Informatika 3

2

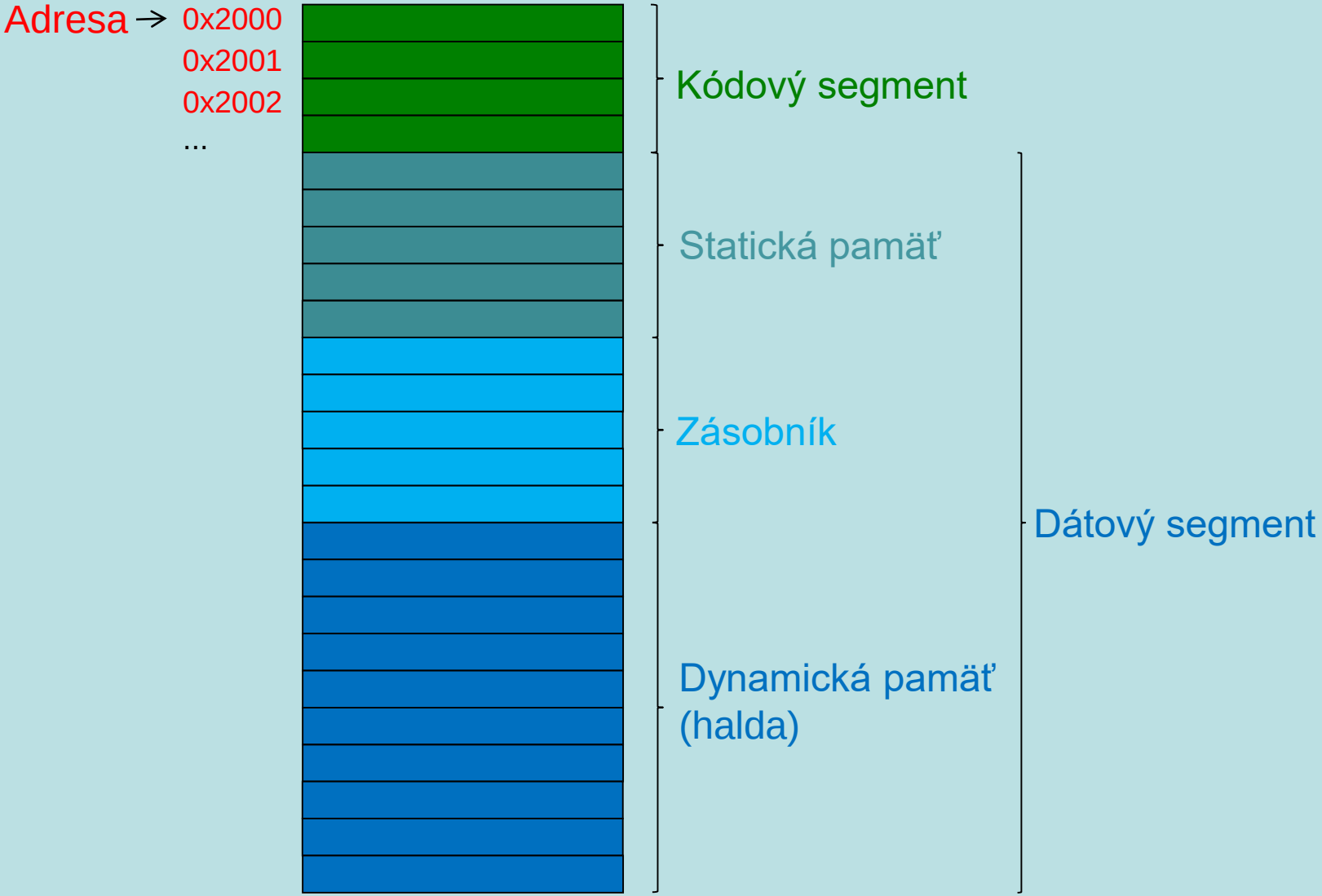
Smerníky

KST

Programátor - praktik

- Myslí za hranice bezprostredného problému
- Posudzuje problém širšom kontexte v rámci celkového obrazu
- Robí rozumné kompromisy a kvalifikované rozhodnutia

Pamäť


Adresa →
0x2000
0x2001
0x2002
...
Kódový segment
Statická pamäť
Zásobník
Dátový segment
Dynamická pamäť
(halda)

Smerníky – čo je to?

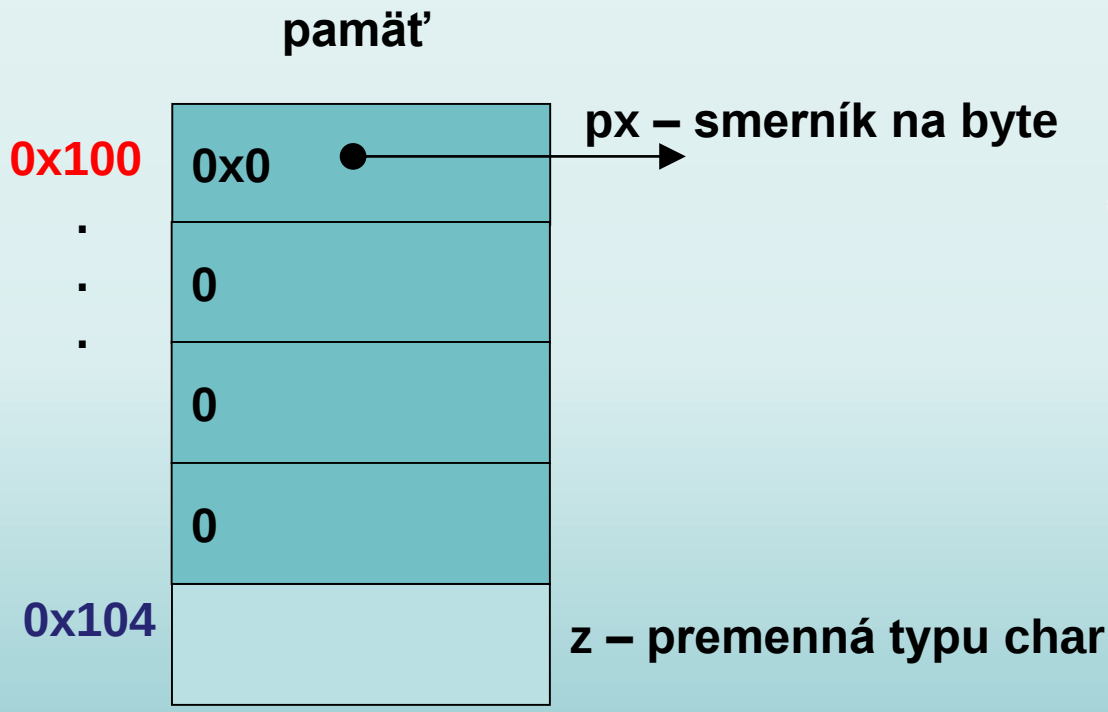
- smerník je premenná, ktorá obsahuje adresu inej premennej

- pomocou smerníka môžeme ku premenným pristupovať tzv. nepriamo

Smerníky - definícia

char *px;

sizeof(px) … 4

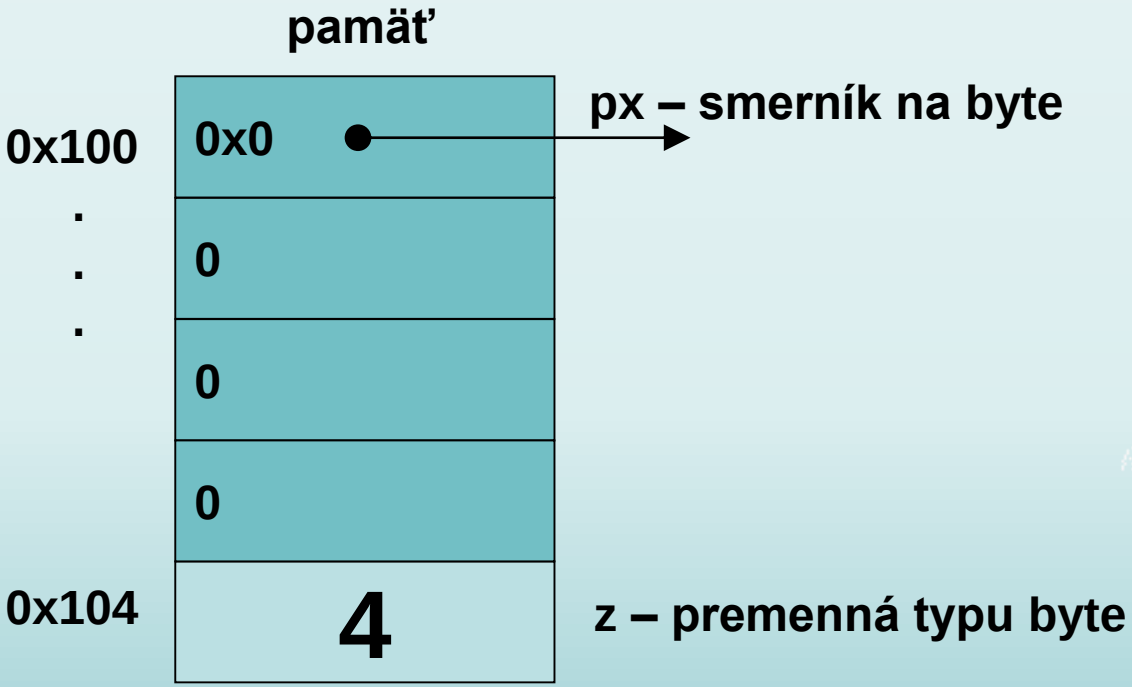


char z;

sizeof(z) … 1

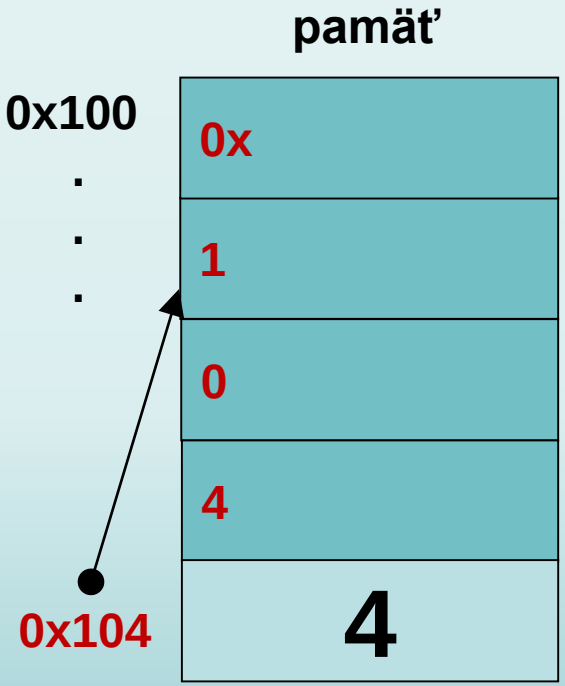
Smerníky – práca so smerníkom

z = 4;



Smerníky – priradenie adresy

```
px = &z;
```

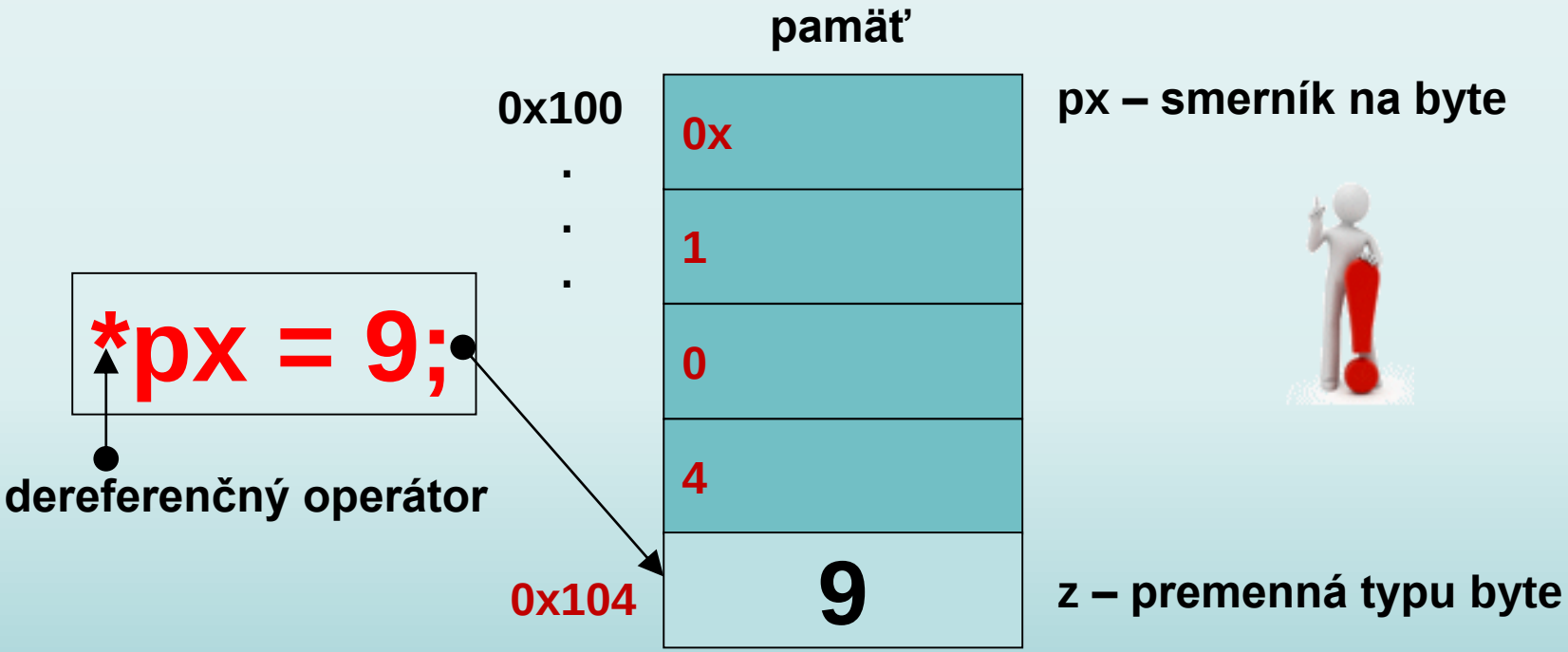


px – smerník na byte

z – premenná typu byte

adresu premennej **z** získame výrazom **&z**

Smerníky – priradenie hodnoty



obsah premennej na ktorú ukazuje smerník **px**
získame/nastavíme výrazom ***px**

Smerníky - typ

- Smerník môže ukazovať len na jeden konkrétny typ

Smerníky a polia

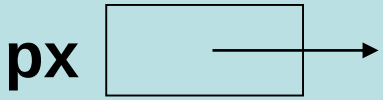
- V jazyku C++ je medzi smerníkmi a poľami veľmi silný vzťah.
- Definícia:

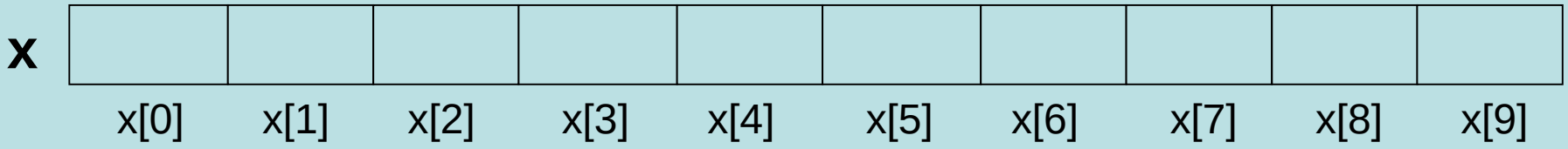
```
int x[10], *px, y;  // definuje  pole x s veľkosťou 10 (x[0],
                    // x[1], x[2],  ... x[9]), smerník  px na
                    // celé  číslo  a  premennú  y  typu int

px = &x[0];         // nastaví px tak, aby  ukazoval na nultý prvok
                    // poľa x, t.j. px  bude obsahovať
                    // adresu prvku x[0].

y = *px;            //skopíruje obsah x[0] do premennej y  */
```

- Ak px ukazuje na daný prvok poľa, px+1 ukazuje na nasledujúci prvok a px-1 ukazuje na predchádzajúci prvok poľa

Práca s poľom cez smerník

```
int x[10], *px, i;
```





Práca s poľom cez smerník

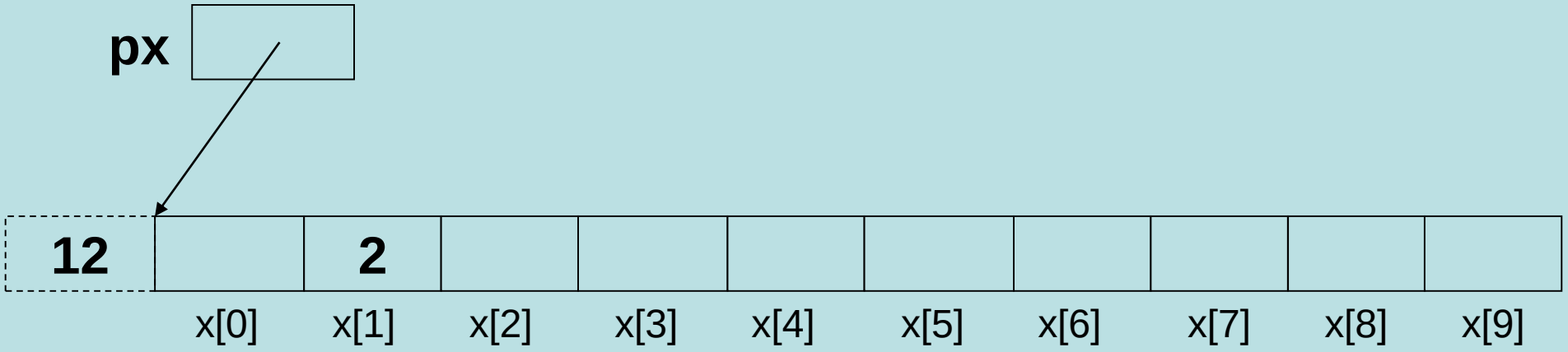
```
int x[10], *px, i;
px = &x[0];
```

Práca s poľom cez smerník

```
int x[10], *px, i;
px = &x[0];
*(px+1) = 2;
```

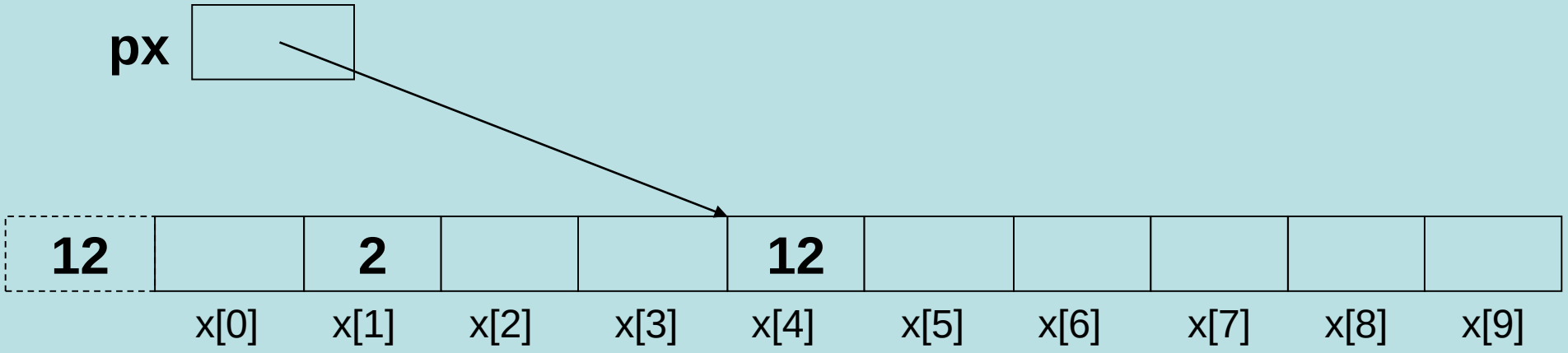
Práca s poľom cez smerník

```
int x[10], *px, i;
px = &x[0];
*(px+1) = 2;
*(px-1) = 12;        // POZOR!!!
```



Práca s poľom cez smerník

```
int x[10], *px, i;
px = &x[0];
*(px+1) = 2;
*(px-1) = 12;        // POZOR!!!
px+=4; *px = 12;
```



Smerník kontra pole

- V skutočnosti aj kompilátor prekladá odkaz na pole ako smerník na začiatok poľa
- Meno poľa je teda smerník na začiatok poľa.

```
px = &x[0];   <=>   px = x;
px + i        <=>   x + i
*(px + i)     <=>   *(x + i)
*(x + i)      <=>   x[i]
*(px + i)     <=>   px[i]
```

Smerník kontra pole

- Pozor na rozdiel medzi menom poľa a smerníkom - smerník je premenná, ale meno poľa je konštanta

```
int x[10], *px, y;
px++;          //  SPRÁVNE
px = &y;       //  SPRÁVNE

x++;           // NESPRÁVNE
x = &y;        // NESPRÁVNE
```

Prečo ?

Smerníková aritmetika: NULL

- Do premennej typu smerník sa nedá priamo priradzovať hodnota okrem hodnoty NULL tzv. univerzálna nula

```
int *px=NULL;
```

- Akýkoľvek smerník môžeme zmysluplne porovnávať na rovnosť alebo nerovnosť s NULL

Smerníková aritmetika - relačné operátory

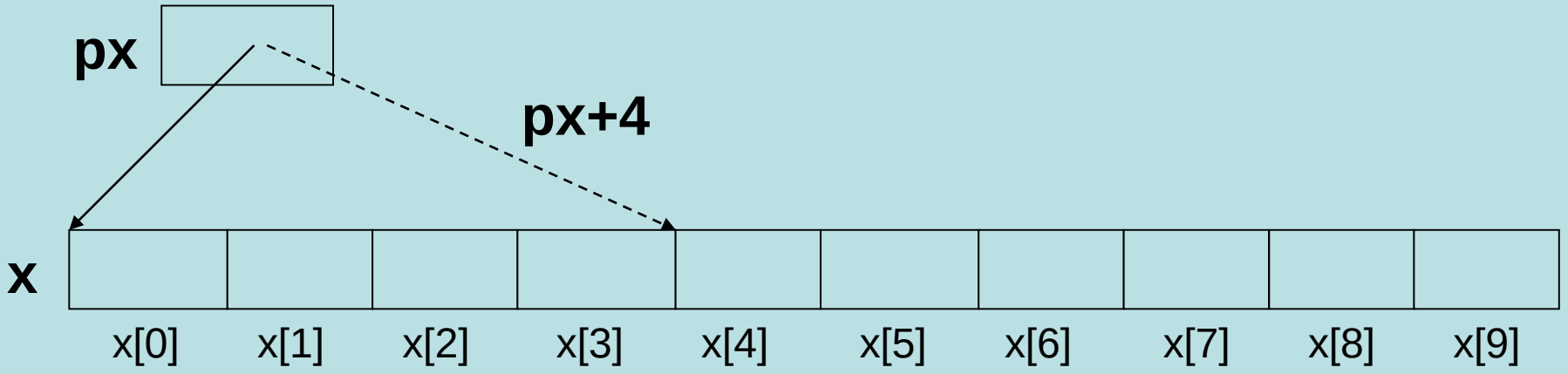
- Smerníky možno za určitých okolností porovnávať
  - Ak p a q ukazujú na prvky toho istého poľa, relácie ako <, <=, >, >=, ==, != fungujú správne. Výraz

    p < q

  - je pravdivý, ak p ukazuje na nižší prvok poľa ako q

Smerníková arititmetika - + - celé číslo

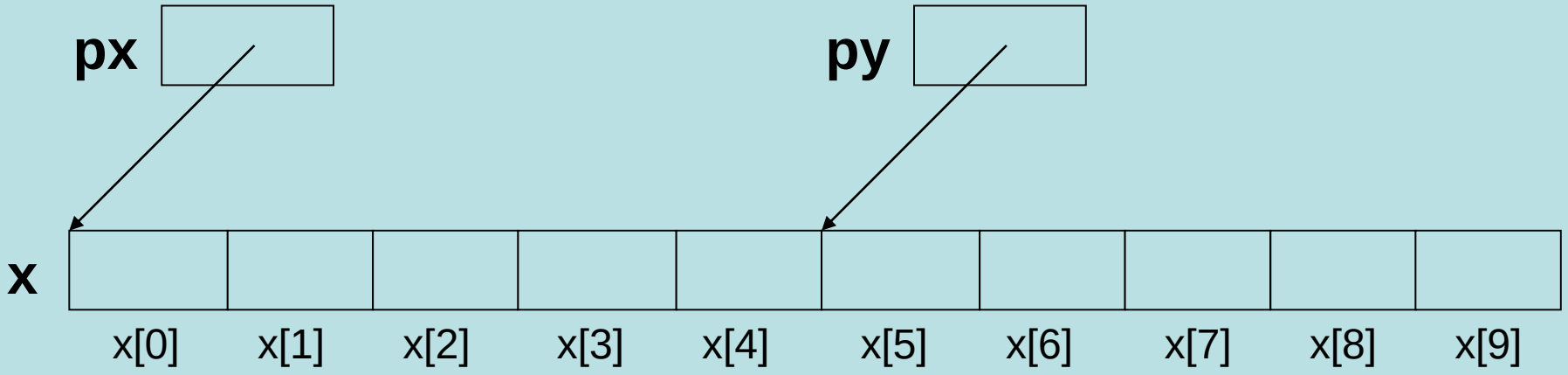
- Ku smerníku je možné pripočítať alebo odpočítať celé číslo
  - ak ku smerníku **p** pripočítame celé číslo **n**, výsledok bude ukazovať na n-tý objekt za objekt, na ktorý ukazuje p, bez ohľadu na veľkosť objektu



Smerníková aritmetika - odčítanie smerníkov

- Je možné odčítavať dva smerníky toho istého typu, ak ukazujú na prvky toho istého poľa. Výsledkom je vzdialenosť (indexová) medzi týmito dvoma prvkami

py – px = …



Smerníková aritmetika - zhrnutie

- Operácie, ktoré sa dajú so smerníkom robiť:
  - Sprístupniť obsah (*)
  - Získať adresu (&)
  - Pripočítať/odpočítať celé číslo
  - Porovnať smerník na NULL
- Ak sú rovnakého typu a ukazujú do toho istého poľa
  - Odpočítať dva smerniky
  - Porovnávať dva smerníky

Práca so smerníkom - program

```
// vynuluj pole – použi smerník
int pole[10];
int *p;
for(p=pole ; p-pole<10 ; p++){
        *p=0;
}
```

- Pozor na rozdiel p1++, (*p1)++, *p1++

Univerzálny smerník

void *ptr;

- Smerník na void
- Môžeme doňho priradiť hodnotu smerníka akéhokoľvek typu
- Nemôžeme vykonávať niektoré operácie, zo smerníkovej aritmetiky lebo nie je známy typ, na ktorý ukazujú:
  - nemôžeme pripočítať ani odpočítať
  - môžeme porovnávať ==,!=,<,>,<=,>=
- Nemôžeme sprístupniť obsah (operátor *) pamäte, na ktorú ukazuje smerník

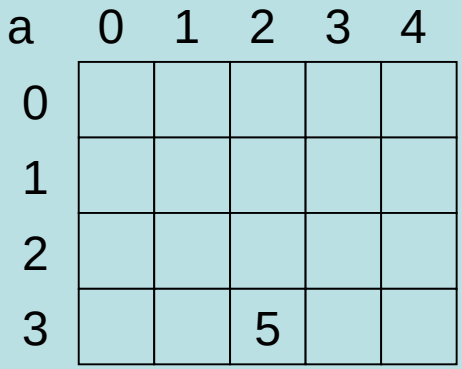
Smerníky kontra 2-rozmerné polia

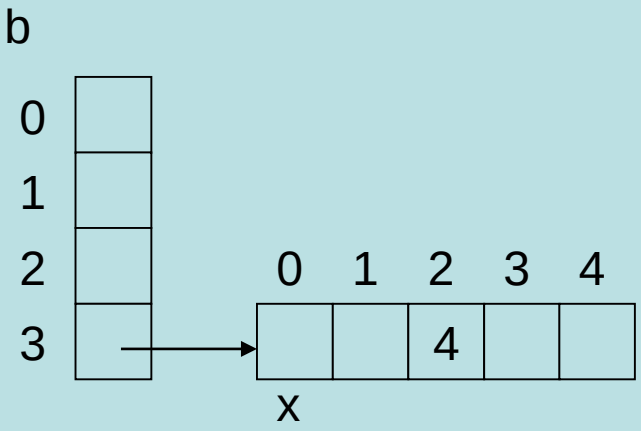
```
// dvojrozmerné pole typu int
int a[4][5];

main()
{
    a[3][2]=5; // prvok a[3,2]
}
```

- Dvojrozmerné pole je vhodné pri veľkom zaplnení

```
// pole smerníkov na int
int *b[4];
int x[5];

main()
{
    b[3]=x;
    b[3][2]=4;
    // prvok b[3] => x[2]
}
```

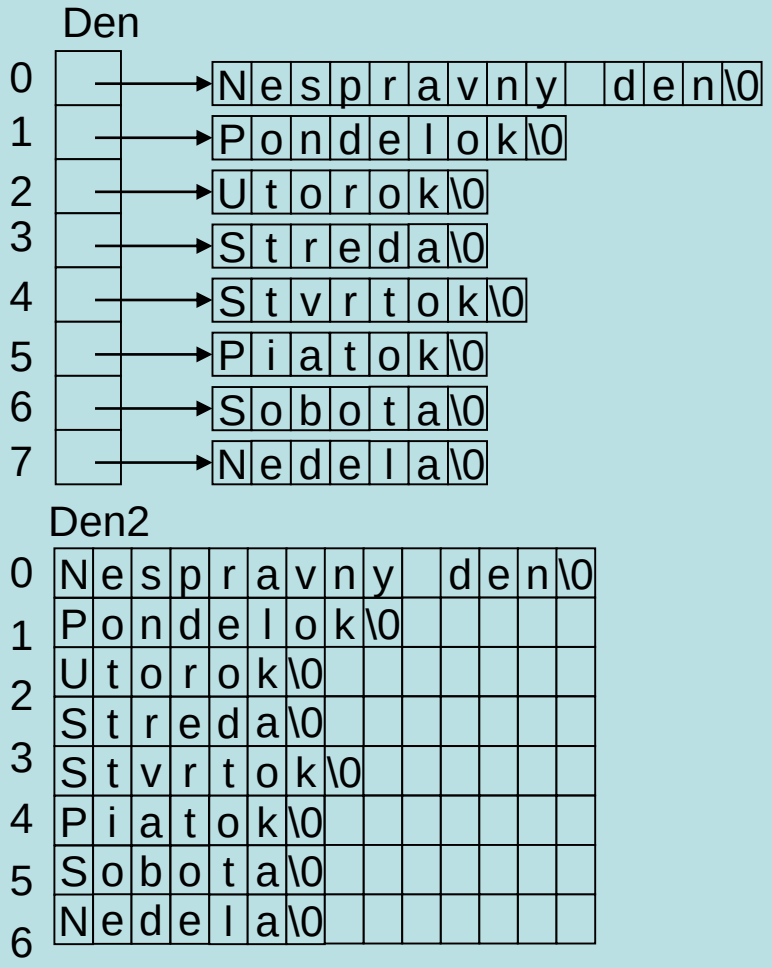
- Pole smerníkov je vhodnejší pre riedke matice





Inicializácia polí smerníkov

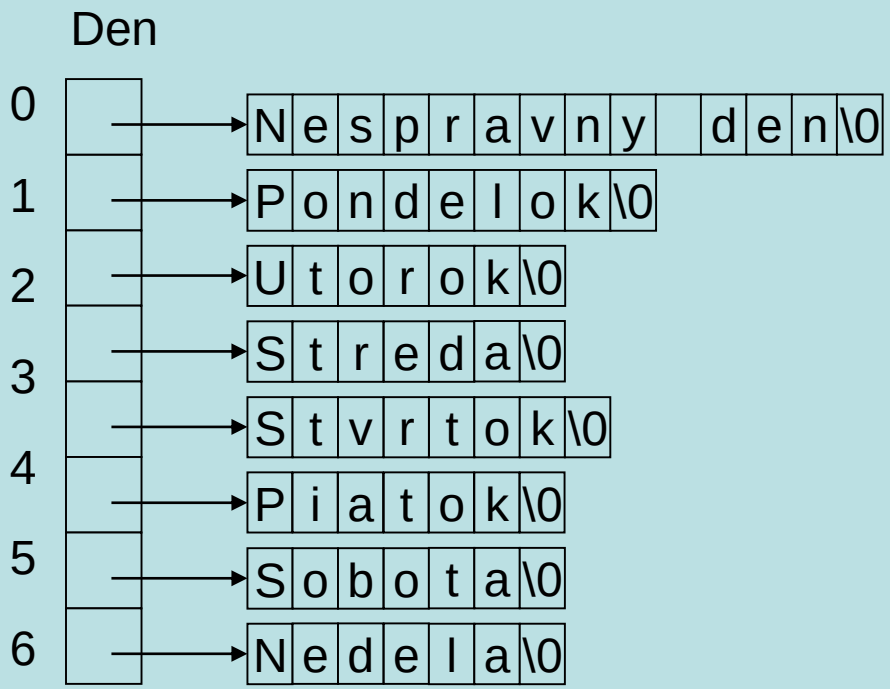
```
char *Den[ ] = {
                “Nespravny den”,
                "Pondelok“,
                "Utorok“,
                "Streda“,
                "Stvrtok“,
                "Piatok“,
                "Sobota“,
                "Nedela“
};
char Den2[ ][15] = {
                “Nespravny den”
                "Pondelok“,
                "Utorok“,
                "Streda“,
                "Stvrtok“,
                "Piatok“,
                "Sobota“,
                "Nedela“
};
main()
{
    putchar( Den [1] [3] );   // vypíše ‘d’
    putchar( Den2 [1] [3] ); // vypíše ‘d’
}
```



- Pole smerníkov na... môže zaberať viac pamäti (dodatočné smerníky)
- Pre pole smerníkov treba každý smerník zvlášť inicializovať – pomôže inicializácia pri definícii

Dvojnásobný smerník

```
char *Den[ ] = {
   “Nespravny den”,
   "Pondelok“,
   "Utorok“,
   "Streda“,
   "Stvrtok“,
   "Piatok“,
   "Sobota“,
   "Nedela“
};

char **DvojDen=Den;
putchar(DvojDen[2][3]);
// vypíše ‘r’
```



operátor ->

- Prístup k položke štruktúry/triedy cez smerník

```
struct Student
{
   char meno[20];
   int vyska;
   int znamky[10];
};

Student  jano, kruzok[20], *ptr;   // ptr je smerník na objekt
ptr=&jano;
```

- Normálne

```
(*ptr).znamky[3]=2;
```

- Pomocou '->'

```
ptr->znamky[3]=2;
```

operátor ->

(* smerník_naobjekt) . položka

- je ekvivalentné s

smerník_naobjekt -> položka

- To isté platí pre smerník na struct, union

Smerník na funkciu

- môžeme pracovať ako s hocijakým iným smerníkom
- môže ukazovať iba na funkciu s rovnakým počtom a typmi parametrov ako bol deklarovaný
- Najčastejšie použitie:
  - implementácia univerzálnych algoritmov (napr. sort)
  - spätné volanie s funkcie (callback)
  - ošetrovanie chybových stavov (_new_handler)

Smerník na funkciu

- Definícia:

```
typ (*f)(typ1 p1, typ2 p2);
```

- Príklad:

```
int (*f1ptr)();
int *(*f2ptr)(int, int);
int (*f3[10])();
int *(*f4[10])(char, char);
typedef int *(*f5[10])(char, char);
```

Práca so smerníkom na funkciu

- Priradenie hodnoty smerníka

```
double (*f)(double x);

f=&sin;          // áno
f=sin;           // áno
f=&sin(3);       // takto nie !!!
```

- Volanie funkcie cez smerník

```
double (*f)(double x);
f=&sin;
double y;
y=(*f)(3.14);    // áno
y=f(3.14);       // áno
```

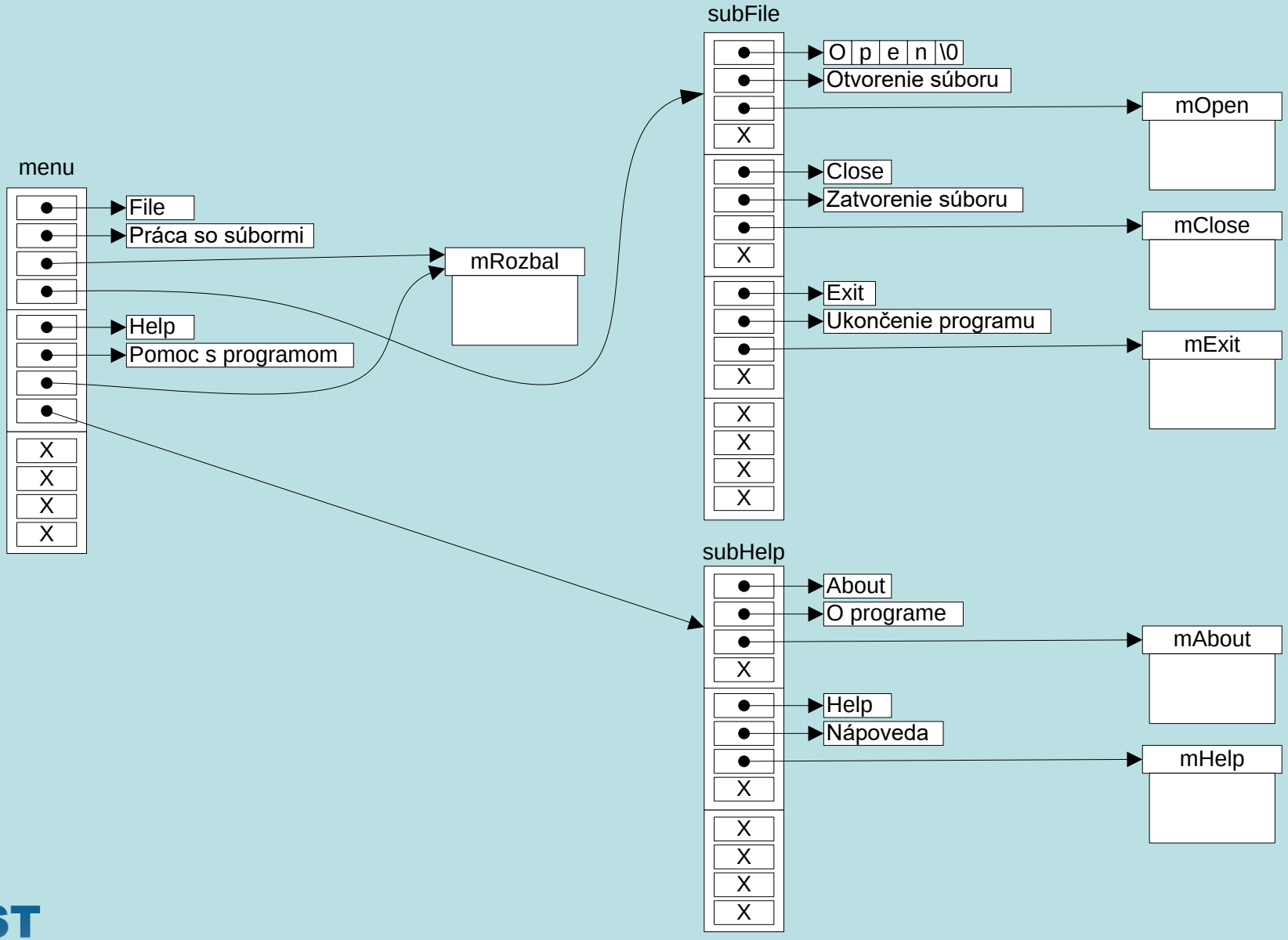
Smerník na funkciu ako parameter funkcie

```
void Vymen(char* ptr1, char* ptr2, int velkostPolozky)
{
   char pom;
   for(char* max=ptr1+velkostPolozky ; ptr1<max ; ptr1++,ptr2++){
            pom=*ptr1;
            *ptr1=*ptr2;
            *ptr2=pom;
   }
}

void _sort(void* data, int pocet, int velkostPolozky,
                        bool (*compare)(void*,void*))
{
   char* dd=(char*)data;
   char* max=dd+pocet*velkostPolozky;
   for(char* p1=dd ; p1<max ; p1+=velkostPolozky){
      for(char* p2=p1 ; p2<max ; p2+=velkostPolozky){
            if(compare(p1,p2))
                        Vymen(p1,p2,velkostPolozky);
      }
   }
}
```

Smerník na funkciu ako parameter funkcie

```
bool compareIntUp(void* x1, void* x2)
{
    return *(int*)x1 > *(int*)x2;
}
bool compareIntDown(void* x1, void* x2)
{
    return *(int*)x1 < *(int*)x2;
}
void Print(int* data, int pocet)
{
    for(int i=0;i<pocet;i++){
            printf("%i,",data[i]);
    }
    printf("\n");
}
int main()
{
    int pole[ ]={2,4,5453,5,46,56,25,6,352,31,43,56,78,4321,4,15,56,546,3};
    Print(pole,sizeof(pole)/sizeof(pole[0]));
    _sort(pole, sizeof(pole)/sizeof(pole[0]), sizeof(int), compareIntUp);
    Print(pole,sizeof(pole)/sizeof(pole[0]));
    return 0;
}
```

Použitie smerníka na funkciu - menu



const a smerníky

- smerník na znak

```
char * s;
*s='a';   // ok
s++;      // ok
```

- smerník na konštantný znak

```
const char * s;
*s='a';   // nie
s++;      // ok
```

- konštantný smerník na znak

```
char const * s;
*s='a';   // ok
s++;      // nie
```

- konštantný smerník na konštantný znak

```
const char const * s;
*s='a';   // nie
s++;      // nie
```

- príklady použitia

```
strcpy(char* dst, const char* src);
strlen(const char* s);
```

Smerníky a argumenty funkcií

- V jazyku C++ sa odovzdávajú parametre funkciám hodnotou
- Nemáme žiadny priamy spôsob, ako vytvoriť funkciu, ktorá by:
  - menila obsah premennej vo volajúcej funkcii
  - vracala viac ako jednu hodnotu
- Na takúto zmenu sa používajú smerníky
- Ako parameter funkcii dáme smerník na premennú
- Smerník je odovzdaný hodnotou ale ukazuje na pamäť ako pôvodný smerník

Smerníky a argumenty funkcií

- Napíšte funkciu, ktorá vymení medzi sebou dva parametre

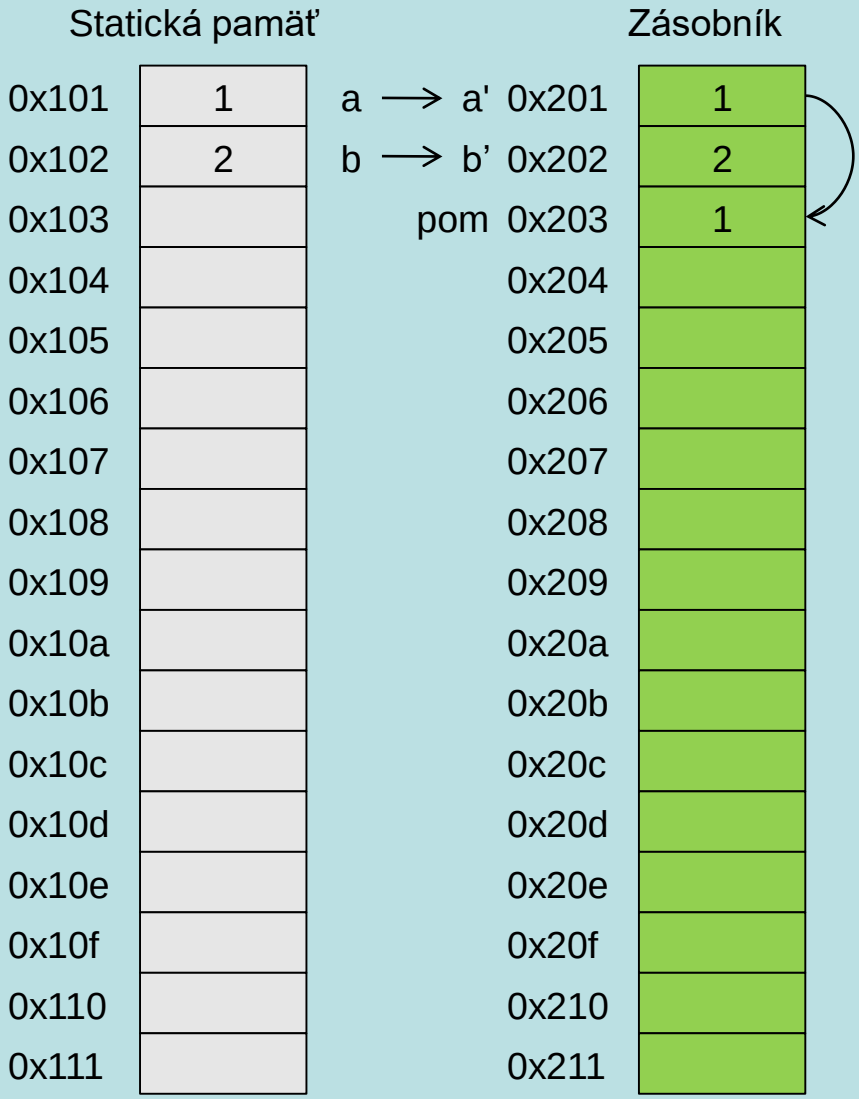
```
void vymena1(byte a, byte b)
{                                       // nesprávne
    byte pom = a;  a = b;  b = pom;
}

void vymena2(byte *a, byte *b)
{                                       // správne
    byte pom = *a;  *a = *b;  *b = pom;
}

void vymena3(byte &a, byte &b)
{                                       // správne
    byte pom = a;  a = b;  b = pom;
}
```

- volanie:

```
byte a=1, b=2;

int main()
{
    vymena1(a,b);       // na tomto mieste je v a=1 a v b=2
    vymena2(&a, &b);    // na tomto mieste je v a=2 a v b=1
    vymena3(a,b);       // na tomto mieste je v a=2 a v b=1
    return 0;
}
```

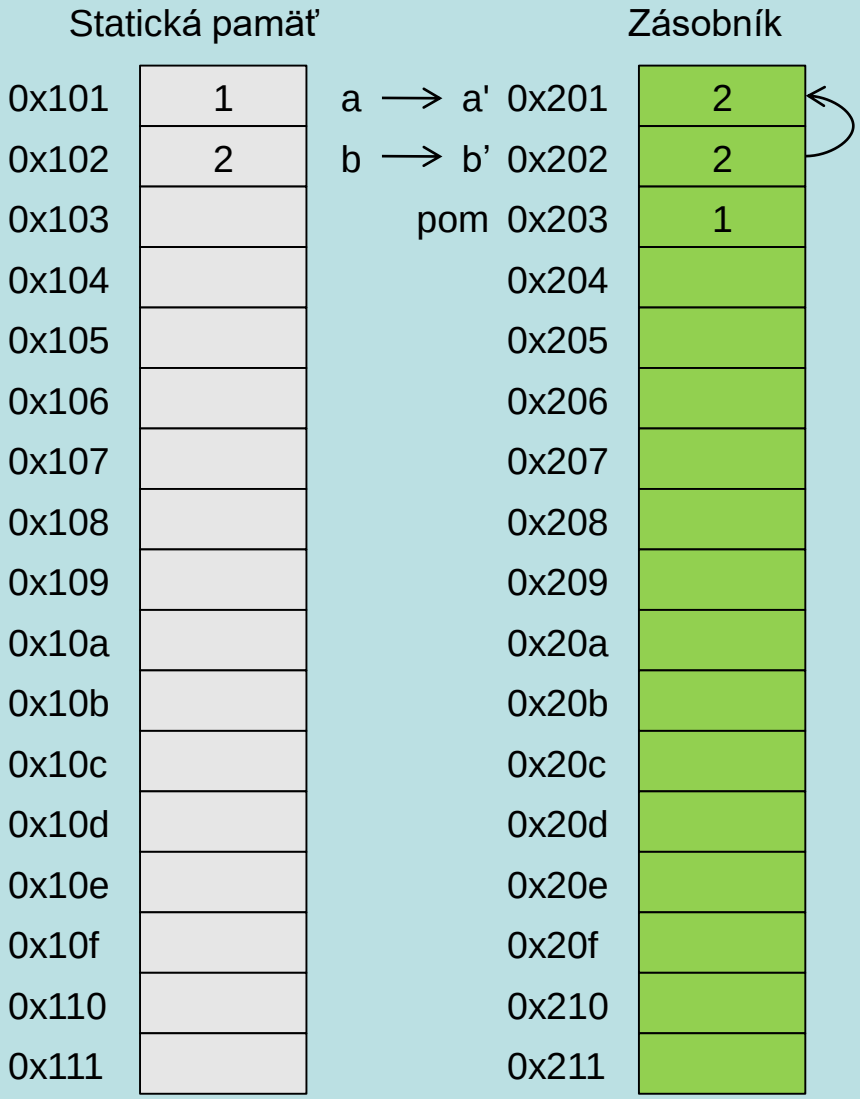
Prenos parametrov hodnotou - 1

```
void vymena1(byte a, byte b)
{
    byte pom = a;
    a = b;
    b = pom;
}

int a=1, b=2;

int main()
{
    vymena1(a,b);
    return 0;
}
```



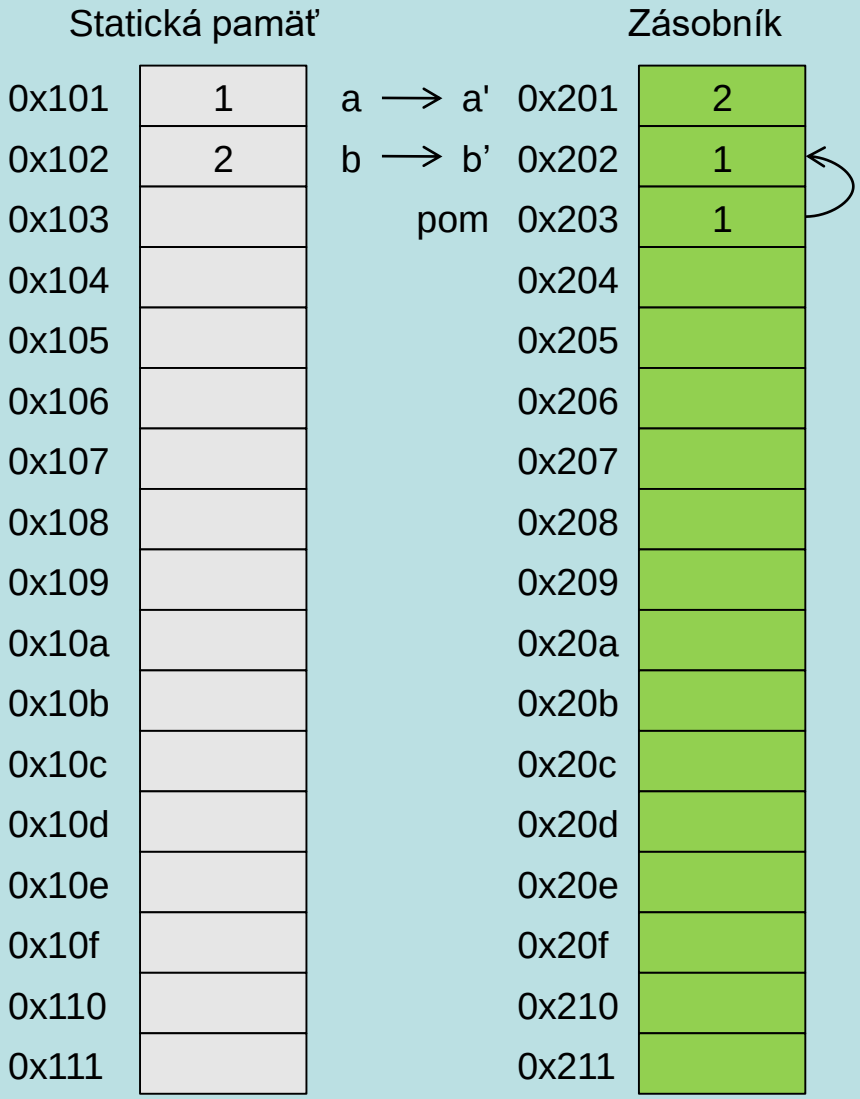
Prenos parametrov hodnotou - 2

```
void vymena1(byte a, byte b)
{
    byte pom = a;
    a = b;
    b = pom;
}

int a=1, b=2;

int main()
{
    vymena1(a,b);
    return 0;
}
```



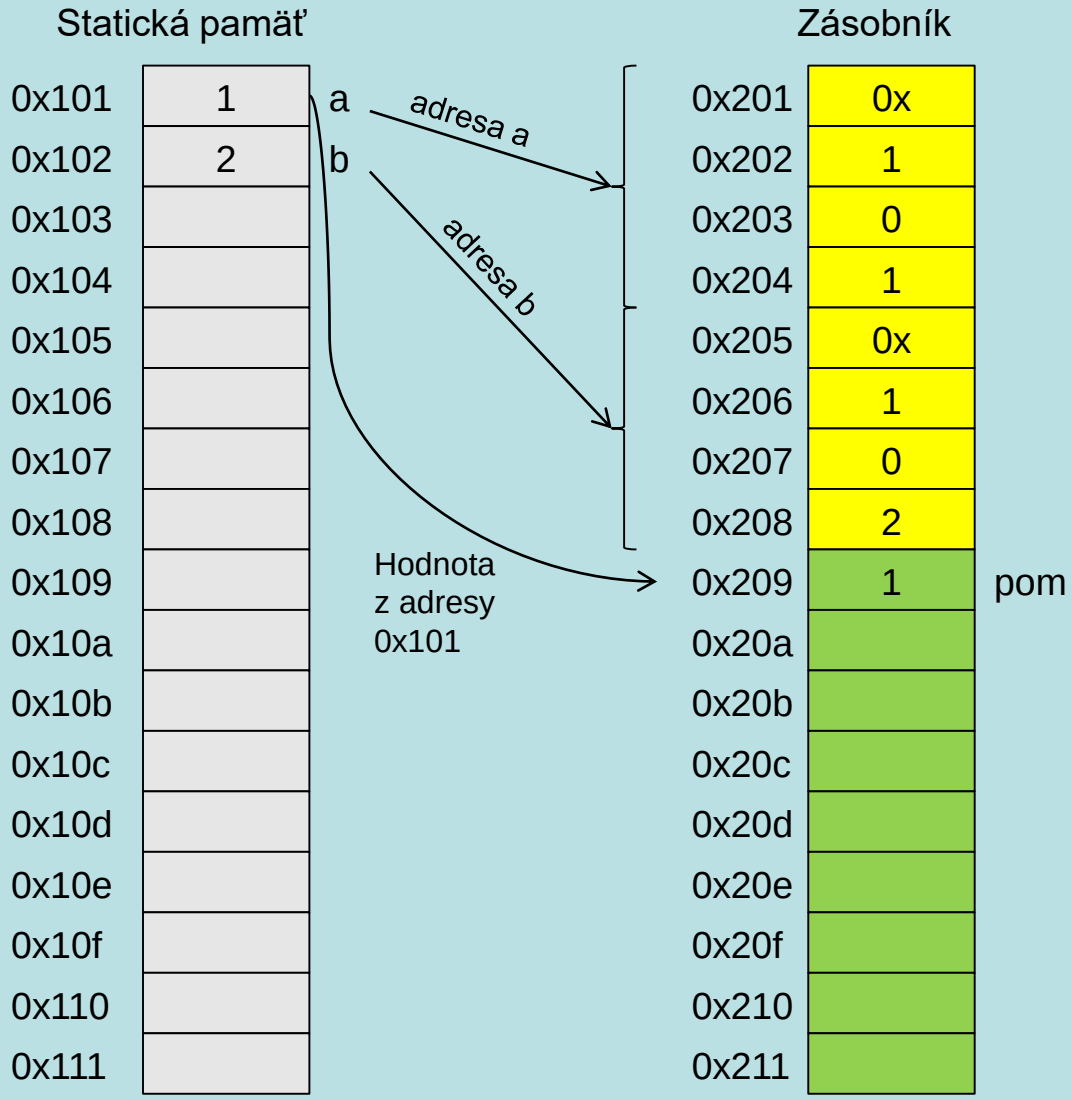
Prenos parametrov hodnotou - 3

```
void vymena1(byte a, byte b)
{
    byte pom = a;
    a = b;
    b = pom;
}

int a=1, b=2;

int main()
{
    vymena1(a,b);
    return 0;
}
```



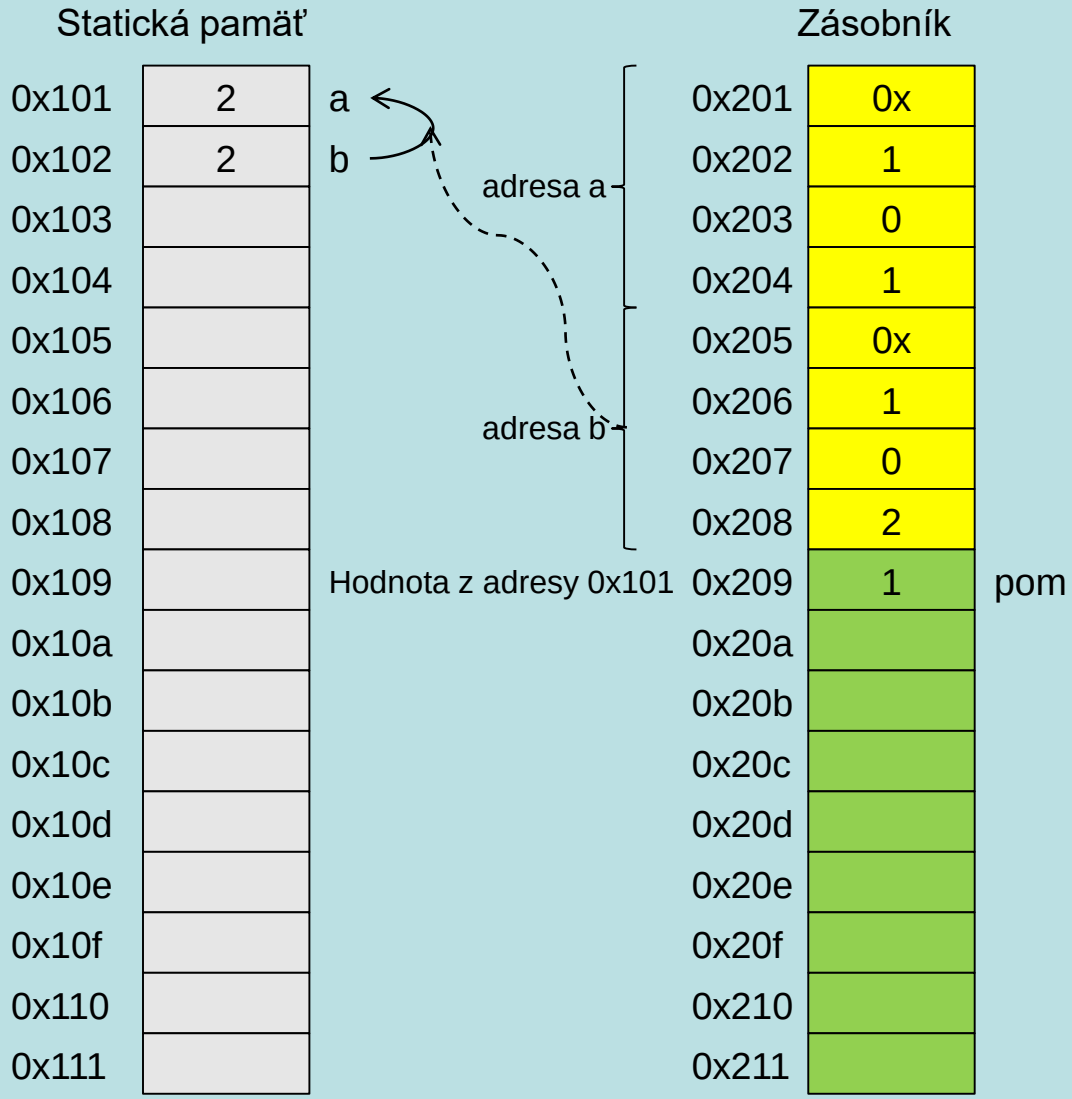
Prenos parametrov smerníkom - 1

```
void vymena2(byte *a, byte *b)
{
    byte pom = *a;
    *a = *b;
    *b = pom;
}

byte a=1, b=2;

int main()
{
    vymena2(&a,&b);
}
```



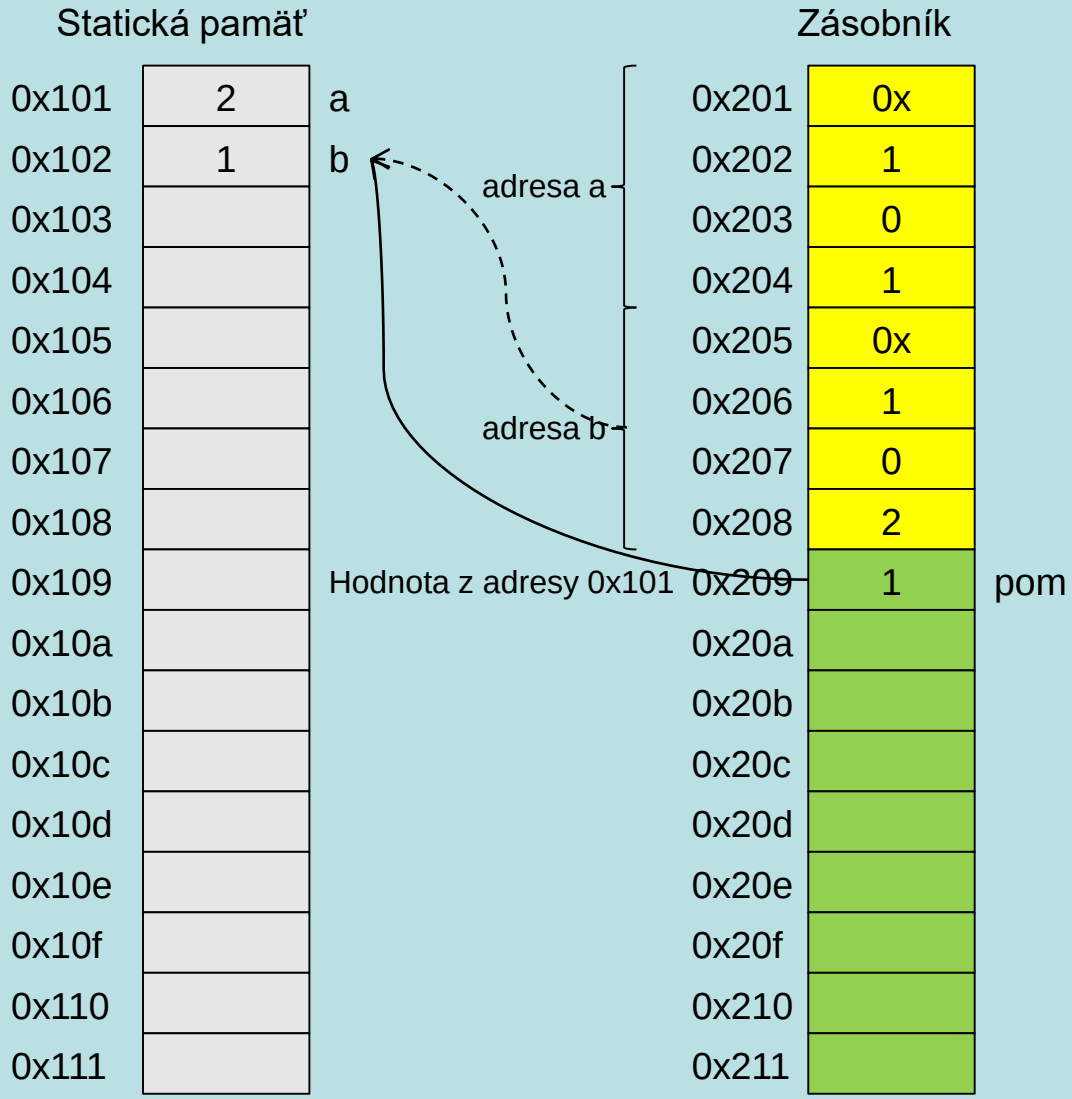
Prenos parametrov smerníkom - 2

```
void vymena2(byte *a, byte *b)
{
    byte pom = *a;
    *a = *b;
    *b = pom;
}

byte a=1, b=2;

int main()
{
    vymena2(&a,&b);
}
```



Prenos parametrov smerníkom - 3

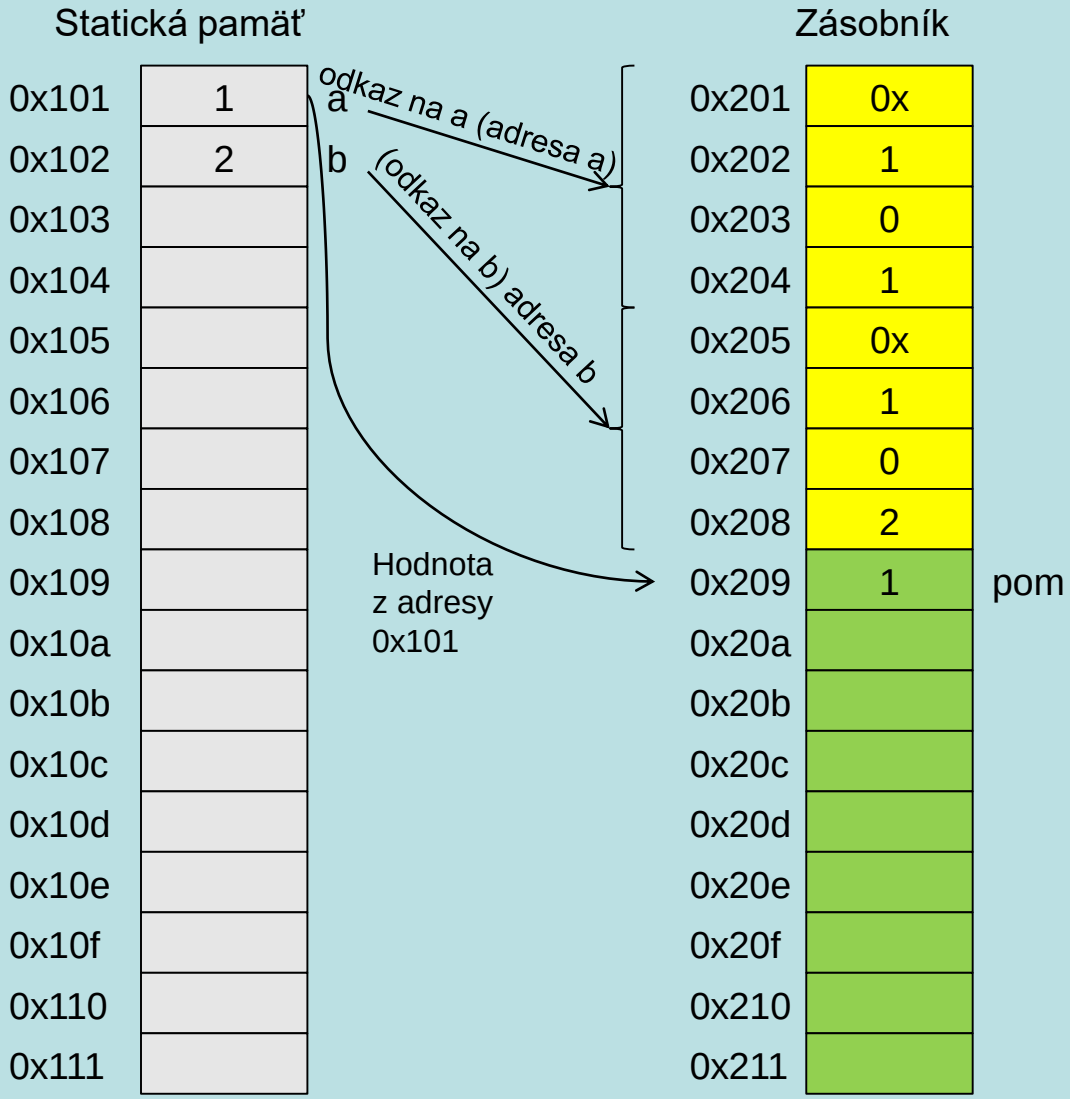
```
void vymena2(byte *a, byte *b)
{
    byte pom = *a;
    *a = *b;
    *b = pom;
}

byte a=1, b=2;

int main()
{
    vymena2(&a,&b);
}
```



Prenos parametrov odkazom - 1

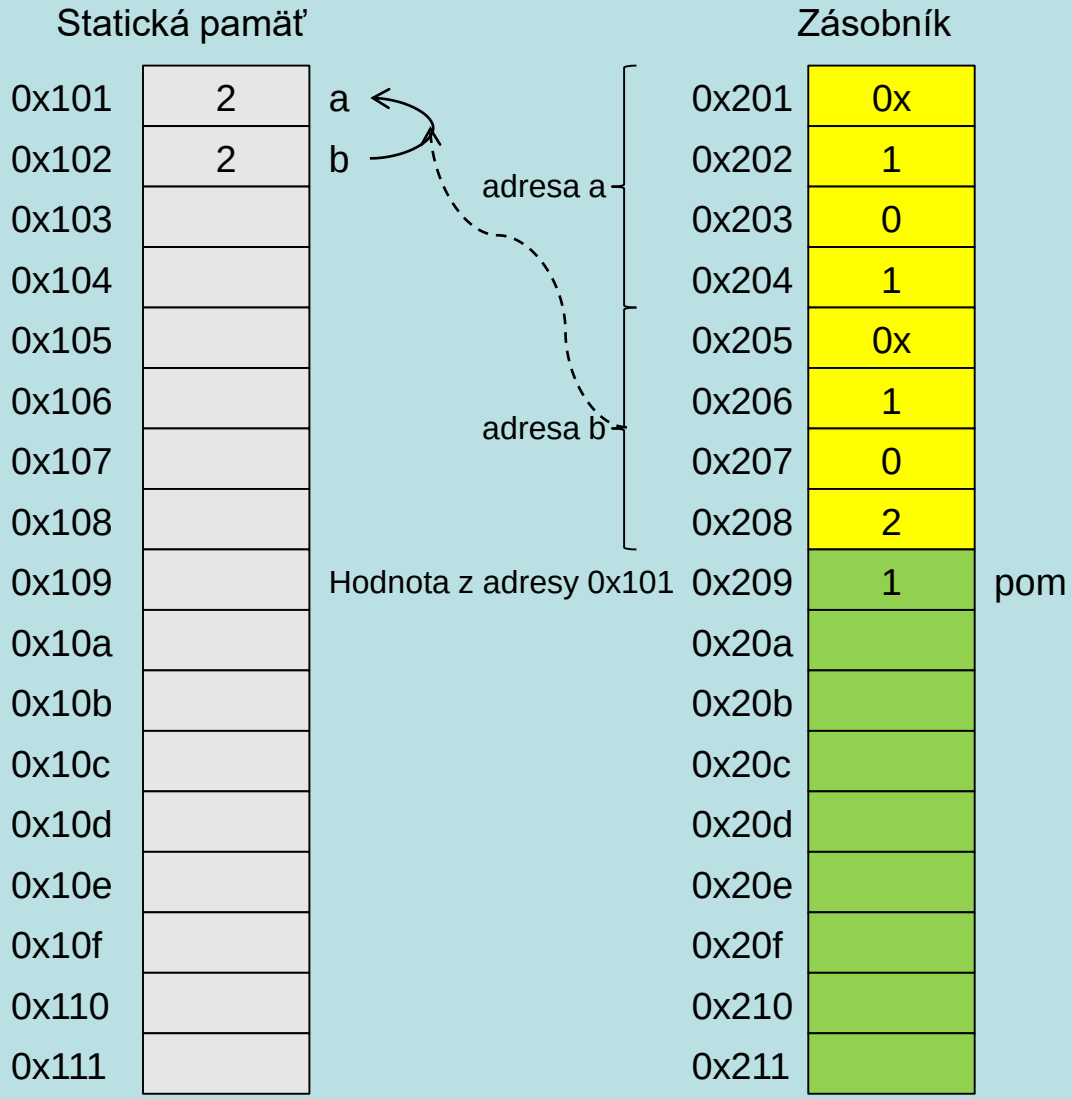
```
void vymena3(byte &a, byte &b)
{
    byte pom = a;
    a = b;
    b = pom;
}

byte a=1, b=2;

int main()
{
    vymena3(a,b);
}
```

```
void vymena2(byte &a, byte &b)
{
    byte pom = &*a;
   &*a = &*b;
   &*b = pom;
}

byte a=1, b=2;

int main()
{
    vymena2(&a,&b);
}
```



Prenos parametrov odkazom - 2

```
void vymena3(byte &a, byte &b)
{
    byte pom = a;
    a = b;
    b = pom;
}

byte a=1, b=2;

int main()
{
    vymena3(a,b);
}
```



Prenos parametrov odkazom - 3

```
void vymena3(byte &a, byte &b)
{
    byte pom = a;
    a = b;
    b = pom;
}

byte a=1, b=2;

int main()
{
    vymena3(a,b);
}
```