Ukazatele

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

Ing. Luboš Zápotočný

04.12.2025

Pointer

Co je to pointer?

Pointer (ukazatel) je proměnná, která obsahuje **adresu do paměti**

Analogie:

- Běžná proměnná = dům (obsahuje hodnotu)
- Pointer = adresa domu (říká, kde hodnota je)

Proč potřebujeme pointery?

- Dynamická alokace paměti
- Efektivní předávání dat do funkcí
- Práce s poli, strukturami a datovými strukturami (seznamy, stromy, …)

Pointer

```
int x = 42;         // Normal variable
int *ptr = &x;      // Pointer to x
```

Vizualizace v paměti:

Typ	Proměnná	Adresa	Hodnota
`int`	`x`	0x1000	42
`int*`	`ptr`	0x2000	0x1000

Pointer

- `&` je operátor „address-of“ - vrací adresu proměnné
- `<typ>*` obsahuje adresu do paměti, ve které je hodnota typu `<typ>`

Dereference je operace, která vrací hodnotu na adrese, na kterou pointer ukazuje

```
int value = *ptr; // * = dereference - read value from address
```

Pointer

Pozor na syntaxi:

```
int* p1, p2; // p1 is pointer, p2 is int!
int *p1, *p2; // both are pointers
```

Pointer

Operátor * má tedy dva významy:

1. **V deklaraci** - říká, že daná proměnná je pointer

```
int *ptr;  // ptr is pointer to int
```

2. **V použití** - přístup k hodnotě (dereference)

```
int x = *ptr;  // read value from address
*ptr = 20;     // write value to address
```

Pointer

```c
int a = 5;
int *p = &a;
printf("%d", p);
```

```
-634536716
```

Vypsali jsme hodnotu pointeru, né hodnotu, na kterou pointer ukazuje!

```c
printf("%d", *p);
```

```
5
```

Pointer

```c
int a = 5;
int *p = &a;
a = 10;
printf("%d\n", a);
printf("%d\n", *p);
```

```
10
10
```

- p je pointer na a
- *p je dereference pointeru p, což je hodnota proměnné a
- Změna a je tedy viditelná přes *p

Pointer

```c
int a = 5;
int *p = &a;
*p = 20;
printf("%d\n", a);
printf("%d\n", *p);
```

```
20
20
```

- `*p = 20` je dereference pointeru `p` a změna hodnoty na dané paměťové adrese

Pointer na pointer

Pointer může ukazovat i na jiný pointer:

```
int a = 1;
int *pa = &a;         // Pointer to int
int **ppa = &pa;      // Pointer to pointer to int
int ***pppa = &ppa; // Pointer to pointer to pointer to int
```

Pointer na pointer

Typ	Proměnná	Adresa	Hodnota
`int`	`a`	0x1000	1
`int*`	`pa`	0x2000	0x1000
`int**`	`ppa`	0x3000	0x2000
`int***`	`pppa`	0x4000	0x3000

Dereference pointeru na pointer

```c
int a = 1;
int *pa = &a;
int **ppa = &pa;
int ***pppa = &ppa;
printf("%d, %d, %d, %d\n", a, *pa, **ppa, ***pppa);
```

```
1, 1, 1, 1
```

- a - přímo 1
- *pa - 1x dereference - 1
- **ppa - 2x dereference - 1
- ***pppa - 3x dereference - 1

Pointer aritmetika

```c
int arr[] = {10, 20, 30, 40, 50}; // static array of 5 ints
int *ptr = arr;  // pointer to the first element of the array

// print pointer address in hexadecimal
printf("%p\n", ptr);
```

```
0x7ffc41093950
```

Pokud je pointer adresa (v podstatě číslo - index) do paměti, tak co bude výsledkem `ptr + 1`?

```c
printf("%p\n", ptr + 1);
```

```
0x7ffc41093954
```

Pointer aritmetika

„Počítání“ s pointery je velmi důležitý princip, který umožňuje přistupovat k prvkům pole pomocí indexu tak, jak to znáte

Hodnoty typu `int` jsou totiž uloženy pomocí 4 bytů a pokud tedy budeme chtít další int, tak musíme přeskočit 4 byty

Pointer aritmetika

Adresa	Byte 1	Byte 2	Byte 3	Byte 4	Hodnota
0x7ffc41093950	0x00	0x00	0x00	0x0A	10
0x7ffc41093954	0x00	0x00	0x00	0x14	20
0x7ffc41093958	0x00	0x00	0x00	0x1E	30
0x7ffc4109395c	0x00	0x00	0x00	0x28	40
0x7ffc41093960	0x00	0x00	0x00	0x32	50

Endianita

Poznámka:

Přesné uložení bitů záleží na architektuře procesoru, konkrétně na tom, jakou endianitu používáme

Endianita je způsob, jakým jsou bity uloženy v paměti a určuje, zdali je MSB (Most Significant Bit) na nižší nebo vyšší adresě

- Little-endian: MSB na vyšší adresě
- Big-endian: MSB na nižší adresě

Endianita

Jakou endianitu dnes v počítači používáme?

Big-endian (např. Motorola 68000, IBM z/Architecture):

- 10 = 0x00 0x00 0x00 0x0A

Little-endian (např. x86, ARM):

- 10 = 0x0A 0x00 0x00 0x00

Pointer aritmetika

```
printf("%d\n", *ptr);
```

```
10
```

```
printf("%d\n", *(ptr+1));
```

```
20
```

```
printf("%d\n", *(ptr+2));
```

```
30
```

Proč?

Pointer aritmetika

Důležité: `ptr+1` nezvyšuje adresu o 1 byte, ale o **velikost datového typu**!

- Pro `int` (4 bytes): `ptr+1` zvýší adresu o 4 bytes
- Pro `char` (1 byte): `ptr+1` zvýší adresu o 1 byte

Ekvivalentní způsoby přístupu k prvkům pole

1. Pomocí pointer aritmetiky:

```
for (int i = 0; i < 5; ++i) {
    printf("%d", *(arr + i));
}
```

2. Pomocí indexu (klasicky):

```
for (int i = 0; i < 5; ++i) {
    printf("%d", arr[i]);
}
```

Fakt: `str[i]` je jen syntaktický cukr pro `*(str + i)` ✨

Pointer aritmetika se strukturami

```c
typedef struct {
    long weight; unsigned char countOfDoors;
} Car;

int main() {
    Car arr[] = {
        {.weight = 1000, .countOfDoors = 5},
        {.weight = 2000, .countOfDoors = 3},
    };
    for (int i = 0; i < 2; ++i) {
        printf("%ld %hhd\n", arr[i].weight, (arr + i)->countOfDoors);
    }
}
```

Pointer aritmetika se strukturami

Co v tomto případě znamená `(arr + 1)`?

```
printf("%p\n", arr);
printf("%p\n", arr + 1);
```

```
0x7ffcf6bf2370
0x7ffcf6bf2380
```

Jedná se tedy o skok o 0x10 bytes = 16 bytů, což je velikost struktury `Car` (8 bytes + 2 bytes a padding)

```
printf("%zu", sizeof(Car));
```

```
16
```