Systémové programovanie / ÚINF/SPR1a

Róbert Novotný
robert.novotny@upjs.sk

7. 12. 2010

Ako na viacrozmerné polia?

- staticky alokované polia nie sú problém
- deklarácia staticky alokovaného poľa:

```
int matica[5][3];
```

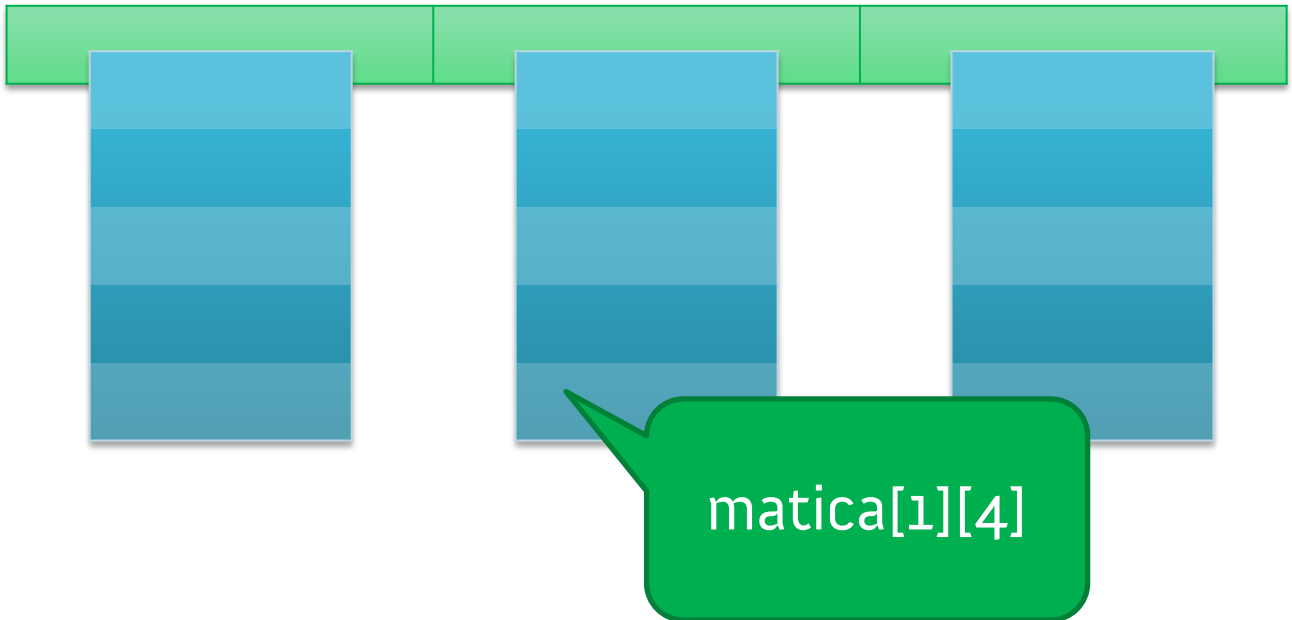
- prístup k prvkom: klasicky cez []
- pozor na indexovanie od nuly!

Chápanie viacrozmerných polí:
viacrozmerné pole je pole polí!

Viacrozmerné pole je pole polí


```
int matica[3][5];
```

- pole 3 prvkov
  - každý prvok je pole piatich prvkov



Viacrozmerné pole možno rovno nainicializovať!


```
int matica[][3] = {{1, 2, 5}, {3, 4, 10}};
```

- pozor! musíme uviesť počet prvkov vo vnorenom poli!
- v C možno staticky alokovať len obdĺžnikové polia

```
int matica[][] = {{1, 2, 5}, {3, 4, 10}};
```

Array type has incomplete element type!

Viacrozmerné pole možno rovno nainicializovať!


```
int matica[][3] = {{1, 2, 5}, {3, 4, 10}};
```

- pozor na dátový typ vo funkciách

```
int sucet(int matica[][3]) {
  /* .. */
}
```

- dátový typ parametra s viacrozmerným poľom musí obsahovať rozmer!

nezabudnúť na rozmer!

Polia sú pointery!

- Polia sú pointery!
- Pole je ekvivalentné pointeru na prvý prvok...
- Viacrozmerné pole je pole polí

So logically!

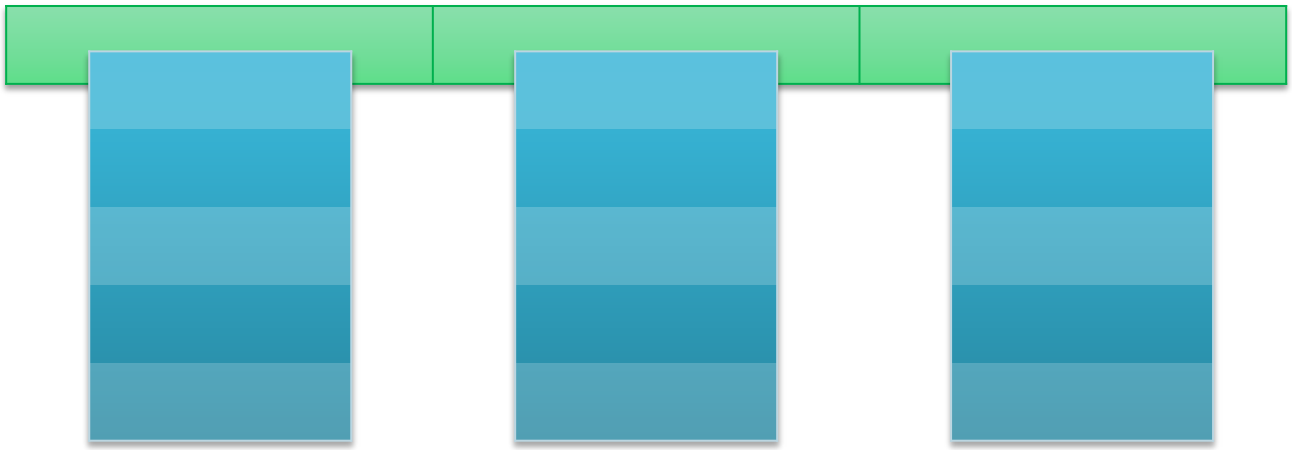
- viacrozmerné pole je pole pointerov na prvé prvky!
- viacrozmerné pole je pointer na pointre!

Viacrozmerné pole je pointer na pointer!


```
int main(int argc, char** argv) {
  /* .. */
}
```

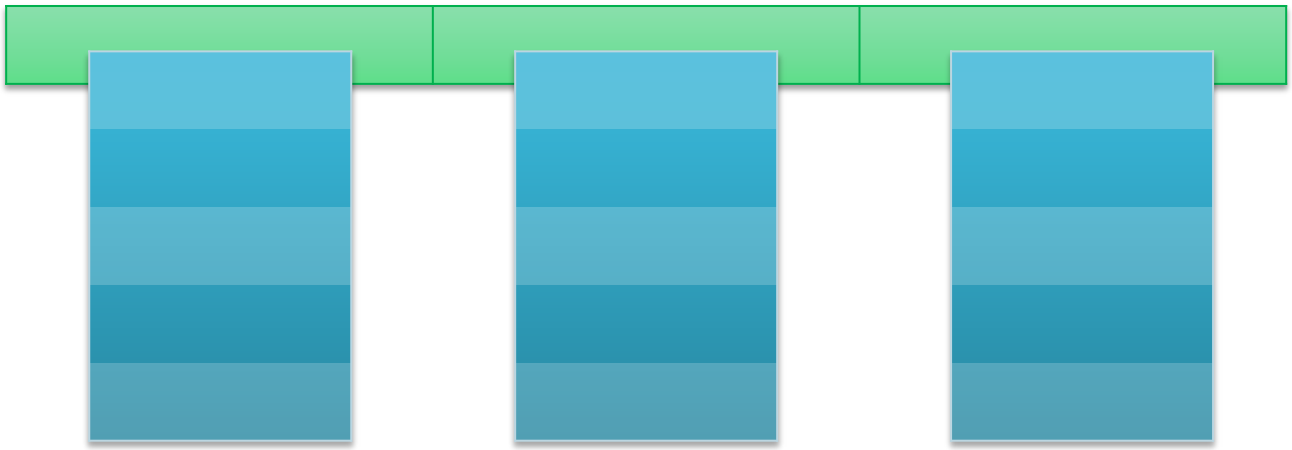
- **char ****:
  - pointer na pointer na char
- v skutočnosti dvojrozmerné pole
  - pole „reťazcov"
  - pole polí znakov

Dynamická alokácia je teraz easy!



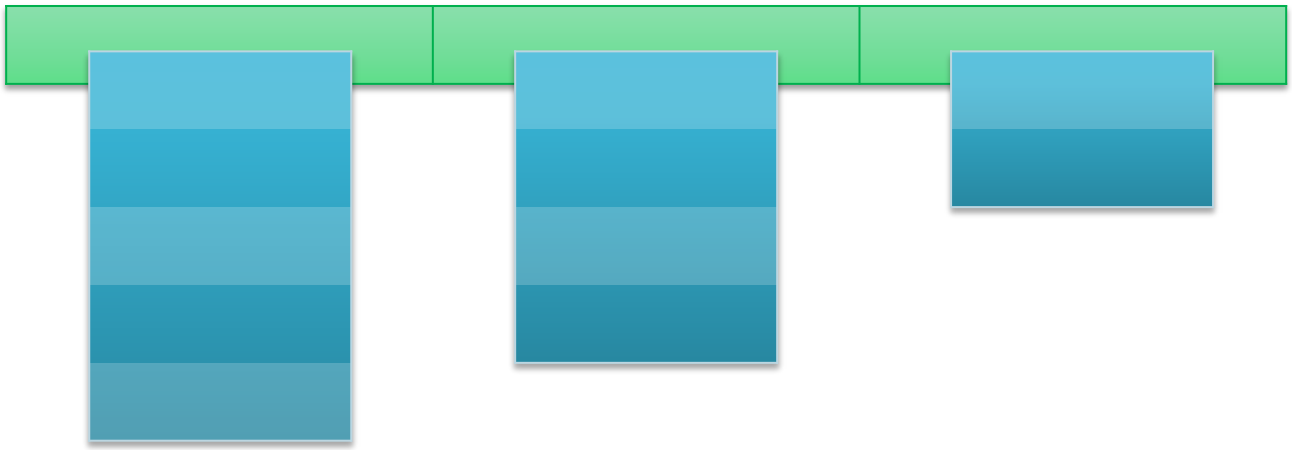
- najprv alokujeme zelené pole
  - pole troch prvkov typu „pointer na **int**"
- v každom prvku zeleného poľa dynamicky alokujeme pole piatich **intov**

Dynamická alokácia je teraz easy!



```
int i; int POCET_STLPCOV = 3; int POCET_RIADKOV = 5;
int * * pole;
pole = calloc(POCET_STLPCOV, sizeof(int));
for (i = 0; i < POCET_STLPCOV; i++) {
  pole[i] = calloc(POCET_RIADKOV, sizeof(int));
}
```

Alternatívny spôsob chápania polí



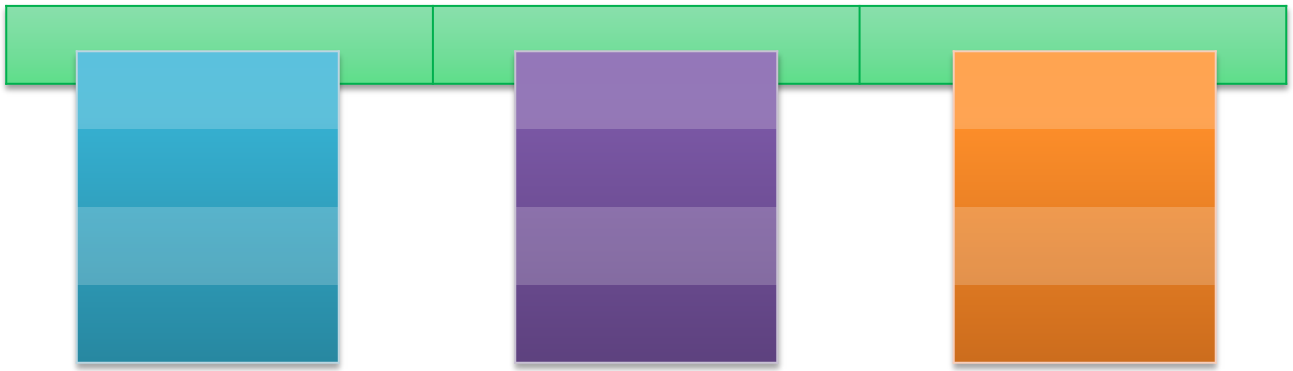
```
int * pole[3];
pole[0] = calloc(5, sizeof(int));
pole[1] = calloc(4, sizeof(int));
pole[2] = calloc(2, sizeof(int));
```

pole troch pointerov na int

Môžeme vyrábať „zubaté polia"

Poznámky k viacrozmerným poliam

- viacrozmerné polia, ktoré sú alokované **staticky**, sú v pamäti uložené súvisle



```
pole[0][5] == pole[1][0]
```

hack využívajúci štruktúru

Poznámky k viacrozmerným poliam

- pozor na zhodu dátových typov!

```
int matica[][2] = { { 4, 5 }, { 2, 8 } }
```

- hlavička metódy, ktorá dokáže prijať maticu:

```
void spracuj(int matica[][2], int pocet_riadkov);
```

```
void spracuj(int (* matica)[2], int pocet_riadkov);
```

Poznámky k viacrozmerným poliam


```
int matica[][2] = { { 4, 5 }, { 2, 8 } }
```

- uvedený výraz sa prevedie na výraz typu „pointer na pole dvoch prvkov typu int"

```
int (* matica)[2]
```

pointer na pole dvoch intov

- zátvorky sú **významné**!

```
int * matica[2]
```

dvojprvkové pole pointerov na int!

Poznámky k viacrozmerným poliam

- ak chceme úplne dynamickú funkciu, zvoľme stratégiu „pointer na pointer na [typ]"

```
void spracuj(int * * matica, int riadky, int stlpce)
```

```
int** matica = calloc(5, sizeof(int));
matica[0] = calloc(3, sizeof(int));
matica[1] = calloc(3, sizeof(int));
matica[0][1] = ...
matica[0][2] = ...
spracuj(matica, 5, 3);
```

Veselé funkcie pre prácu s poliami

- int **memcmp**(void **s1*, void **s2*, size_t *n*);
  - porovná ,,lexikograficky" prvých **n** bajtov blokov v pamäti a vráti nulu v prípade zhody
- void ***memcpy**(void **s1*, const void **s2*, size_t *n*);
  - temer ako **strcpy()**
  - kopíruje n bajtov blok z s2 do s1
  - ak sa bloky prekrývajú, nedefinované správanie

Veselé funkcie pre prácu s poliami

- void ***memmove**(void *s1, const void *s2, size_t n);
  - to isté ako memcpy()
  - interne používa pomocné pole, ktoré ošetrí prekryvy
- void ***memset**(void *s, int c, size_t n);
  - nastaví hodnotu prvých n prvkov na požadovaný znak
  - používa sa pri nulovaní polí

Funkcionálne programovanie v C

- **funkcionálne programovanie** je paradigma, v ktorej kód a dáta splývajú
- množstvo problémov sa rieši vo funkcionálnom programovaní elegantnejšie než v procedurálnom / objektovom
- príklad: operácie nad zoznamami
  - vráť zoznam s dvojnásobkami prvkov
  - vráť zoznam s mocninami prvkov
  - líšia sa len v **operácii**, ktorá sa udeje nad každým **prvkom**

Funkcionálne programovanie

- v Jave: interfejsy

```
public List<Integer> each(List<Integer> list, Operation o)
{
  List<Integer> novýZoznam = new LinkedList<Integer>();
  for(Integer element : list) {
    novýZoznam.add(o.invoke(element));
  }
}
```

```
public interface Operation {
  public void invoke(Integer i);
}
```

Funkcionálne programovanie


```
public interface Operation {
  public void invoke(Integer i);
}
```

- ak chceme zdvojnásobovať prvky, vyrobíme triedu ZdvojnásobOperation implements Operation
- ak chceme umocňovať, vyrobíme UmocniOperation implements Operation
- operácia hovorí **čo** sa má vykonať s každým prvkom
  - objekt je potom kus kódu, ktorý možno pohadzovať
  - áno, closure ;-)

Aj C vie byť funkcionálne

- tá istá filozofia je aj v C
- namiesto interfejsov... **pointery na funkcie**!

```
void each(int * zoznam, int velkost_zoznamu,
          void (* funkcia)(int)) {
  /* ... */
}
```

pointer na funkciu!

```
public List<Integer> each(List<Integer> list, Operation o)
{
  /* ... */
}
```

Pointer na funkciu


```
void (* funkcia)(int)
```

- áno, toto je dátový typ
- čítame zvnútra von:
  - **(* funkcia)**: premenná **funkcia** je pointer na
  - **(int)**: funkciu, ktorá berie ako parameter int
  - **void**: a vracia void

Použitie pointrov na funkcie


```
void zdvojnasob(int prvok) {
  return prvok * 2;
}
/* .. */
int pole[] = {2, 3, 5};
each(pole, zdvojnasob);
```

zadáme dáta (pole) a kód (funkciu), ktorá ich spracuje

- do funkcie **each** potom dať odkaz na ľubovoľnú funkciu v kóde
- pozor na zhodu hlavičiek

Kde sa to používa?

- Pointre na funkcie sa používajú kade tade
- typický príklad: **callback**
  - spätné volanie
  - zavoláme funkciu, ktorá nás chce priebežne informovať o výsledkoch
- Win32 API: **EnumWindows**
  - odovzdáme handle okna (nedôležité)
  - odovzdáme **pointer na našu funkciu**, ktorá vráti true, ak vyhľadávanie má skončiť
  - a odovzdáme pomocný parameter (nedôležité)