Úvod do programovacího jazyka C

DELTA - Střední škola informatiky a ekonomie, s.r.o.

Luboš Zápotočný

16.10.2025

Bit vs. Byte

Bit vs. Byte

Adresa	0000	0001	0010	0011
Data	137	0**b**10001001	0**x**89	‘a’

- **1 bit** je základní a nejmenší jednotkou informace v počítači
  - Nabývá pouze hodnot 0 či 1
- **1 byte** = 8 bitů
  - Nejmenší adresovatelná jednotka v paměti počítače
  - Nelze tedy od paměti požadovat například 11. bit v pořadí
  - Musíme si nechat nahrát celý byte a z něho poté v vybrat 3. bit

Bit vs. Byte

- **Adresa do paměti**
  - Kladné celé číslo ($\mathbb{N}^+$)
  - Index buňky v paměti
  - Operační systém dává programu **virtuální adresy** místo fyzických

Binární a hexadecimální soustava

Binární soustava

Převod binárního čísla 10001001 do desítkové soustavy

1	0	0	0	1	0	0	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

Binární soustava

$$
\begin{aligned}
10001001 &= 1 \times 128 \\
&\quad +0 \times 64 \\
&\quad +0 \times 32 \\
&\quad +0 \times 16 \\
&\quad +1 \times 8 \\
&\quad +0 \times 4 \\
&\quad +0 \times 2 \\
&\quad +1 \times 1 \\
&\quad = 137
\end{aligned}
$$

Hexadecimální soustava

Hexadecimální kódování číslic

0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9

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

Převod z hexadecimální soustavy

Převod hexadecimálního čísla 5FE9 do desítkové soustavy

5	F (15)	E (14)	9
16^3	16^2	16^1	16^0
4096	256	16	1

Převod z hexadecimální soustavy

$$
\begin{aligned}
\mathrm{0x5FE9} &= 5 \times 4096 \\
&\quad + 15 \times 256 \\
&\quad + 14 \times 16 \\
&\quad + 9 \times 1 \\
&\quad = 24553
\end{aligned}
$$

Převod do binární soustavy

Jak převést číslo 24553 v desítkové soustavě do binární soustavy?

Musíme správně nastavit koeficienty bitů, aby výsledný součet byl roven tomuto číslu

Převod do binární soustavy

?	?	?	?	?	?	?	?
2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}
32768	16384	8192	4096	2048	1024	512	256

?	?	?	?	?	?	?	?
2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
128	64	32	16	8	4	2	1

Převod do binární soustavy

$$
\begin{aligned}
&0 \times 2^{15} + 1 \times 2^{14} + 0 \times 2^{13} + 1 \times 2^{12} \\
&+1 \times 2^{11} + 1 \times 2^{10} + 1 \times 2^{9} + 1 \times 2^{8} \\
&+1 \times 2^{7} + 1 \times 2^{6} + 1 \times 2^{5} + 0 \times 2^{4} \\
&+1 \times 2^{3} + 0 \times 2^{2} + 0 \times 2^{1} + 1 \times 2^{0} \\
&= 24553
\end{aligned}
$$

A tedy binární zápis čísla 24553 je 101111111101001

Hello world

Instalace GCC

Linux: `sudo apt install gcc`

Windows: nainstalujte si Cygwin

Mac: `brew install gcc`

Hello world

```c
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

Co je na kódu zajímavé?

- `printf` na výpis
- `return 0` ??
- `\n` ??

Hello world

Kompilace a spuštění

1. `gcc hello-world.c`
2. `./a.out`
3. `echo $?`

Hello world

Výsledky

1. Výpis řetězce na stdout (terminál) = file descriptor č. 1
2. Nastavená návratová hodnota v shellu (`echo $?`)

Datové typy

Celočíselné datové typy

Jaké znáte **celočíselné** datové typy?

```
char c = 'A';                                // 1 byte
short s = 32767;                             // 2 bytes
int i = 2147483647;                          // 4 bytes
long l = 2147483647L;                        // 4 or 8 bytes
long long ll = 9223372036854775807LL; // 8 bytes
// Unsigned variants
unsigned char uc = 255;
unsigned short us = 65535;
unsigned int ui = 4294967295U;
unsigned long ul = 4294967295UL;
unsigned long long ull = 18446744073709551615ULL;
```

Datové typy s plovoucí desetinnou čárkou

Jaké znáte datové typy s plovoucí **desetinnou čárkou**?

```
float f = 3.14159f;                         // 4 bytes, ~7 decimal digits
double d = 3.141592653589793;               // 8 bytes, ~15 decimal digits
long double ld = 3.14159265358979323 8L; // 16 bytes (typically)
```

Výstup programu na standardní proudy

Výstup programu na standardní proudy

```c
#include <stdio.h>  // `printf`
#include <unistd.h> // `write`
int main() {
    // stdout (file descriptor 1)
    printf("Write to stdout!\n");
    write(1, "Also write to stdout via write!\n", 32);
    // stderr (file descriptor 2)
    fprintf(stderr, "Write to stderr!\n");
    write(2, "Also write to stderr!\n", 22);
    // arbitrary file descriptor
    write(9, "Write to arbitrary file-descriptor!\n", 36);
    return 0;
}
```

Výstup programu na standardní proudy

1. `gcc standard-stream-output.c`
2. `./a.out 1>fd1.txt 2>fd2.txt 9>fd9.txt`

Výstup programu na standardní proudy

fd1.txt (stdout)

```
Write to stdout!
Also write to stdout via write!
```

fd2.txt (stderr)

```
Write to stderr!
Also write to stderr!
```

fd9.txt (file descriptor 9)

```
Write to arbitrary file-descriptor!
```

Výstup programu na standardní proudy

Pozor! Nemíchejte různé metody, které pracují se vstupem či výstupem. Každá metoda používá o kapku jiný princip a jiné buffery.

Výpis na stdout může také vypadat následovně

```
Also write to stdout via write!
Write to stdout!
```

Formátování výstupu

Formátování výstupu - padding

```c
int num = 7;
printf("%d\n", num);    // "7"
printf("%3d\n", num);   // "  7" (padded with spaces)
printf("%03d\n", num);  // "007" (padded with zeros)
```

Formátování výstupu - desetinné místa

```c
double n = 7.123456789;
printf("%.1f\n", n);       // 7.1
printf("%05.2f\n", n);     // 07.12
printf("%.3f\n", n);       // 7.123
printf("%.9f\n", n);       // 7.123456789
printf("%.10f\n", n);      // 7.1234567890
printf("%.15f\n", n);      // 7.123456789000000
```

Pozor! Co se stane, když vypíšeme více desetinných míst?

```c
printf("%.16f\n", n);      // 7.1234567889999996
printf("%.20f\n", n);      // 7.12345678899999956712
```

Formátování výstupu - desetinné místa

Číslo n bylo explicitně nastaveno na hodnotu 7.123456789, přitom je v proměnné hodnota **ostře menší** $<$ **než** 7.123456789

Reprezentace čísel v počítači

Reprezentace čísel v počítači

Celočíselné datové typy

- Reprezentace diskrétních jevů
- „Přesné“ výpočty
  - V rámci rozsahu daného datového typu

Reálné datové typy

- Reprezentace spojitých jevů
- Výpočty jsou **nepřesné** - obsahují zaokrouhlovací chybu

Vyjádření čísla v různých soustavách

Vyjádření čísla x v soustavě z

$$(x)_z = \pm \sum_{i=0}^{n} b_i z^i, \quad b_i \in \langle 0, z-1 \rangle$$

Poznámka: Suma $\sum$ je v podstatě cyklus `for` s následující logikou:

```
int sum = 0;
for (int i = 0; i <= n; i++) {
  sum += b[i] * z[i];
}
```

Vyjádření čísla v různých soustavách

Decimální (z = 10): $(x)_{10} = 200$

$$200_{10} = 2 \times 10^2 + 0 \times 10^1 + 0 \times 10^0$$

Binární (z = 2): $(x)_2 = 11001000$

$$\begin{aligned} 11001000_2 &= 1 \times 2^7 + 1 \times 2^6 \\ &+0 \times 2^5 + 0 \times 2^4 \\ &+1 \times 2^3 + 0 \times 2^2 \\ &+0 \times 2^1 + 0 \times 2^0 = 200_{10} \end{aligned}$$

Vyjádření čísla v různých soustavách

Hexadecimální (z = 16): $(x)_{16} = \mathrm{C8}$

$$\mathrm{C8}_{16} = 12 \times 16^1 + 8 \times 16^0 = 200_{10}$$

Celočíselné datové typy

Reprezentaci **celočíselných** datových typů lze rozdělit dle:

Přesnosti

- short - nižší přesnost
- long - vyšší přesnost

Znaménka (sign)

- unsigned - bez znaménka - $\mathbb{Z}_{\geq 0}$
- signed - se znaménkem - $\mathbb{Z}$

Rozsahy celočíselných datových typů

Typ	Paměť	Rozsah	Znaménko	Formát
short [int]	2 byte	$\langle -2^{15}; 2^{15} - 1 \rangle$	ano	`%hd`
unsigned short [int]	2 byte	$\langle 0; 2^{16} - 1 \rangle$	ne	`%hu`
int	4 byte	$\langle -2^{31}; 2^{31} - 1 \rangle$	ano	`%d`
unsigned int	4 byte	$\langle 0; 2^{32} - 1 \rangle$	ne	`%u`
long [int]	≥ 4 byte	$\langle -2^{31}; 2^{31} - 1 \rangle$	ano	`%ld`

unsigned long [int]	≥ 4 byte	$\langle 0; 2^{32} - 1 \rangle$	ne	`%lu`
long long [int]	≥ 8 byte	$\langle -2^{63}; 2^{63} - 1 \rangle$	ano	`%lld`
unsigned long long [int]	≥ 8 byte	$\langle 0; 2^{64} - 1 \rangle$	ne	`%llu`

Poznámka

Některé datové typy mohou mít na 64-bitové architektuře větší velikost než na 32-bitové architektuře

Rozsahy celočíselných datových typů

Otázka

Co se stane při výpisu unsigned int pomocí formátovacího řetězce %d?

```
unsigned int a = (unsigned int)INT_MAX * 2 + 1;
printf("%d\n", a);
```

Odpověď

Výsledek bude interpretován jako signed int, což může vést k zobrazení nesprávného čísla, pokud je MSB (Most Significant Bit) nastaven na 1

- V tomto případě je MSB nastaven na 1 a kód zobrazí hodnotu −1

Kódování znamének

Přímý kód

Číslo je v počítači uloženo v binárním tvaru a problematika znamének je řešena pomocí **kódování**

První (naivní) přístup jak znaménka ukládat je použít **znaménkový bit**

bit na první pozici (MSB) je vyhrazen **pro znaménko**, zbytek je číslo

- MSB = Most Significant Bit
- **0** = + (*kladné číslo*)
- **1** = — (*záporné číslo*)

Přímý kód

$$P(5)_{10} = 0101$$

$$P(-5)_{10} = 1101$$

Přímý kód

Problém dvojí reprezentace nuly:

$$P(0)_{10} = 0000$$
$$P(-0)_{10} = 1000$$

Přímý kód

Otázka

Jakého rozsahu nabývá 8-bitové číslo reprezentované přímým kódem?

$$n = \#\text{bitů}, \quad m = n - 1, \quad x \in \langle -(2^m - 1); 2^m - 1 \rangle$$

Pro $n = 8$ je $m = 7$, tedy rozsah: $\langle -127; 127 \rangle$

Pro $n = 4$ je $m = 3$, tedy rozsah: $\langle -7; 7 \rangle$

Přímý kód

Aritmetické operace s přímým kódem jsou složité a nepoužívají se

Pouze pro představu:

- Pokud mají čísla **stejná znaménka** sečteme absolutní hodnoty těchto čísel a znaménko ponecháme stejné
- Pokud mají čísla **různá znaménka**, musíme od, v absolutní hodnotě, většího čísla odečíst menší a výsledné znaménko získáme ze znaménka většího čísla

Jednodušší implementace aritmetických operací nabízejí jiná kódování

Inverzní kód

Inverzní kód je alernativní přístup k reprezentaci čísel se znaménkem

Záporné číslo je **negací** (jedničkovým doplňkem) kladného čísla

Příklad

$$I(-100)_{10} = !(01100100)_2 = (10011011)_2$$

Inverzní kód

Výhody

1. Jednoduchá změna znaménka (stačí invertovat všechny bity)
2. Jednodušší hardwarová implementace sčítání

Inverzní kód - výpočet 5 + 2

$$I(5)_{10} = 0101$$

$$I(2)_{10} = 0010$$

Princip výpočtu je téměř identický s **postupem ze základní školy**

$$\begin{array}{rcccc} & 0 & 1 & 0 & 1 \\ + & 0 & 0 & 1 & 0 \\ \hline & 0 & 1 & 1 & 1 \end{array}$$

Výsledek je tedy $0111 = I(7)_{10}$

Inverzní kód - výpočet 5 - 3

$$I(5)_{10} = 0101$$

$$I(-3)_{10} = !I(3)_{10} = !0011 = 1100$$

$$\begin{array}{rcccc} & 0 & 1 & 0 & 1 \\ + & 1 & 1 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 & 1 \end{array}$$

Inverzní kód - výpočet 5 - 3

Vzniklo nám takzvané **end-around carry**, které musíme **ještě jednou přičíst** k výsledku

$$\begin{array}{rcccc} & 0 & 0 & 0 & 1 \\ + & & & & 1 \\ \hline & 0 & 0 & 1 & 0 \end{array}$$

Výsledek je tedy $0010 = I(2)_{10} = 5 - 3$

Inverzní kód - výpočet 5 - 3

Problém dvojí reprezentace nuly

$$\begin{aligned} I(0)_{10} &= 00000000 \\ I(-0)_{10} &= !00000000 \\ &= 11111111 \end{aligned}$$

Problém výpočetní složitosti (dva součty namísto jednoho)

Inverzní kód - výpočet 5 - 3

Inverzní kód je někdy označován jako **jedničkový doplněk** (**one's complement**)

Doplňkový kód

Doplňkový kód

Přičtením **1** k jedničkovému doplňku (inverznímu kódu) získáváme **doplňkový kód** (**dvojkový doplněk**, **two's complement**)

Příklad:

$$\begin{aligned} D(-100)_{10} &= !(01100100)_2 + 1 \\ &= (10011011)_2 + 1 \\ &= (10011100)_2 \end{aligned}$$

Doplňkový kód

Výhody

1. Jediná reprezentace nuly

$$
\begin{aligned}
D(0)_{10} &= 00000000 \\
D(-0)_{10} &= !(00000000) + 1 \\
&= 11111111 + 1 \\
&= 00000000
\end{aligned}
$$

Doplňkový kód

2. Sčítání a odčítání funguje stejně pro kladná i záporná čísla
3. Nevzniká problém s end-around carry

Doplňkový kód

Problém nesymetrického intervalu hodnot

Obecný zápis

$$x \in \langle -2^{n-1}; 2^{n-1} - 1 \rangle$$

- pro 8 bitů: $\langle -128; 127 \rangle$
- pro 4 bity: $\langle -8; 7 \rangle$

Nelze vyjádřit $|\min|$ (absoltní hodnota nejzápornějšího čísla)

Doplňkový kód

Doplňkový kód se dnes **standardně používá** pro celá čísla se znaménkem

Doplňkový kód - výpočet 5 + 2

$$D(5)_{10} = (0101)_2$$
$$D(2)_{10} = (0010)_2$$

$$\begin{array}{rcccc} & 0 & 1 & 0 & 1 \\ + & 0 & 0 & 1 & 0 \\ \hline & 0 & 1 & 1 & 1 \end{array}$$

Výsledek je tedy $0111 = D(7)_{10}$

Doplňkový kód - výpočet 5 - 3

$$D(5)_{10} = (0101)_2$$

$$D(-3)_{10} = !(0011)_2 + 1 = (1100)_2 + 1 = (1101)_2$$

$$\begin{array}{rcccc} & 0 & 1 & 0 & 1 \\ + & 1 & 1 & 0 & 1 \\ \hline 1 & 0 & 0 & 1 & 0 \end{array}$$

Carry bit můžeme bezpečně **ignorovat**

Výsledek je tedy $0010 = D(2)_{10} = 5 - 3$

Opakování mocnění

$$2^2 = 2 \times 2 = 4$$

$$2^4 = 2 \times 2 \times 2 \times 2 = 16$$

$$2^{-1} = \frac{1}{2} = 0.5$$

$$2^{-2} = \frac{1}{2^2} = \frac{1}{2} \times \frac{1}{2} = 0.25$$

$$2^{-3} = \frac{1}{2^3} = \frac{1}{2} \times \frac{1}{2} \times \frac{1}{2} = 0.125$$

Pevná řádová čárka

Desetinné číslo je reprezentováno **pevnou řádovou čárkou** následovně

- **1** bit pro znaménko (sign)
- **n** bitů pro celou část
- **m** bitů pro desetinnou část

Obecný zápis

$$(x)_2 = (x_c + x_d), \quad x_c = \sum_{i=0}^{n} b_i 2^i, \quad x_d = \sum_{i=-m}^{-1} b_i 2^i$$

Pevná řádová čárka

Jak reprezentovat desetinné číslo 2.25 v binární soustavě?

Nejdříve najdeme binární reprezentaci celé části čísla $2_{10} = 10_2$

Poté nalezneme binární reprezentaci desetinné části čísla

$$\begin{aligned} 0.25_{10} &= 0 \times \frac{1}{2} + 1 \times \frac{1}{4} \\ &= 0 \times 2^{-1} + 1 \times 2^{-2} = 0.01_2 \end{aligned}$$

Výsledný binární zápis je tedy 10.01_2

Plovoucí řádová čárka

Semilogaritmický tvar čísla (vědecký zápis)

$$x = \mathrm{m} \times \mathrm{z}^{\mathrm{e}}$$

- m je mantisa
- z je základ soustavy
- e je exponent

Příklad z desítkové soustavy

$$1234567890 = 12345.67890 \times 10^{5} = 1.234567890 \times 10^{9}$$

Plovoucí řádová čárka

Normalizační podmínka

1. $1 \leq m < z$

- Hodnota mantisy je vždy v intervalu $\langle 1, z)$

2. Binární zápis daného čísla začíná vždy číslicí **1**

- Nepíšeme zbytečné nuly před první číslicí **1**

Pokud číslo splňuje tuto normalizační podmínku, nazýváme ho **normalizované**

Příklady normalizovaných čísel

$$9.125_{10} = 1001.001_2 = 1.001001_2 \times 2^3$$

$$0.625_{10} = 0.101_2 = 1.01_2 \times 2^{-1}$$

$$13.625_{10} = 1101.101_2 = 1.101101_2 \times 2^3$$

$$15.03125_{10} = 1111.00001_2 = 1.11100001_2 \times 2^3$$

Binární zápis reálného čísla

$\pm$	exponent (n bitů)	mantisa (m bitů)
	2^{n-1} ... 2^2 2^1 2^0	2^{-1} 2^{-2} 2^{-3} 2^{-4} ... 2^{-m}

Přesnost binárně uloženého reálného čísla

- Čím více bitů má mantisa, tím vyšší přesnost čísla
- Na uložení exponentu typicky stačí menší počet bitů

float - 32 bitů ($n = 8, m = 23$)

double - 64 bitů ($n = 11, m = 52$)

Semilogaritmický tvar a plovoucí řádová čárka

Dekadické číslo

$$-123,000,000,000,000 = -1.23 \times 10^{14}$$

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

Binární číslo

$$110\ 1100\ 0000\ 0000 = 1.1011 \times 2^{14}$$

Princip posouvání řádové čárky dal název této reprezentaci
plovoucí řádová čárka - **floating point**

Přesnost desetinných čísel v počítači

```c
float a = 0.1;
float b = 0.2;
float c = a + b;

printf("%.6f\n", c);
```

Odpověď

```
0.300000
```

Ale pozor! Interně není přesně 0.3, jak uvidíme dále…

Číslo 0.1 v binární soustavě

$$0.1_{10} = 0.00011\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011$$
$$0011\ 0011\ 0011\ 0011\ 0011\ \ldots$$

Nekonečná a opakující se sekvence bitů

Nepřesné zobrazení reálných čísel

$$\frac{1}{3} \approx 0.0101\ 0101\ 0101\ \ldots\ 01_2$$

$$\frac{1}{5} \approx 0.0011\ 0011\ 0011\ \ldots\ 0011_2$$

$$\frac{1}{10} \approx 0.00011\ 0011\ 0011\ \ldots\ 0011_2$$

Přesně lze vyjádřit pouze čísla ve tvaru $x \times 2^{-k}$,
kde x je celé číslo a k je nezáporné celé číslo

Všechna ostatní čísla se ukládají **nepřesně** (zaokrouhleně)

Přesně uložitelná čísla s desetinnou čárkou

Přesně lze uložit pouze čísla ve tvaru $x \times 2^{-k}$

$$0.5 = 1 \times 2^{-1}$$
$$6.5 = 13 \times 2^{-1}$$
$$10.0 = 10 \times 2^{0}$$
$$0.875 = 7 \times 2^{-3}$$
$$1.75 = 7 \times 2^{-2}$$
$$12.625 = 101 \times 2^{-3}$$

Norma IEEE754

Norma IEEE754 definuje, jak se mají reálná čísla ukládat do paměti

- Jednoduchá přesnost (32 bitů) - v jazyce C **float**
- Dvojnásobná přesnost (64 bitů) - v jazyce C **double**

Poznámka

Norma se rozšiřuje v rámci IEEE 754-2008 a dále v rámci IEEE 754-2019

- 16 bitová přesnost (využíváno při grafice)
- 128 a 256 bitová přesnost (vědecké výpočty)
- ...

Aritmetika čísel s desetinou čárkou

```c
float a = 0.1;
float b = 0.2;
float c = a + b;

if (c == 0.3) {
    printf("Rovná se\n");
} else {
    printf("Nerovná se\n");
}
```

Odpověď

```
Nerovná se
```

Aritmetika čísel s desetinou čárkou - epsilon

```c
#include <math.h>
#define EPSILON 0.000001

float a = 0.1;
float b = 0.2;
float c = a + b;

if (fabs(c - 0.3) < EPSILON) {
    printf("Rovná se (s tolerancí)\n");
} else {
    printf("Nerovná se\n");
}
```

Čtení a kontrola validity vstupu

Čtení standardního vstupu

```c
#include <stdio.h>
int main() {
    int age;
    float height;

    printf("Enter your age: ");
    scanf("%d", &age);
    printf("Enter your height: ");
    scanf("%f", &height);
    printf("Loaded age: %d and height: %f\n", age, height);

    return 0;
}
```

Čtení standardního vstupu

```c
int a, b;
char ch;
char str[100];

// Read multiple values
scanf("%d %d", &a, &b);
// Skip whitespaces and read one character
scanf(" %c", &ch);

// Read string (stops at whitespaces)
scanf("%s", str);
// Read entire line (stops at newline)
scanf("%[^\n]", str);
```

Kontrola standardního vstupu

Funkce **scanf** z knihovny `stdio.h`

```
int scanf(const char *restrict format, ...);
```

$$\text{return value} = \begin{cases} -1 & \text{if end of file was reached} \\ n & \text{if } n \text{ values had been read correctly} \end{cases}$$

Podobné formátovací řetězce jako pro formátování výstupu

Nutné předávat reference (pointery) pro “naplnění” daných proměnných

Kontrola standardního vstupu

```c
int num;
int result;
printf("Enter a number: ");
result = scanf("%d", &num);
if (result == 1) {
    printf("Successfully read: %d\n", num);
} else if (result == 0) {
    printf("Invalid input - not a number\n");
    // Clear invalid line
    while (getchar() != '\n');
} else if (result == EOF) { // global variable defined in stdio.h
    printf("End of file reached\n");
}
```

Modulární aritmetika

Modulární aritmetika

Modulo (%) je binární operátor, který dává zbytek po celočíselném dělení

Příklady

$$5 \% 3 = 2$$
$$9 \% 4 = 1$$
$$1024 \% 2 = 0$$

Logické a bitové operace

Posun bitů doleva

Posun doleva

Posun bitů o jednu pozici **vlevo** je stejná operace jako **násobení 2**

```
int a = 5;       // ... 0000 0101
int b = a << 1; // ... 0000 1010 = 10
int c = a << 2; // ... 0001 0100 = 20
int d = a << 3; // ... 0010 1000 = 40
printf("a = %d\n", a);         // 5
printf("a << 1 = %d\n", b);  // 10
printf("a << 2 = %d\n", c);  // 20
printf("a << 3 = %d\n", d);  // 40
```

Posun bitů doprava

Posun doprava

Posun bitů o jednu pozici **vpravo** je stejná operace jako **dělení 2 následované zaokrouhlením dolů**

```
int a = 20;      // ... 0001 0100
int b = a >> 1; // ... 0000 1010 = 10
int c = a >> 2; // ... 0000 0101 = 5
int d = a >> 3; // ... 0000 0010 = 2
printf("a = %d\n", a);         // 20
printf("a >> 1 = %d\n", b);  // 10
printf("a >> 2 = %d\n", c);  // 5
printf("a >> 3 = %d\n", d);  // 2
```

Další bitové operace

Bitové operace

```
int a = 12;       // ... 1100
int b = 10;       // ... 1010

// Bitwise AND
int c = a & b;    // ... 1000 = 8
// Bitwise OR
int d = a | b;    // ... 1110 = 14
// Bitwise XOR
int e = a ^ b;    // ... 0110 = 6
// Bitwise NOT (complement)
int f = ~a;       // ... 0011 = -13
```

Příklady na bitové operace

1. Nastavte 3. nejnižší bit na 1 v čísle 10.

```
int a = 10;             // ... 0000 1010
int b = a | (1 << 2); // ... 0000 1110 = 14
```

2. Nastavte první 4 bity na 0 v čísle 19.

```
int a = 19;             // ... 0001 0011
int b = a & ~0xF;       // ... 0001 0000 = 16
```

3. Nastavte 4. nejnižší bit na jeho negaci nezávisle na zvoleném čísle.

```
int a = 12;             // ... 0000 1100
int b = a ^ (1 << 3); // ... 0000 0100 = 4
```