapisie zmiennoprzecinkowym:

$$\langle -x_{\max}, -x_{\min} \rangle \cup \{0\} \cup \langle x_{\min}, x_{\max} \rangle$$



- Największa i najmniejsza wartość liczby w danej reprezentacji:

$$x_{\min} = M_{\min} \cdot B^{E_{\min}}$$

$$x_{\max} = M_{\max} \cdot B^{E_{\max}}$$

Standard IEEE 754

- IEEE Std. 754-2008 - IEEE Standard for Floating-Point Arithmetic
- Standard definiuje następujące klasy liczb zmiennoprzecinkowych:

Precyzja	Długość słowa [bity]	Znak [bity]	Wykładnik		Mantysa	
			Długość [bity]	Zakres	Długość [bity]	Cyfry znaczące
Pojedyncza (Single Precision, binary32)	32	1	8	$2^{\pm 127} \approx 10^{\pm 38}$	23	7
Pojedyncza rozszerzona (Single Extended)	≥ 43	1	≥ 11	$\geq 2^{\pm 1023} \approx 10^{\pm 308}$	≥ 31	≥ 10
Podwójna (Double Precision, binary64)	64	1	11	$2^{\pm 1023} \approx 10^{\pm 308}$	52	16
Podwójna rozszerzona (Double Extended)	≥ 79	1	≥ 15	$\geq 2^{\pm 16383} \approx 10^{\pm 4932}$	≥ 63	≥ 19

Standard IEEE 754

- W przypadku liczb:
 - pojedynczej rozszerzonej precyzji (ang. Single Precision)
 - podwójnej rozszerzonej precyzji (ang. Double Precision)

standard podaje jedynie minimalną liczbę bitów pozostawiając szczegóły implementacji producentom procesorów i kompilatorów

- Bardzo popularny jest 80-bitowy format **podwójnej rozszerzonej precyzji** (Extended Precision) wprowadzony przez firmę Intel

- W 80-bitowym formacie Intela:

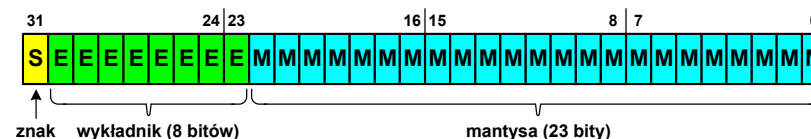
- długość słowa: 80 bitów
- znak: 1 bit
- wykładnik: 15 bitów (zakres: $2^{\pm 16383} \approx 10^{\pm 4932}$)
- mantysa: 63 bity (cyfry znaczące: 19)

Standard IEEE 754

- Standard IEEE 754 definiuje także dziesiętne typy zmiennoprzecinkowe (operujące na cyfrach dziesiętnych):
 - **decimal32** (32 bity, 7 cyfr dziesiętnych)
 - **decimal64** (64 bity, 16 cyfr dziesiętnych)
 - **decimal128** (128 bitów, 34 cyfry dziesiętnych)
- Standard IEEE 754 definiuje także:
 - sposób reprezentacji specjalnych wartości, np. nieskończoności, zera
 - sposób wykonywania działań na liczbach zmiennoprzecinkowych
 - sposób zaokrąglania liczb

Standard IEEE 754 - liczby 32-bitowe

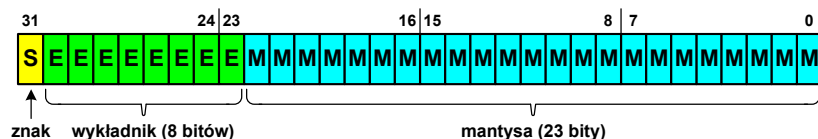
- Liczba pojedynczej precyzji przechowywana jest na 32 bitach:



- Pierwszy bit w zapisie (bit nr 31) jest **bitem znaku** (0 - liczba dodatnia, 1 - liczba ujemna)
- **Wykładnik** zapisywany jest na **8 bitach** (bity nr 30-23) z nadmiarem o wartości 127
- **Wykładnik** może przyjmować wartości od -127 (wszystkie bity wyzerowane) do 128 (wszystkie bity ustawione na 1)

Standard IEEE 754 - liczby 32-bitowe

- Liczba pojedynczej precyzji przechowywana jest na 32 bitach:



- **Mantysa** w większości przypadków jest znormalizowana
- Wartość mantysy zawiera się pomiędzy **1** a **2**, a zatem w zapisie liczby pierwszy bit jest zawsze równy 1
- Powyższy bit nie jest zapamiętywany, natomiast jest automatycznie uwzględniany podczas wykonywania obliczeń
- Dzięki pominięciu tego bitu zyskujemy dodatkowy bit mantysy (zamiast 23 bitów mamy 24 bity)

Standard IEEE 754 - liczby 32-bitowe

- Przykład:

- obliczmy wartość dziesiętną liczby zmiennoprzecinkowej

$$01000010110010000000000000000000_{(IEEE754)} = ?_{(10)}$$

- dzielimy liczbę na części

$$\begin{array}{ccc} 0 & 10000101 & 100100000000000000000000 \\ \text{S-bit znaku} & \text{E-wykładnik} & \text{M-mantysa (tylko czesc ułamkowa)} \end{array}$$

- określamy **znak liczby**

$$S = 0 \quad \text{—liczba dodatnia}$$

- obliczamy **wykładnik** (nadmiar: 127)

$$10000101_{(2)} = 128 + 4 + 1 = 133 \Rightarrow E = 133 - \underset{\text{nadmiar}}{127} = 6_{(10)}$$

Standard IEEE 754 - liczby 32-bitowe

■ Przykład (cd.):

- wyznaczamy **mantysę** dopisując na początku **1**, (część całkowita)

$$M = 1,100100000000000000000000 = \\ = 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-4} = 1 + 0,5 + 0,0625 = 1,5625_{(10)}$$

- wzór na wartość dziesiętną liczby zmiennoprzecinkowej:

$$L = (-1)^S \cdot M \cdot 2^E$$

- podstawiając otrzymujemy:

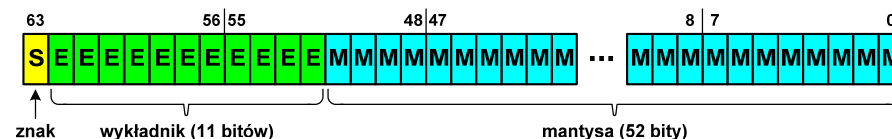
$$S = 0, \quad E = 6_{(10)}, \quad M = 1,5625_{(10)}$$

$$L = (-1)^0 \cdot 1,5625 \cdot 2^6 = 100_{(10)}$$

$$01000010110010000000000000000000_{(IEEE754)} = 100_{(10)}$$

Standard IEEE 754 - liczby 64-bitowe

- Liczba podwójnej precyzji przechowywana jest na 64 bitach:



- Pierwszy bit w zapisie (bit nr 63) jest **bitem znaku** (0 - liczba dodatnia, 1 - liczba ujemna)
- **Wykładnik** zapisywany jest na **11 bitach** (bity nr 62-52) z nadmiarem o wartości 1023
- **Wykładnik** może przyjmować wartości od -1023 (wszystkie bity wyzerowane) do 1024 (wszystkie bity ustawione na 1)
- **Mantysa** zapisywana jest na 52 bitach (pierwszy bit mantysy, zawsze równy 1, nie jest zapamiętywany)

Standard IEEE 754 - zakres liczb

■ Pojedyncza precyzja:

- największa wartość: $\approx 3,4 \cdot 10^{38}$
- najmniejsza wartość: $\approx 1,4 \cdot 10^{-45}$
- zakres liczb: $\langle -3,4 \cdot 10^{38} \dots -1,4 \cdot 10^{-45} \rangle \cup \{0\} \cup \langle 1,4 \cdot 10^{-45} \dots 3,4 \cdot 10^{38} \rangle$

■ Podwójna precyzja:

- największa wartość: $\approx 1,8 \cdot 10^{308}$
- najmniejsza wartość: $\approx 4,9 \cdot 10^{-324}$
- zakres liczb: $\langle -1,8 \cdot 10^{308} \dots -4,9 \cdot 10^{-324} \rangle \cup \{0\} \cup \langle 4,9 \cdot 10^{-324} \dots 1,8 \cdot 10^{308} \rangle$

■ Podwójna rozszerzona precyzja:

- największa wartość: $\approx 1,2 \cdot 10^{4932}$
- najmniejsza wartość: $\approx 3,6 \cdot 10^{-4951}$
- zakres liczb: $\langle -1,2 \cdot 10^{4932} \dots -3,6 \cdot 10^{-4951} \rangle \cup \{0\} \cup \langle 3,6 \cdot 10^{-4951} \dots 1,2 \cdot 10^{4932} \rangle$

Standard IEEE 754 - precyzja liczb

- **Precyzja** - liczba zapamiętywanych cyfr znaczących w systemie (10)
 $4,86452137846 \rightarrow 4,864521$ - 7 cyfr znaczących
- Precyzja liczby zależy od **liczby bitów mantysy**
- Liczba bitów potrzebnych do zakodowania **1** cyfry dziesiętnej:

$$10^1 = 2^n \rightarrow n = \log_2(10) \approx 3,321928$$

- Liczba cyfr dziesiętnych (**d**) możliwa do zakodowania na **m** bitach:

$\log_2(10)$ bitów - 1 cyfra dziesiętna
m bitów - d cyfr dziesiętnych

$$d = \frac{m}{\log_2(10)}$$

Standard IEEE 754 - precyzja liczb

- Dla formatu pojedynczej precyzji:

- mantysa: $23 + 1 = 24$ bity
 - cyfry znaczące: 7
- $$d = \frac{24}{\log_2(10)} = \frac{24}{3,321928} = 7,2247 \approx 7$$

- Dla formatu podwójnej precyzji:

- mantysa: $52 + 1 = 53$ bity
 - cyfry znaczące: 16
- $$d = \frac{53}{\log_2(10)} = \frac{53}{3,321928} = 15,9546 \approx 16$$

- Dla formatu podwójnej rozszerzonej precyzji:

- mantysa: $63 + 1 = 64$ bity
 - cyfry znaczące: 19
- $$d = \frac{64}{\log_2(10)} = \frac{64}{3,321928} = 19,2659 \approx 19$$

Standard IEEE 754 - precyzja liczb

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
float x;
```

```
double y;
```

```
x = 1234567890.0; /* 1.234.567.890 */
```

```
y = 1234567890.0; /* 1.234.567.890 */
```

```
printf("float -> %f\n", x);
```

```
printf("double -> %f\n", y);
```

```
y = 12345678901234567890.0;
```

```
printf("double -> %f\n", y);
```

```
return 0;
```

```
}
```

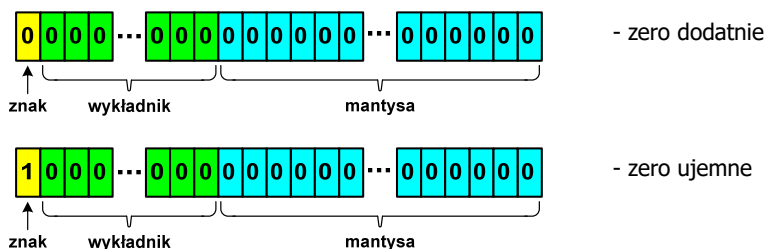
```
float -> 1234567936.000000
```

```
double -> 1234567890.000000
```

```
double -> 12345678901234567000.000000
```

Standard IEEE 754 - wartości specjalne

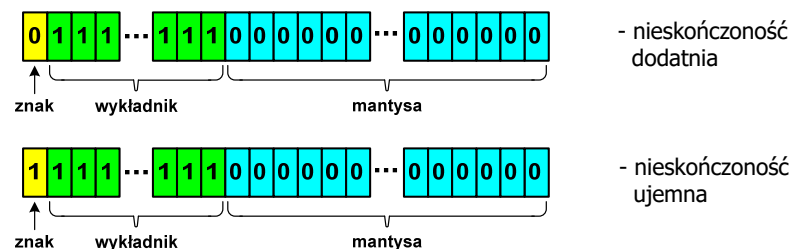
- Zero:



- Podczas porównań zero dodatnie i ujemne są traktowane jako równe sobie

Standard IEEE 754 - wartości specjalne

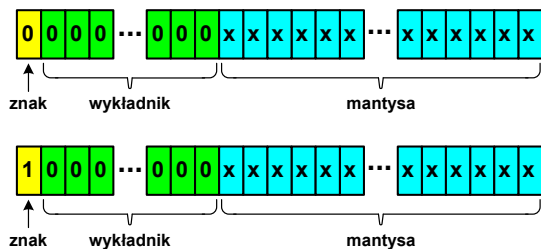
- Nieskończoność:



- Nieskończoność występuje w przypadku wystąpienia nadmiaru (przepełnienia) oraz przy dzieleniu przez zero

Standard IEEE 754 - wartości specjalne

- Liczba zdenormalizowana:



- Pojawia się, gdy występuje **niedomiary** (ang. **underflow**), ale wynik operacji można jeszcze zapisać denormalizując mantysę
- Mantysa nie posiada domyślnej części całkowitej równej 1, tzn. reprezentuje liczbę o postaci **0,xxx...xxx**, a nie **1,xxx...xxx**

Standard IEEE 754 - wartości specjalne

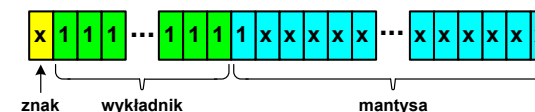
- Standard IEEE 754 definiuje dokładnie wyniki operacji, w których występują specjalne argumenty

Operacja	Wynik
$x / \pm\infty$	0
$\pm\infty \cdot \pm\infty$	$\pm\infty$
$\pm\text{wart_niezer} / 0$	$\pm\infty$
$\infty + \infty$	∞
$\pm 0 / \pm 0$	NaN
$\infty - \infty$	NaN
$\pm\infty / \pm\infty$	NaN
$\pm\infty \cdot 0$	NaN

Standard IEEE 754 - wartości specjalne

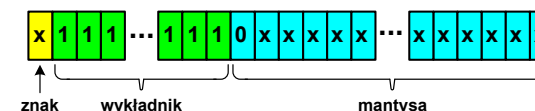
- Nieliczby - NaN (Not A Number)** - nie reprezentują wartości liczbowej
- Powstają w wyniku wykonania niedozwolonej operacji

- QNaN (ang. Quiet NaN)** - ciche nieliczby



- „przechodzą” przez działania arytmetyczne (brak przerwania wykonywania programu)

- SNaN (ang. Signaling NaN)** - sygnalizujące, istotne, głośne nieliczby



- zgłoszenie wyjątku (przerwanie wykonywania programu)

Język C - operacje z wartościami specjalnymi

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    printf("1.0/0.0    = %f\n", 1.0/0.0);
    printf("-1.0/0.0    = %f\n", -1.0/0.0);
    printf("0.0/0.0    = %f\n", 0.0/0.0);
    printf("sqrt(-1.0) = %f\n", sqrt(-1.0));
    printf("1.0/INF    = %f\n", 1.0/(1.0/0.0));
    printf("0*INF      = %f\n", 0.0*(1.0/0.0));

    return 0;
}
```

```
1.0/0.0    = 1.#INF00
-1.0/0.0   = -1.#INF00
0.0/0.0    = -1.#IND00
sqrt(-1.0) = -1.#IND00
1.0/INF    = 0.000000
0*INF      = -1.#IND00
```

- Środowisko: Code::Blocks 20.03

Operacja	Wynik
$x / \pm\infty$	0
$\pm\infty \cdot \pm\infty$	$\pm\infty$
$\pm\text{wart_niezer} / 0$	$\pm\infty$
$\infty + \infty$	∞
$\pm 0 / \pm 0$	NaN
$\infty - \infty$	NaN
$\pm\infty / \pm\infty$	NaN
$\pm\infty \cdot 0$	NaN

Reprezentacja liczb zmiennoprzecinkowych w C

- Typy zmiennoprzecinkowe w języku C:

Nazwa typu	Rozmiar (bajty)	Zakres wartości	Cyfry znaczące
float	4 bajty	$-3,4 \cdot 10^{38} \dots 3,4 \cdot 10^{38}$	7-8
double	8 bajtów	$-1,8 \cdot 10^{308} \dots 1,8 \cdot 10^{308}$	15-16
long double	10 bajtów	$-1,2 \cdot 10^{4932} \dots 1,2 \cdot 10^{4932}$	19-20

- Typ **long double** może mieć także inny rozmiar:

Środowisko	Rozmiar (bajty)
MS Visual C++ 2008 EE	8 bajtów
Borland Turbo C++ Explorer	10 bajtów
Code:Blocks 20.03	16 bajtów
Dev-C++ 5.11	16 bajtów

Reprezentacja liczb zmiennoprzecinkowych w C

```
#include <stdio.h>

int main(void)
{
    float    sf = 0.0f;
    double   sd = 0.0;
    long double slg = 0.0L;

    for (int i=0; i<10000; i++)
    {
        sf = sf + 0.01f;
        sd = sd + 0.01;
        slg = slg + 0.01L;
    }

    printf("float:      %.20f\n", sf);
    printf("double:     %.20f\n", sd);
    printf("long double: %.20Lf\n", slg);

    return 0;
}
```

Reprezentacja liczb zmiennoprzecinkowych w C

- Microsoft Visual C++ 2008 Express Edition (long double - 8 bajtów)

```
float:      100.00295257568359000000
double:    100.000000000001425000000
long double: 100.000000000001425000000
```

- Borland Turbo C++ Explorer (long double - 10 bajtów)

```
float:      100.00295257568359375000
double:    100.000000000001425349000
long double: 100.0000000000001388000
```

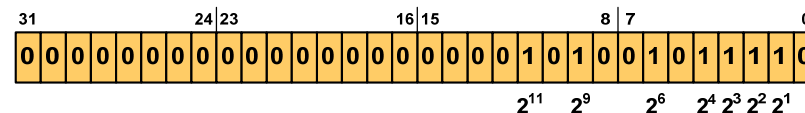
- Code::Blocks 20.03 (long double - 16 bajtów)

```
float:      100.00295257568359000000
double:    100.000000000001425000000
long double: 0.000000000000000000000
```

```
warning: unknown conversion
type character 'L' in format
[-Wformat=]
```

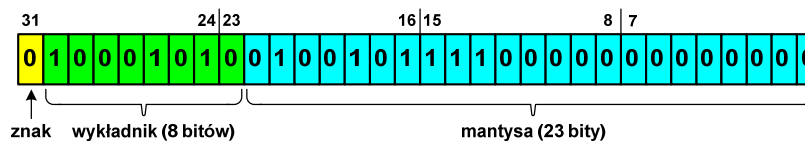
Liczba $2654_{(10)}$ jako całkowita i rzeczywista w C

- int (4 bajty): $2654_{(10)} = 00\ 00\ 0A\ 5E_{(16)}$



$$2^{11} + 2^9 + 2^6 + 2^4 + 2^3 + 2^2 + 2^1 = 2048 + 512 + 64 + 16 + 8 + 4 + 2 = 2654_{(10)}$$

- float (4 bajty): $2654_{(10)} = 45\ 25\ E0\ 00_{(IEEE\ 754)}$



$$+ 138 - 127 = 11_{(10)}$$

$$1.0100101111_{(2)} = 1.2958984_{(10)}$$

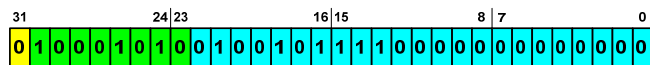
$$1.2958984 \cdot 2^{11} = 2654_{(10)}$$

Język C - nieprawidłowy specyfikator formatu

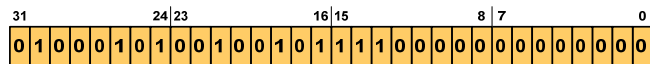
```
int x;  
printf("x (%f) = "); scanf("%f", &x);  
printf("x (%d) = %d\n", x);  
printf("x (%f) = %f\n", x);  
printf("x (%e) = %e\n", x);
```

```
x (%f) = 2654  
x (%d) = 1160110080  
x (%f) = 0.000000  
x (%e) = 5.731705e-315
```

- Zgodnie ze standardem języka C wynik jest **niezdefiniowany**
- Zapamiętana wartość:



- Wyświetlona wartość przy wykorzystaniu %d:



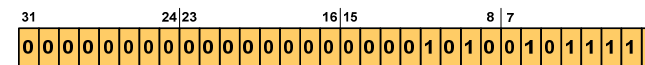
$$2^{30} + 2^{26} + 2^{24} + 2^{21} + 2^{18} + 2^{16} + 2^{15} + 2^{14} + 2^{13} = 1.160.110.080_{(10)}$$

Język C - nieprawidłowy specyfikator formatu

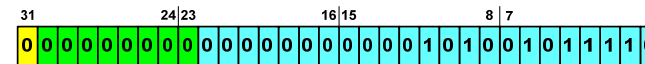
```
float x;  
printf("x (%d) = "); scanf("%d", &x);  
printf("x (%d) = %d\n", x);  
printf("x (%f) = %f\n", x);  
printf("x (%e) = %e\n", x);
```

```
x (%d) = 2654  
x (%d) = 0  
x (%f) = 0.000000  
x (%e) = 3.719046e-042
```

- Zgodnie ze standardem języka C wynik jest **niezdefiniowany**
- Zapamiętana wartość:



- Wyświetlona wartość przy wykorzystaniu %e:



Liczba zdenormalizowana: 3,719046E-42

Koniec wykładu nr 4

Dziękuję za uwagę!