Algoritmizácia a programovanie
8. prednáška

Ján Grman

Obsah prednášky

1. Dynamické pridelovanie pamäte
2. Viacrozmerné polia

Dynamické prideľovanie a uvoľňovanie pamäte

- prideľovanie pamäte za chodu programu
  - v zásobníku (stack) - riadi operačný systém
  - v hromade (heap) - riadi programátor

budeme sa zaoberať týmto prideľovaním

- pomocou *run-time* funkcií
- životnosť dynamických dát:
  - od alokovania po uvoľnenie pamäte

Prideľovanie pamäte

- pomocou funkcie definovanej v **`stdlib.h`** (niekedy v **`alloc.h`**):

počet Bytov

```
void *malloc(unsigned int)
```

Adresa prvého prideleného prvku - je vhodné pretypovať.
Ak nie je v pamäti dosť miesta, vráti NULL.

Testovanie pridelenia pamäte

- kontrola, či **`malloc()`** pridelil pamäť:

```
int * p_i;

if((p_i = (int *) malloc(1000)) == NULL) {
    printf("Nepodarilo sa pridelit pamat\n");
    exit;
}
```

Kedy potrebujeme prideliť pamäť

- Inicializácia premennej – ukazovateľa vytvorí miesto v pamäti pre ukazovateľ:

```
int *p_i;
```

- Pri použití ukazovateľa len na prístup k už vyhradeným premenným (miestam v pamäti) nie je potrebné alokovať pamäť!

```
int pole[20];
for(p_i = pole; p_i < pole+20; p_i++)
   printf("%d ", *p_i);
```

- Alokujeme len vtedy, keď v skutočnosti potrebujeme používať ďalšiu pamäť!

```
int i;
p_i = (int *) malloc(1000*sizeof(int));
for(i = 0; i<20; i++)
   *(p_i + i) = pole[i];
```

Uvoľňovanie pamäte

- nepotrebnú pamäť je vhodné ihneď vrátiť operačnému systému
- pomocou funkcie: `void free(void *)`

príklad:

```
char *p_c;

p_c = (char *) malloc(1000 * sizeof(char));
...
free(p_c);
p_c = NULL;
```

Príklad prideľovania pamäte: pre jeden char

```
char *p_c;
if ((p_c = (char *) malloc(1)) == NULL)
    ... /* chybová správa a ukončenie */

...
free(p_c);
```

správne, ale neošetrujeme pamäť

c

pamäte - program spadne

Príklad prideľovania pamäte: pre jeden `int`

```
int *p_i;
if ((p_i = (int *) malloc(1 * sizeof(int)))
    == NULL) {
    printf("Nie je dostatok pamate\n");
    exit(1);
}
free(p_i);
```

`sizeof(p_i) == 2`

Funkcia `calloc()`

- rovnkako ako **`malloc()`**, len automaticky inicializuje Byty na 0:

```
void *calloc(unsigned int)
```

Príklad: načítanie 5 čísel a vypočítanie ich súčinu

- 3 pomocné funkcie + funkcia **`main()`**:
  - alokovanie pamäte **n** čísel
  - načítanie **n** čísel z klávesnice
  - vypočítanie súčinu **n** čísel

Príklad: funkcia na alokovanie pamäte n čísel

```
int *alokuj(int n)
{
    return ((int *) malloc(n * sizeof(int)));
}
```

Príklad: funkcia na načítanie n čísel z klávesnice

```
void nacitaj(int *p_i, int n)
{
    int i;

    for (i = 0; i < n; i++) {
        printf("Zadajte %d-te cislo: ", i+1);
        scanf("%d", p_i + i);
}
```

Príklad: funkcia na vypočítanie súčinu n čísel

```
void sucin(int *p, int n, int *sucin)
{
    int i;

    *sucin = 1;
    for (i = 0; i < n; i++)
        *sucin *= *(p + i);
}
```

pozrite si kratšie napísanú funkciu v (Herout)

Príklad: funkcia main

```
int main()
{
    int *cisla, suc;
    cisla = alokuj(N);
    nacitaj(cisla, N);
    sucin(cisla,N,& suc);
    printf("Sucin je: %d\n", suc);
    return 0;
}
```

doplňte, aby to bolo správne

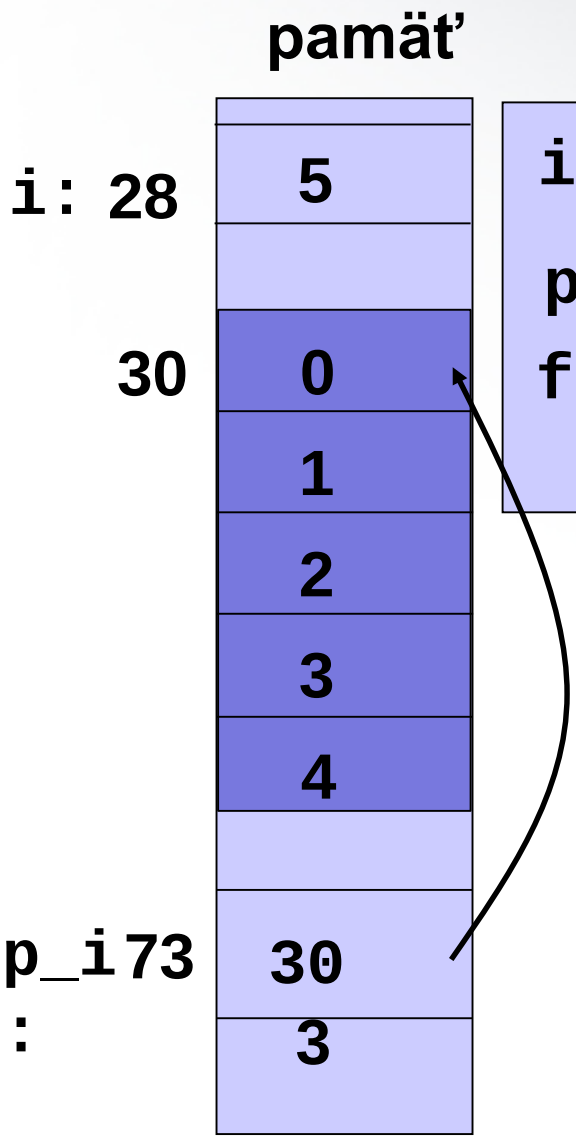
na začiatok programu nezabudnúť:

```
#include <stdio.h>

#include <stdlib.h>

#define N 5
```

Dynamické polia




```
int i=5, *p_i;

p_i = (int *) malloc(n*sizof(int)));
for(i=0; i<n; i++)
   p_i[i] = i;
```

Prepíšte pomocou ukazovateľovej aritmetiky

Ukazovatele na funkcie

- Funkcia môže vrátiť ukazovateľ na typ:
  - **`FILE *fopen(...)`** vracia smerník na typ **`FILE`**
- Definovanie premennej ako ukazovateľ na funkciu: napr. **`double (*p_fd)();`**

```
double (*p_fd)();
```

ukazovateľ na funkciu

```
double scitaj(double x, double y)
p_fd = scitaj;
```

`p_fd` má adresu funkcie **`scitaj()`**

Príklad ukazovateľa na funkciu

funkcia na výpočet hodnôt polynómov (napr. $x^2 + 3$, $x + 8$) pre zadanú hornú, dolnú hranicu a krok
- najprv pomocné funkcie pre polynómy

```
double pol1(double x)
{
    return  (x * x + 3);
}
```

```
double pol2(double x)
{
    return  (x + 8);
}
```

Príklad ukazovateľa na funkciu

funkcia **`vypis()`** na vypísanie tabuľky

```
void vypis(double d, double h, double k,
double (*p_f)()) {
    double x;
    for(x=d; x<=h; x+=k)
        printf("%lf, %lf \n", x, (*p_f)(x));
}
```

volanie
vo funkcii main():

```
vypis(-1.0, 1.0, 0.1, pol1);
vypis(-2.0, 2.0, 0.05, pol2);
```

Príklad ukazovateľa na funkciu

Program bude načítavať písmená. Po stlačení **`'A'`** vypíše **`Ahoj`**, po stlačení **`'C'`** vypíše **`Cau`**, po stlačení **`'K'`** skončí. Použijeme ukazovateľ na funkcie.

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

void ahoj() {
       printf("Ahoj\n");
}

void cau() {
       printf("Cau\n");
}

int main() {
   int c;
   void (* p_fnc)();    /* definicia ukazovatela na funkciu */

   printf("Ahoj / Cau / Koniec\n");
   while((c = toupper(getchar())) != 'K') {
      if (c == 'A') p_fnc = ahoj;
      else if (c == 'C') p_fnc = cau;
       else continue;

       (*p_fnc)();
   }
   return 0;
}
```

Pole ukazovateľov na funkcie

- prvkami poľa môžu byť aj ukazovatele
  - na prvky → viacrozmerné polia
  - na funkcie (všetky funkcie musia byť toho istého typu)

```
typedef void (* P_FNC)();


P_FNC  funkcie[10];
```

definícia ukazovateľa na funkciu vracajúcu typ void

pole 10 ukazovateľov

Pole ukazovateľov na funkcie

- pole ukazovateľov na funkcie pri riadení programu pomocou menu

```
typedef void (* P_FNC)();

P_FNC  funkcie[5] = {file, edit, search,
     compile, run} ;
...
```

- volanie funkcie:

```
funkcia[1]();
```

Príklady definícií

`int i;`	- **`i`** je typu **`int`**
`float *y;`	- **`y`** je ukazovateľ na typ **`float`**
`double *z();`	- **`z`** je funkcia vracajúca ukazovateľ na **`double`**
`int (*v)();`	- ukazovateľ na funkciu vracajúcu **`int`**
`int *(*v)();`	- ukazovateľ na funkciu vracajúcu ukazovateľ na **`int`**

Ako čítať zložitejšie definície

1. Nájdeme identifikátor, od neho čítame doprava
2. pokým nenarazíme na samotnú pravú zátvorku ")". Vraciame sa k zodpovedajúcej ľavej zátvorke. Potom pokračujeme doprava (preskakujeme prečítané)
3. Ak narazíme na ";" , vraciame sa na najľavejšie spracované miesto v čítame doľava

Príklad: čítanie definície
int *(*v)();

```
int *(*v)());
```

- **v** je pointer na funkciu vracajúcu pointer na **int**

1. Nájdeme identifikátor: **v**, čítame doprava
2. Nájdeme **)**, k nej zodpovedajúcu **(**, od nej čítame doprava: ***
3. Doprava, preskakujeme prečítané, po **)**, k nej **(**
4. Doprava, preskakujeme prečítané, po **;**, doľava

Definícia s využitím typedef

- Operátor typedef
  - vytvára nový typ
  - najmä na definovanie zložitejších typov

```
typedef float *P_FLOAT;
```

P_FLOAT je ukazovateľ na typ **float**

Príklady použitia typedef

→
```
int *p_i, **p_p_i;
```

p_i – ukazvateľ na **int**
p_p_i – ukazvateľ na ukazovateľ na **int**

je ekvivalentné

```
typedef int *P_INT;
typedef P_INT *P_P_INT;

P_INT  p_i;
P_P_INT p_p_i;
```

→
```
typedef double (*P_FD)();
```

ukazovateľ na funkciu vracajúcu **double**

Príklad

program načíta celé číslo **n** a alokuje blok pamäte pre **n** celých čísel. Od používateľa čísla načíta. Nakoniec vypočíta ich priemer.

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

int *alokuj(int pocet);
void nacitaj(int *pole, int pocet);
float priemer(int *pole, int pocet);
void vypis(int *pole, int pocet);

int main()
{
   int *pole, n;

   printf("Zadajte pocet cisel: ");
   scanf("%d", &n);
   if ((pole = alokuj(n)) == NULL) {
      printf("Nepodarilo sa alokovat pole.\n");
      return 1;
   }
   nacitaj(pole, n);
   printf("Priemer cisel: \n");
   vypis(pole, n);
   printf("je %.3f.\n", priemer(pole, n));
   free(pole);
   return 0;
}
```

```
int *alokuj(int pocet)
{
   return (int *) malloc(pocet * sizeof(int));
}

void nacitaj(int *pole, int pocet)
{
   int i;
   for(i = 0; i < pocet; i++) {
      printf("%d-te cislo: ", i);
      scanf("%d", pole + i);
   }
}

float priemer(int *pole, int pocet)
{
   int i, suma = 0;
   for(i = 0; i < pocet; i++)
      suma += *(pole + i);
   return (float) suma / (float) pocet;
}
```

```
void vypis(int *pole, int pocet)
{
   int i;
   for(i = 0; i < pocet; i++)
      printf("%d, ", *(pole + i));
}
```

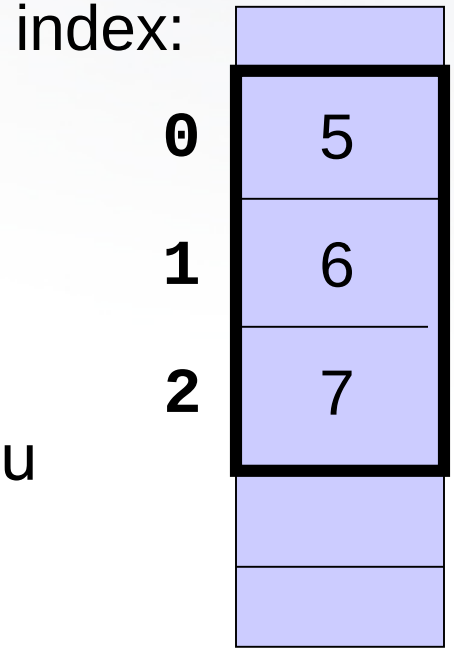
Základy práce s poliami (staticky)

- pole je štruktúra zložená z niekoľkých prvkov rovnakého typu (blok prvkov)

```
TYP x[N];
```

statická definícia poľa

N: 3
TYP: **int**



- pole obsahuje **N** prvkov
- dolná hranica je vždy **0**

  ⇒ horná hranica je **N-1**
- číslo **N** musí byť známe v čase prekladu
- hodnoty nie sú inicializované na 0

Polia a ukazovatele

- adresa `i`-teho prvku poľa `x`:

  `&x[i] =` bázová adresa `x + i * sizeof(typ)`

  – `x` je adresa v pamäti
- platí:

  `x + i == &x[i]`

  `*(x + i) == x[i]`

Polia a ukazovatele

```
int *p;
p = (int *) malloc(4 * sizeof(int));
```

- platí:
  - `p[0] == *p`
  - `p[1] == *(p + 1)`
  - `p[2] == *(p + 2)`
  - `p[3] == *(p + 3)`

Rozdiel medzi statickými a dynamickými poliami je najmä v spôsobe prideľovania pamäte.

Polia a ukazovatele

```
int x[4];
```

- platí:
  - `&x[0] == &*(x + 0) == x`
  - `&x[i] == &*(x + i) == (x + i)`

`x` je statický ukazovateľ, nemôžeme spraviť `x = p_i;`

môžeme ale urobiť `*x = 2;` (to isté ako `x[0] = 2;`)

Príklad prístupu k prvkom poľa pomocou ukazovateľa

Prepísanie nasledujúcej časti programu tak, aby sa k poľu **`slovo`** pristupovalo prostredníctvom ukazovateľov.

```
i = 0;                          /* naplnenie pola */
while (i < N && (slovo[i] != '\0') {
   hist[toupper(slovo[i]) - 'A']++;
   i++;
}
```

```
i = 0;                          /* naplnenie pola */
while (i < N && (*(slovo + i) != '\0') {
   hist[toupper(*(slovo + i)) - 'A']++;
   i++;
}
```

Zistenie veľkosti poľa

```
int x[10], *p_x;
p_x = (int *) malloc (10 * sizeof(int));
```

- po alokovaní pamäte pre **p_x** budú **x** aj **p_x** ukazovatele na pole 10 prvkov typu **int**, s rozdielom, že:
  - **x** je statický ukazovateľ
  - **p_x** je dynamický ukazovateľ
- preto dáva **sizeof()** iné výsledky:

  sizeof(x) == 10 * sizeof(int) (napr. 20)

  sizeof(p_x) == sizeof(int *) (napr. 4)

Pole meniace svoju veľkosť

```
int *x, n = 5, *p1, *p2, *p;

x = (int *) malloc(n * sizeof(int));
x[0] = 10;  x[4] = 3;
...
/* potreba zvacsit pole*/
p = (int *) malloc (10 * n * sizof(int));
p1 = x;
p2 = p;
while(p1 < x + n) *p2++ = *p1++;
n *= 10;
free(x);
x = p;
```

alokovanie poľa x

alokovanie poľa p

kopírovanie obsahu poľa

uvoľnenie menšieho poľa x

nastavenie x na p

Pole meniace svoju veľkosť - pomocou `realloc()`

- funkcia

  `void *realloc(void *pole, unsigned int size)`
  definovaná v **`stdlib.h`**
  - **`pole`** - ukazovateľ na pamäť
  - **`size`** - veľkosť
  - zväčší **`pole`**, alebo vytvorí nové a prekopíruje tam hodnoty z pôvodného poľa

```
x = realloc(x, 10 * n * sizeof(int));
```

Pole ako parameter funkcie

`int pole[]` je ekvivalentné `int *pole`

Pri použití **`int pole[]`** je jasnejšie, že ide o pole a nie o ukazovateľ na **`int`**.

Volanie funkcie s poľom ako parametrom:

```
max = maximum(pole, 10);
```

Pole ako parameter funkcie

```
int maximum(int *pole, int n) {...}
```

- dá sa použiť aj na zistenie maxima napr. na zistenie maxima 3. až 7. prvku

```
int x[10];
max = maximum(&x[2], 5);
```

Pole ako parameter funkcie

- prepísanie funkcie **`maximum()`** na procedúru

```
void maximum(int pole[], int n, int *p_max)
{
    int *p;
    *p_max = pole[0];
    for (p = pole + 1; p < pole + n; p++) {
        if (*p > *p_max)
            *p_max = *p;
    }
}
```

ak by sme dali **`p_max = p;`** stratili by sme ukazovateľ na premennú, kam treba vrátiť maximum

Pole ako parameter funkcie: vytváranie poľa vo funkcii

```
void init(double **p_f)
{
    double *a;
    int i;
    a = (double *) malloc(5 *
          sizeof(double);
    for(i = 0; i < 5; i++) {
        printf("Zadaj %d. cislo");
        scanf("%lf", &a[i]);
    }
    *p_f = a;
}
```

```
void main()
{
   double *p_d;
   init(&p_d);
}
```

p_d bude ukazovať na pole 5 **double** prvkov, ale **a** bolo vyrobené v zásobníku, a tento zásobník sa pri ukončovaní funkcie zruší

Viacrozmerné polia

Základné definície

- definícia dvojrozmerného poľa:

```
int x[5][4];
```

- Pri definovaní viacerých polí:

```
typedef int DVA[5][4];
DVA d;
```

- Nový typ pre dvojrozmerné pole
  - aj pomocou už existujúceho typu:

```
typedef int JEDEN[4];
typedef JEDEN DVA[5];
DVA d;
```

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16
17	18	19	20

Prístup k prvkom poľa

- pomocou indexov: rovnaký ako pre jednorozmerné polia

dvojrozmerné pole:

```
int tabulka[5][10];

tabulka[1][6] = 4;
tabulka[4][9] = 0;
```

trojrozmerné pole:

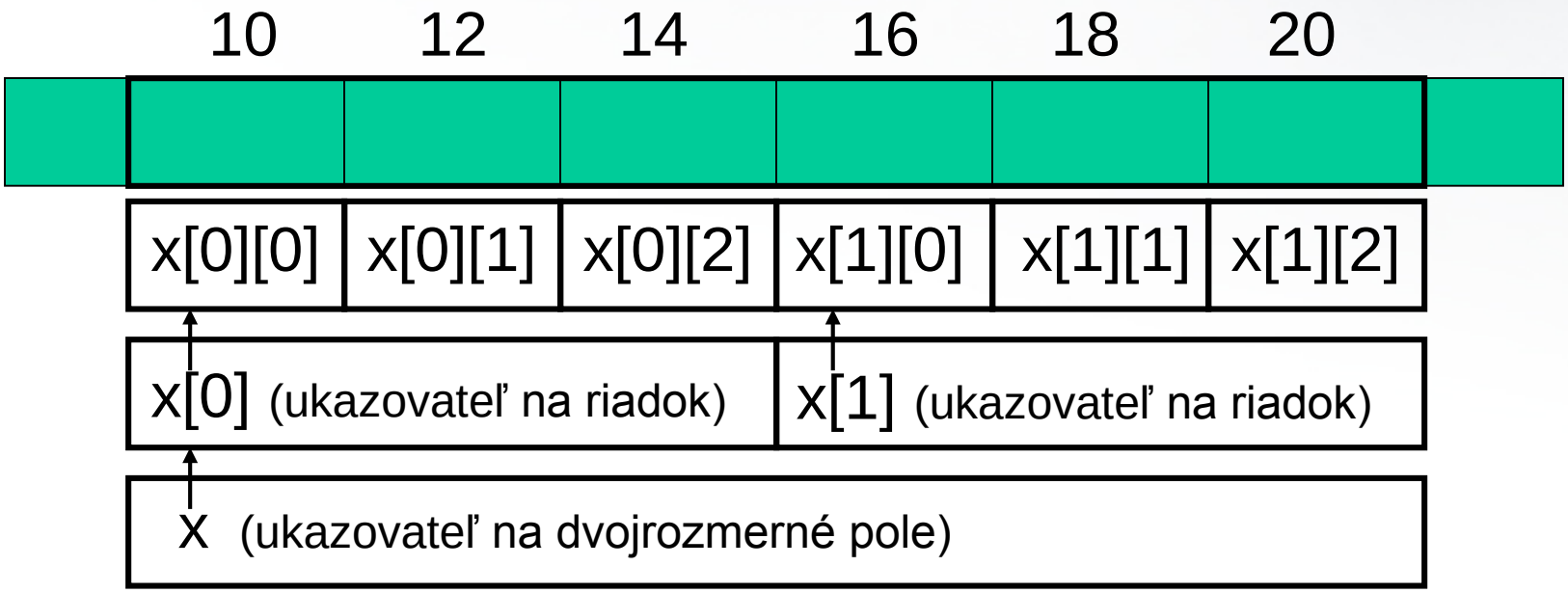
```
int trojtabulka[5][6][7];

tabulka[0][5][0] = 4;
```

Uloženie viacrozmerného poľa v pamäti

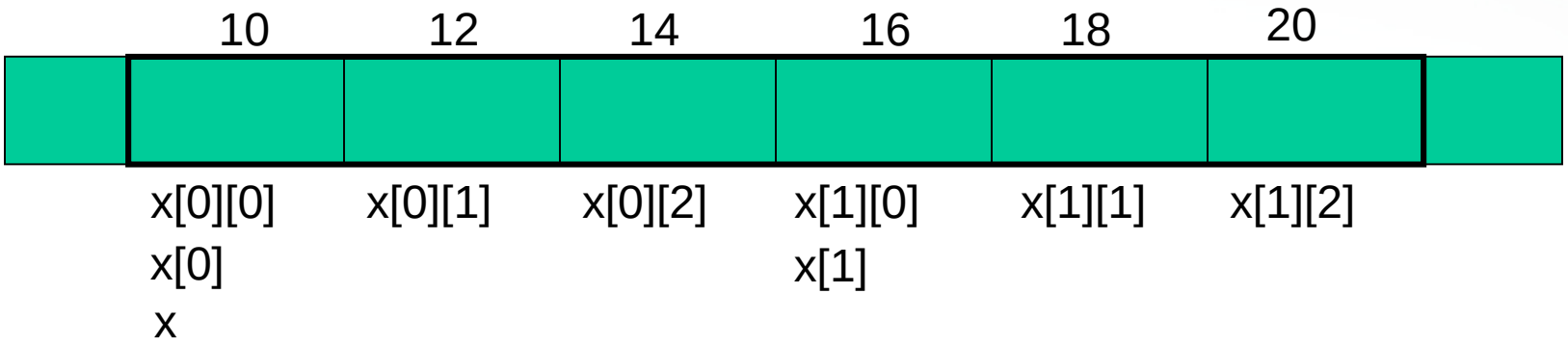
- po riadkoch

```
int x[2][3];
```



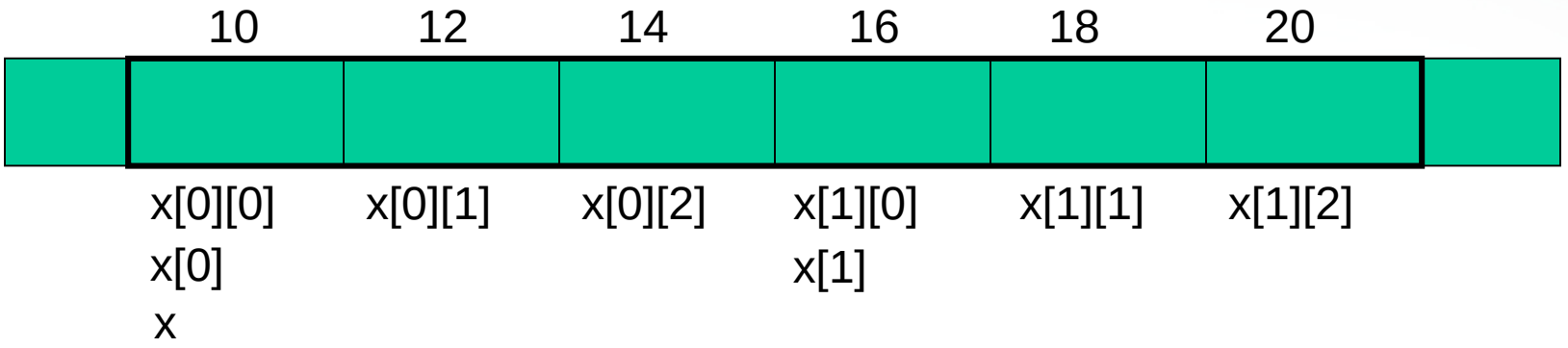
Uloženie viacrozmerného poľa v pamäti

- **`x`** a **`x[0]`** - tá istá adresa, len iného typu
- **`x+1`** a **`x[0]+1`** - predstavujú odlišné adresy
- **`x`** - ukazovateľ na dvojrozmerné pole
- **`x[i]`** - ukazovateľ na i-ty riadok
- **`*(x + 1) == x[1] ==16`** - adresa prvého riadku
- **`x[i][j]`** - hodnota prvku dvojrozmerného poľa



Uloženie viacrozmerného poľa v pamäti

- `x[i] == *(x + i)` - adresa `i`-teho riadku
- `&x[i][j] == x[i] + j == *(x + i) + j`
  - adresa premennej
- `x[i][j] == *(x[i] + j) == *(*(x + i) + j)`
  - hodnota premennej



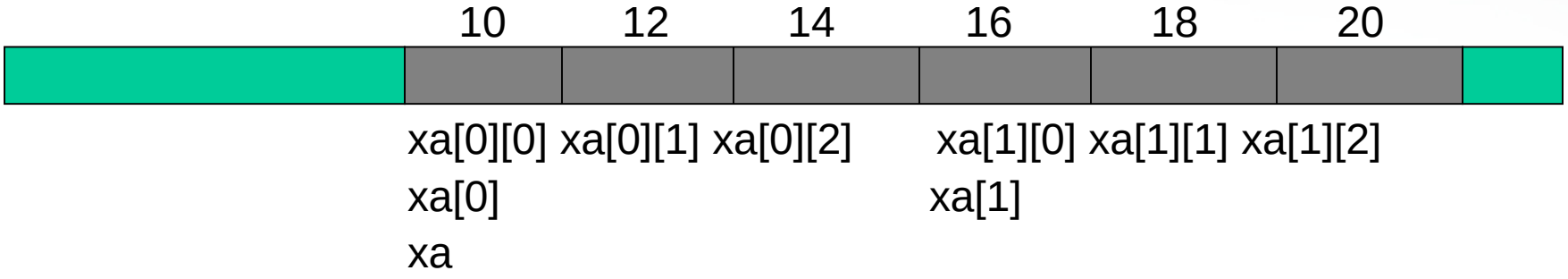
Rôzne spôsoby definície dvojrozmerných polí

1. statické dvojrozmerné pole
2. pole ukazovateľov
3. ukazovateľ na pole
4. ukazovateľ na ukazovateľ

Statické dvojrozmerné pole

```
int xa[2][3];
```

- pole **xa**:
  - alokované pri preklade
  - súvislý blok 6 prvkov
  - uložené po riadkoch
  - konštantný ukazovateľ



Pole ukazovateľov

```
int *xb[2];
```

- pole **xb**:
  - jednorozmerné pole dvoch ukazovateľov na **int**
  - ukazovatele sa využijú na riadky poľa, pre ktoré musíme alokovať pamäť

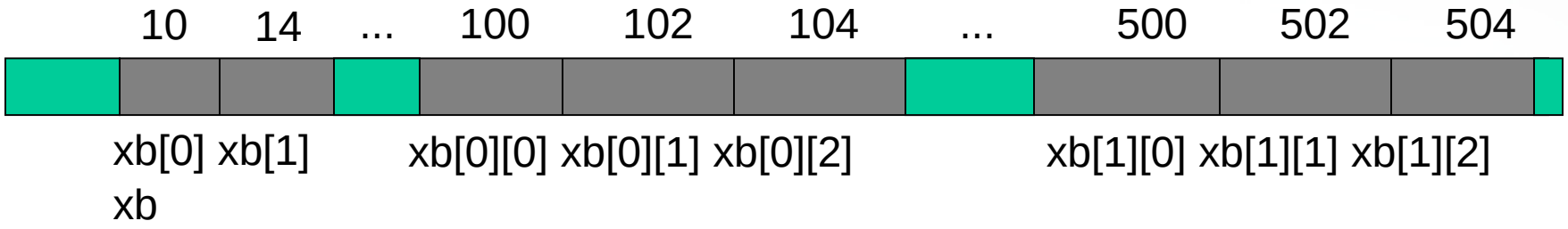
```
xb[0] = (int *) malloc(3* sizeof(int));
xb[1] = (int *) malloc(3* sizeof(int));
```

potom sa dá použiť: `xb[0][2] = 5;`

Pole ukazovateľov

```
int *xb[2];
```

- pole **xb**:
  - jednotlivé riadky nemusia nasledovať v pamäti bezprostredne za sebou
  - ak u statického poľa **`int xa[2][3]`** priradíme **`xa[0][3] = 5`**, potom sa hodnota priradí **`xa[1][0]`**, u poľa **xb** to nemusí platiť



Ukazovateľ na pole

```
int (*xc)[3];
```

- pole **xc**:
  - **xc** je ukazovateľ na pole troch **int**-ov
  - ak alokujeme dostatok pamäte - ako dvojrozmerné pole

```
xc = (int *) malloc(2 * 3 * sizeof(int));
```

  - **xc** ukazuje na pole 6 prvkov združených po troch
  - obdoba statického poľa

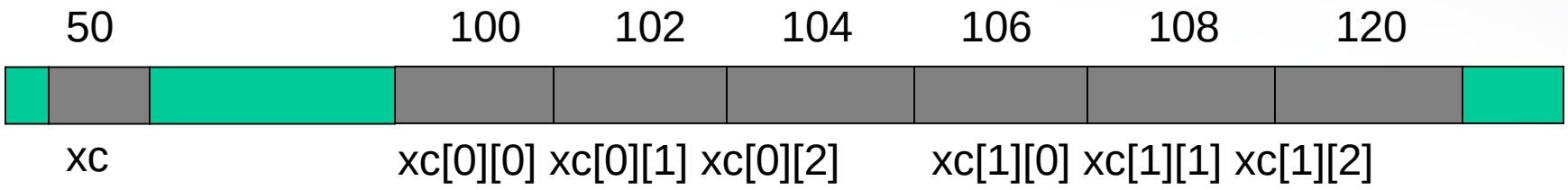
dá sa použiť:

```
xb[0][2] = 5;
```

Ukazovateľ na pole

```
int (*xc)[3];
```

- pole **xc**:
  - jednotlivé riadky nasledujú v pamäti bezprostredne za sebou



Ukazovateľ na ukazovateľ

```
int **xd;
```

- pole **xd**:
  - ukazovateľ na ukazovateľ, preto

(1) alokujeme ukazovatele na riadky

```
xd = (int **) malloc( sizeof(int *));
```

(2) alokujeme jednotlivé riadky

```
xd[0] = (int *) malloc(3* sizeof(int));
xd[1] = (int *) malloc(3* sizeof(int));
```

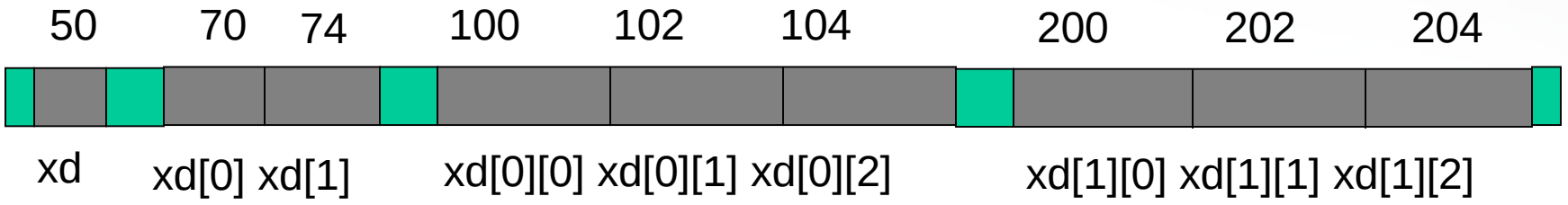
potom sa dá použiť:

```
xd[0][2] = 5;
```

Ukazovateľ na ukazovateľ

```
int **xd;
```

- pole **xd**:
  - **xd** je ukazovateľ na ukazovateľ na typ **int**
  - ***xd** je ukazovateľ na typ **int**
  - ****xd** je prvok typu **int**

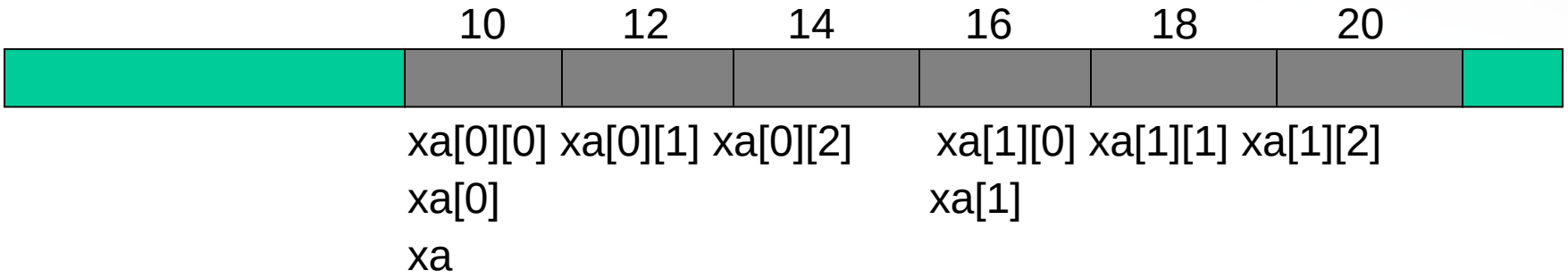


Výhody a nevýhody spôsobov vytvárania polí: typ poľa

- Definícia **`xa`** (**`int xa[2][3]`**) predstavuje statické pole
- Definícia **`xb`** (**`int *xb[2]`**), **`xc`** (**`int (*xc)[3]`**) a **`xd`** (**`int **xd`**) predstavujú po alokácii dynamické polia

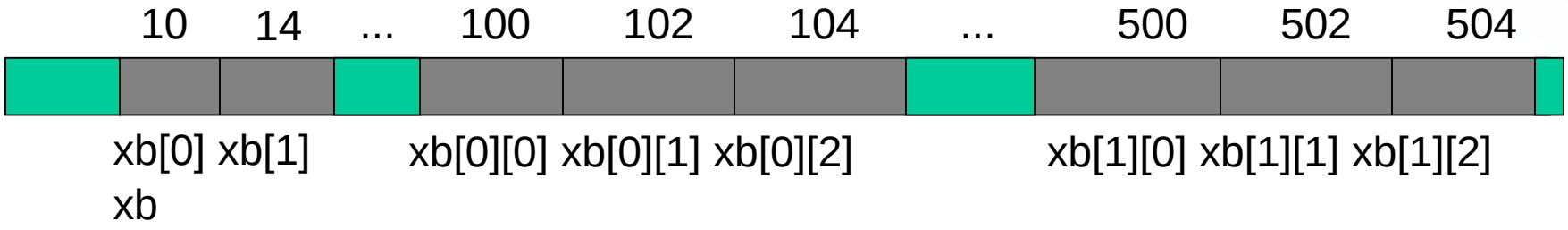
Výhody a nevýhody spôsobov vytvárania polí: pamäť. nároky

- **`xa`** (**`int xa[2][3]`**): pamäťovo najvýhodnejšia



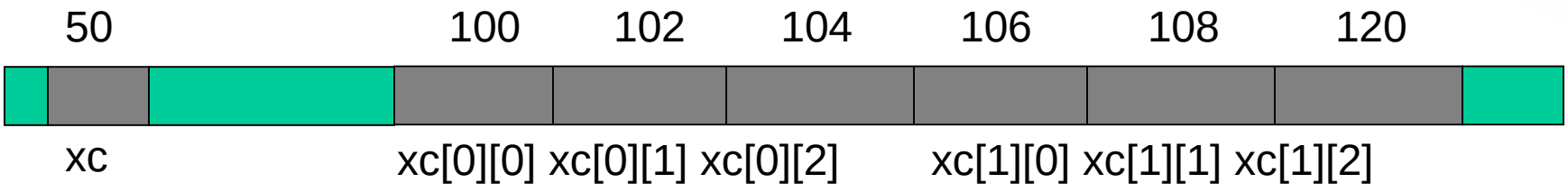
Výhody a nevýhody spôsobov vytvárania polí: pamäť. nároky

- **`xa`** (**`int xa[2][3]`**): pamäťovo najvýhodnejšia
- **`xb`** (**`int *xb[2]`**): naviac pamäť pre 2 ukazovatele (počet riadkov **`xb[0]`**, **`xb[1]`**)



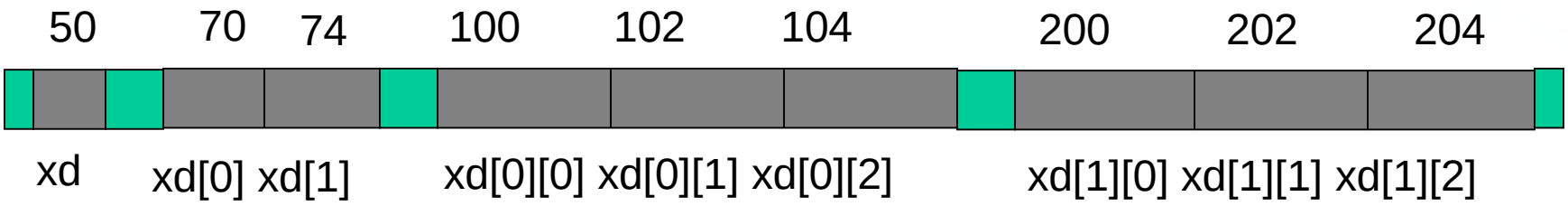
Výhody a nevýhody spôsobov vytvárania polí: pamäť. nároky

- **`xa`** (**`int xa[2][3]`**): pamäťovo najvýhodnejšia
- **`xb`** (**`int *xb[2]`**): naviac pamäť pre 2 ukazovatele (počet riadkov **`xb[0]`**, **`xb[1]`**)
- **`xc`** (**`int (*xc)[3]`**): naviac pamäť pre 1 ukazovateľ na typ **`int`**: **`xc`**



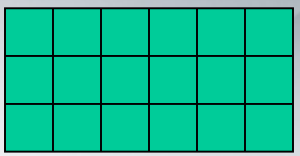
Výhody a nevýhody spôsobov vytvárania polí: pamäť. nároky

- **`xa`** (**`int xa[2][3]`**): pamäťovo najvýhodnejšia
- **`xb`** (**`int *xb[2]`**): naviac pamäť pre 2 ukazovatele (počet riadkov **`xb[0]`**, **`xb[1]`**)
- **`xc`** (**`int (*xc)[3]`**): naviac pamäť pre 1 ukazovateľ na typ **`int`**: **`xc`**
- **`xd`** (**`int **xd`**): naviac 3 ukazovatele pre **`xd`** a 2 ukazovatele na riadky (**`xd[0], xd[1]`**)

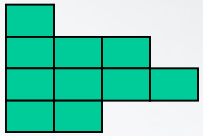


Výhody a nevýhody spôsobov vytvárania polí: charakter

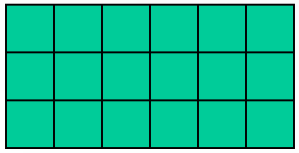
- **`xa`** (**`int xa[2][3]`**): pravoúhle pole



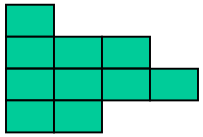
- **`xb`** (**`int *xb[2]`**): "zubaté" pole



- **`xc`** (**`int (*xc)[3]`**): pravoúhle pole



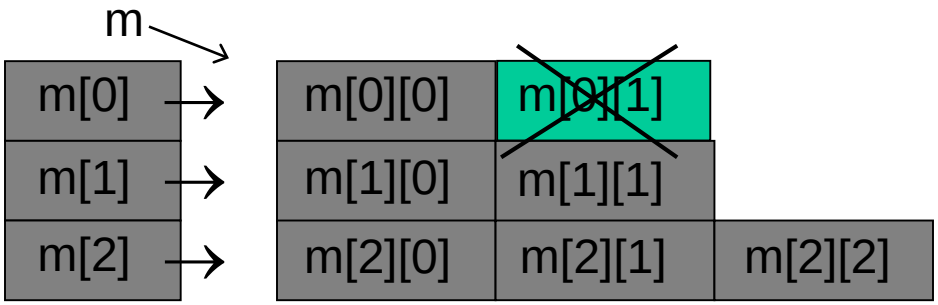
- **`xd`** (**`int **xd`**): "zubaté" pole



Príklad: "zubaté" pole

- dvojrozmerné pole s rôznou dĺžkou riadkov - časť matice pod diagonálou (vrátane) - **`int *m[3]`**

```
int *m[3], i;
for(i = 0; i < 3; i++)
    m[i] = (int *) malloc((i+1) * sizeof(int));
```



Príklad: alokovanie dynamického pravoúhleho poľa

```
int **create(int riadky, int stlpce)
{
  int **p, i;

  p = (int **) malloc(riadky * sizeof(int *));
  for(i = 0; i < riadky; i++)
    p[i] = (int *) malloc(stlpce * sizeof(int));

  return p;
}
```

```
int **a, **b;
a = create(3, 5);
b = create(10, 20);
```

príklad volania funkcie:

Prístup k prvkom statického dvojrozmerného poľa

```
int **x, y[5][6];



x = (int **) y;
```

definícia ukazovateľa na ukazovateľ na **int** (**x**) a statického poľa (**y**)

ukazovateľ **x** sa nedá používať na prístup do poľa **y**, pretože **x** nie je definovaná na prístup do dvojrozmerného poľa - nemá informáciu o veľkosti riadkov (inak je potrebené riadiť sa podľa počtu riadkov a stĺpcov)

Dvojrozmerné pole ako parameter funkcie

- ako jednorozmerné pole
- odlišnosť:
  - prvá dimenzia - prázdna []
  - druhá dimenzia musí byť uvedená, napr. [10]
- preto
  - je potrebné preniesť do funkcie aj počet riadkov
  - skutočný parameter: len pravouhlé polia (**xa**, **xc**)

pre: **`double x[5][6];`**

`double x[][6]` alebo **`double (*x)[6]`**

~~**`double *x[6]`**~~

Dvojrozmerné pole ako parameter funkcie: príklad

funkcia vráti maximum z prvkov dvojrozmerného poľa

```
double maximum(double pole [][4], int riadky)
{
    double pom = pole[0][0];
    int i, j;

    for (i = 0; i < riadky; i++) {
        for (j = 0; j < 4; j++)
            if (pole[i][j] > pom)
                pom = pole[i][j];
    }
    return (pom);
 }
```

Inicializácia polí

- najčastejšie u reťazcov (aj pre iné polia)

```
double f[3] = {1.5, 3.0, 7.6 };
```

```
double f[] = {1.5, 3.0, 7.6 };
```

ak nie je uvedený počet prvkov, určí sa podľa počtu hodnôt.

```
double f[3] = {1.5, 3.0 };
```

ak je hodnôt menej, doplní sa hodnotami 0.0

```
double f[3] = {1.5, 3.0, 7.6, 3.8 };
```

ak je hodnôt viac → chyba

Inicializácia dvojrozmerných polí

počet stĺpcov musí byť uvedený

```
double f[][2] = {
                  { 1.5, 3.0 },
                  { 7.6, 3.8 }
                };
```

počet riadkov môže byť uvedený

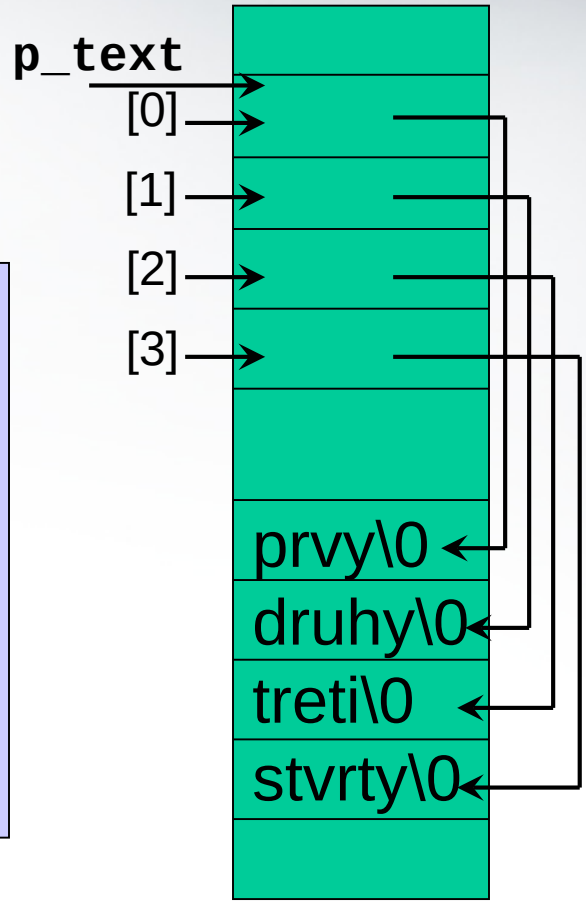
Pole reťazcov

- asi najčastejšie využívané pole s rôznou dĺžkou riadkov

len reťazec **`p_text[2]`** je alokovaný dynamicky, ostatné sú statické

```
char *p_text[4];

p_text[0] = "prvy";
p_text[1] = "druhy";
p_text[2] = (char *) malloc(6);
strcpy(p_text[2], "treti");
p_text[3] = "stvrty";
```



Pole reťazcov

```
char *p_text[4], c, *p;
...
c = p_text[0][0];

p = &p_text[0][0];
while (*p != '\0')
    putchar(*p++);
printf("%s \n", p_text[1]);

puts(p_text[2]);
```

prístup k jednotlivým prvkom reťazca

vytlačenie reťazca po znakoch

vytlačenie reťazca pomocou **`printf()`**

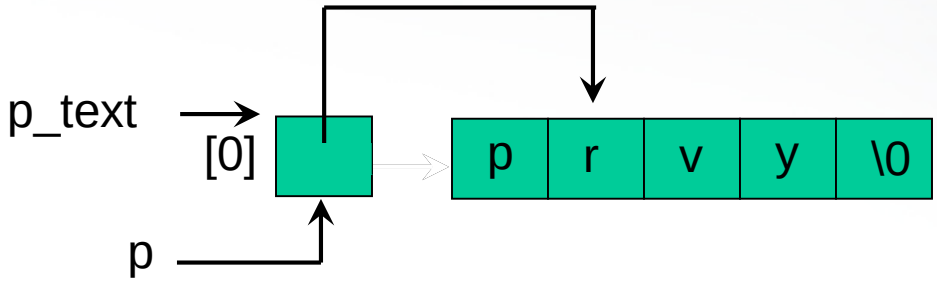
vytlačenie reťazca pomocou **`puts()`**

Pole reťazcov

```
char *p_text[4], **p;
...
p = p_text;
puts(++*p);
```

- **p** ukazuja na **p_text**,
- ***p** ukazuje na **p_text[0]**
- príkaz **++*p** zväčší hodnotu na tej adrese o 1, teda zväčší **p_text[0]**

- vytlačí sa **"rvy"** pretože ***p** ukazuje na nultý prvok poľa **p_text**
- **p_text[0]** potom ukazuje na **"rvy"** a táto zmena je trvalá



Pole reťazcov

```
char *p_text[4], **p;
...
p = p_text;
puts(*++p);
```

vytlačí sa **"druhy"** pretože sme najprv zvýšili **p** o 1 (posunuli sme ho na druhý riadok a potom vytlačili režazec, kam ukazuje **p**)

```
char *p_text[4], **p;
...
p = p_text;
for (i = 0; i < 4; i++)
    puts(*p++);
```

++ má väčšiu prioritu ako *****, riadok sa najprv vypíše a ukazovateľ **p** sa posunie na druhý riadok...

Parametre funkcie `main()`

```
int main()
```

- návratová hodnota: vracia správu operačnému sytému
- argumenty:
  - **`int argc`**: počet reťazcov vstupného poľa
  - **`char *argv[]`**: vstupné pole

Parametre funkcie `main()`

```
int main(int argc, char *argv[])
```

program nazveme napr. **`test`**,
volanie: **`test parameter1 parameter2`**

→ **`argc: 3`**
`argv[0]: test`
`argv[1]: parameter1`
`argv[2]: parameter2`

pozn.: názov je v **`argv[0]`**

volanie: **`test "ahoj nazdar" cau`** → **`argc`**: 3

Parametre funkcie `main()`: príklad

- ak je argument **`"-h"`**, program vypíše **`"help"`**, inak **`"program"`**

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  if(argc == 2 && !strcmp(argv[1], "-h"))
    printf("help\n");
  else
    printf("program\n");
  return 0;
}
```

Príklad: práca s maticami

- načítanie matice zo súboru
  - názov súboru ako argument programu
  - 3 matice: **`m1, m2, m3`**
- menu:
  - výpis matíc
  - sčítanie matíc: **`m3 = m1 + m2`**
  - násobenie matíc: **`m3 = m1 * m2`**
  - výmena matíc: **`m1` ↔ `m2`**

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

#define N 3
#define SUBOR "matice.txt"

void nacitaj(int m1[N][N], int m2[N][N], int m3[N][N],
   char meno[]);
void vypis(int m1[N][N], int m2[N][N], int m3[N][N]);
void scitaj(int m1[N][N], int m2[N][N], int m3[N][N]);
void nasob(int m1[N][N], int m2[N][N], int m3[N][N]);
void vymen(int m1[N][N], int m2[N][N]);
int nacitaj_pom(int m[N][N], FILE *f);
```

```
int main(int argc, char *argv[])
{
   int c, m1[N][N], m2[N][N], m3[N][N];
   char f_meno[50];

   nacitaj(m1, m2, m3, argc == 2 ? argv[1] : SUBOR);

   do {
   printf("\n*** MATICE *** \nv: vypis \ns: scitanie \n")
   printf("n: nasobenie \nm: vynena \nk: koniec\n");
              while ((c = tolower(getchar())) == '\n');

              switch (c) {
              case 'v': vypis(m1, m2, m3); break;
              case 's': scitaj(m1, m2, m3); break;
              case 'n': nasob(m1, m2, m3); break;
              case 'm': vymen(m1, m2); break;
              }
       } while (c != 'k');

       return 0;
}
```

```
void nacitaj(ir
   char meno[])
{
   FILE *f;

   if ((f = fop
      printf("S
      exit(1);
   }

   if(nacitaj_pom(m1, f) || nacitaj_pom(m2, f) ||
      nacitaj_pom(m3, f)) {
      printf("Nepodarilo sa nacitat matice.\n");
       exit(1);
   }

   if (fclose(f) == EOF)
      printf("Subor sa nepodarilo zatvorit.\n");
}
```

```
int nacitaj_pom(int m[N][N], FILE *f)
{
   int i, j;

   for (i=0; i<N; i++)
      for (j=0; j<N; j++)
         if (fscanf(f, "%d", &m[i][j]) != 1)
            return 1;
   return 0;
}
```

```
void vypis(int m1[N][N], int m2[N][N], int m3[N][N])
{
   int i, j, k, (*m)[N];

   for (k=0; k<3; k++) {
      switch (k) {
      case 0: m = m1; break;
      case 1: m = m2; break;
      case 2: m =  m3; break;
      }
   printf("Matica c.%d:\n", k+1);

      for (i=0; i<N; i++) {
         for (j=0; j<N; j++)
            printf("%d ", m[i][j]);
         putchar('\n');
       }
       printf("\n");
   }
}
```

```
void scitaj(int m1[N][N], int m2[N][N], int m3[N][N])
{
   int i, j;

   for (i=0; i<N; i++)
      for (j=0; j<N; j++)
         m3[i][j] = m1[i][j] + m2[i][j];

   printf("Sucet matic c.1 a 2 je v matici c.3\n");
}
```

```
void nasob(int m1[N][N], int m2[N][N], int m3[N][N])
{
   int i, j, k;

   for (i=0; i<N; i++)
      for (j=0; j<N; j++)
         m3[i][j] = 0;

   for (i=0; i<N; i++)
      for (j=0; j<N; j++)
         for (k=0; k<N; k++)
            m3[i][j] += m1[i][k] * m2[k][j];

   printf("Sucin matic c.1 a 2 je v matici c.3\n");
}
```

```
void vymen(int m1[N][N], int m2[N][N])
{
   int i, j, m[N][N];

   for (i=0; i<N; i++)
      for (j=0; j<N; j++)
         m[i][j] = m1[i][j];

   for (i=0; i<N; i++)
      for (j=0; j<N; j++)
         m1[i][j] = m2[i][j];

   for (i=0; i<N; i++)
      for (j=0; j<N; j++)
         m2[i][j] = m[i][j];

   printf("Matice c.1 a 2 su vymenene.\n");
}
```