



PROGRAMOVACIE JAZYKY PRE VSTAVANÉ SYSTÉMY

Dynamická správa pamäte, viacrozmerné polia

OTÁZKY Z MINULEJ PREDNÁŠKY

- Na čo slúžia používateľom definované typy?
- Aký je rozdiel medzi štruktúrou a union-om?
- Ako je implementovaný vymenovaný typ?
- Na čo slúžia operátory „.“ a „->“ pri práci so štruktúrami?
- Aká je všeobecná schéma práce so súborom?



GENERICKÝ SMERNÍK

- Pod generickým smerníkom sa rozumie typ **void***.
- Vlastnosti:
 - môže uchovávať adresu objektu ľubovoľného dátového typu
 - nemôže byť dereferencovaný
 - každý smerník na objekt (napr. `char*`, `int*`, `float*`, ...) môže byť **implicitne** konvertovaný na `void*` a naopak (v C++ to neplatí)
 - pre ľubovoľný smerník na objekt **T *pointer** platí:
$$(T *)((void *)pointer) == pointer$$
- Použitie:
 - tvorba generických rozhraní:
 - funkcia prijíma argument ľubovoľného typu (callback funkcia pre `qsort`)
 - funkcia vracia argument ľubovoľného typu (`malloc`)
- Kontrolná otázka: Môžem aplikovať smerníkovú aritmetiku na premennú typu `void*`?



PRÁCA S PAMÄŤOU NA ÚROVNI BAJTOV (1)

- Jazyk C poskytuje niekoľko funkcií, pomocou ktorých je možné pracovať s celými blokmi operačnej pamäte.
- Prototypy funkcií sú v hlavičkovom súbore `<string.h>` (<http://en.cppreference.com/w/c/string/byte>):
 - `void* memcpy(void *dest, const void *src, size_t n);`
 - `void* memmove(void *dest, const void *src, size_t n);`
 - `void* memset(void *ptr, int c, size_t n);`
 - `int memcmp(const void *ptr1, const void *ptr2, size_t n);`
 - `void* memchr(const void *ptr, int c, size_t n);`
- `size_t` – nezáporný celočíselný typ, návratový typ operátora `sizeof`.



PRÁCA S PAMÄŤOU NA ÚROVNI BAJTOV (2)

- `void* memcpy(void *dest, const void *src, size_t n)`
 - kopíruje `n` bajtov z adresy `src` na adresu `dest`;
 - bloky pamäte sa **nesmú** prekrývať;
 - funkcia vracia smerník `dest`.
- `void* memmove(void *dest, const void *src, size_t n)`
 - kopíruje `n` bajtov z adresy `src` na adresu `dest`;
 - bloky pamäte sa **môžu** prekrývať
 - funkcia vracia smerník `dest`.
- Kontrolná otázka: Ako zistím, či sa prekrývajú dva bloky pamäte?



PRÁCA S PAMÄŤOU NA ÚROVNI BAJTOV (3)

- `void* memset(void *ptr, int c, size_t n)`
 - nastaví `n` bajtov od adresy `ptr` na hodnotu znaku `c`;
 - funkcia vracia smerník `dest`.
- `int memcmp(const void *ptr1, const void *ptr2, size_t n)`
 - porovná `n` bajtov v dvoch blokoch pamäte;
 - funkcia vracia hodnotu:
 - `< 0`, ak `ptr1 < ptr2` pre prvý rôzny bajt;
 - `== 0`, ak `ptr1 == ptr2` pre všetky bajty;
 - `> 0`, ak `ptr1 > ptr2` pre prvý rôzny bajt.
- `void* memchr(const void *ptr, int c, size_t n)`
 - hľadá výskyt znaku `c` v poli bajtov dĺžky `n`, začínajúcom na adrese `ptr`;
 - vráti adresu, na ktorej sa našla prvá zhoda, alebo hodnotu `NULL`, ak sa žiadna zhoda nenašla.
 - Ako by som hľadal druhú zhodu?



UKÁŽKA – UNIVERZÁLNÁ METÓDA PRE VÝMENU OBSAHU DVOCH PREMENNÝCH

```
1  #include <stdio.h>
2  #include <string.h>
3
4  void genericSwap(void* a, void* b, size_t size) {
5      char tmp[size];
6      memcpy(tmp, a, size);
7      memcpy(a, b, size);
8      memcpy(b, tmp, size);
9  }
10
11 int main(int argc, char* argv[]) {
12     int iA, iB;
13     printf("Zadaj dve cele cisla:\n");
14     scanf("%d%d", &iA, &iB);
15     printf("Pred vymenou: a = %d, b = %d\n", iA, iB);
16     genericSwap(&iA, &iB, sizeof(iA));
17     printf("Po vymene: a = %d, b = %d\n", iA, iB);
18
19     double dA, dB;
20     printf("Zadaj dve realne cisla:\n");
21     scanf("%lf%lf", &dA, &dB);
22     printf("Pred vymenou: a = %f, b = %f\n", dA, dB);
23     genericSwap(&dA, &dB, sizeof(dA));
24     printf("Po vymene: a = %f, b = %f\n", dA, dB);
25
26     return 0;
27 }
```



DYNAMICKÁ SPRÁVA PAMÄTE (1)

- V mnohých situáciách potrebujeme pamäť alokovať až za behu programu, podľa informácií, ktoré v čase kompilácie nie sú k dispozícií.
- V takom prípade musíme alokovať pamäť z haldy.
- Funkcie pre prácu s dynamickou pamäťou – hlavičkový súbor `<stdlib.h>`

(<http://en.cppreference.com/w/c/memory>):

- `void* malloc(size_t size);`
- `void* calloc(size_t nitems, size_t size);`
- `void* realloc(void *ptr, size_t new_size);`
- `void free(void *ptr);`
- `void* aligned_alloc(size_t alignment, size_t size) (C11).`



DYNAMICKÁ SPRÁVA PAMÄTE (2)

○ void* malloc(size_t size)

- funkcia alokuje **súvislý** blok pamäte o veľkosti „size“ bajtov, pričom vracia adresu prvého bajtu;
- alokované bajty **nie sú** implicitne inicializované;
- v prípade neúspechu vracia hodnotu NULL;
- ak „size == 0“, návratová hodnota je implementačne závislá.

○ void* calloc(size_t nitems, size_t size)

- funkcia alokuje **súvislý** blok pamäte o veľkosti „nitems * size“ bajtov, pričom vracia adresu prvého bajtu;
- alokované bajty sú inicializované na hodnotu 0;
- v prípade neúspechu vracia hodnotu NULL;
- ak „nitems * size == 0“, návratová hodnota je implementačne závislá.



DYNAMICKÁ SPRÁVA PAMÄTE (3)

- `void* realloc(void *ptr, size_t new_size)`
 - funkcia zväčší/zmenší **daný** blok pamäte (`ptr`) na veľkosť „`new_size`“ bajtov, pričom vracia adresu prvého bajtu (v prípade zväčšenia majú nové bajty náhodný obsah);
 - ak sa funkcii nepodarí zväčšiť/zmenšiť blok pamäte (`ptr`), **alokuje** nový **súvislý** blok pamäte o veľkosti „`new_size`“ bajtov, pričom do neho **prekopíruje** obsah zo starého bloku pamäte a ten následne **uvolní**;
 - v prípade neúspechu vracia hodnotu `NULL`, pričom starý blok pamäte (`ptr`) sa neuvoľní;
 - ak „`ptr == NULL`“, funkcia sa správa ako `malloc(new_size)`;
 - ak „`new_size == 0`“, správanie je totožné s volaním `free(ptr)`, avšak návratová hodnota je implementačne závislá.
- `void free(void *ptr)`
 - funkcia uvoľní pamäť alokovanú prostredníctvom vyššie popísaných funkcií alebo funkcie `aligned_alloc` (C11);
 - správanie funkcie je nedefinované, ak:
 - `ptr` je rôzne od toho, čo vrátili funkcie `malloc`, `calloc`, `realloc` alebo `aligned_alloc`;
 - `ptr` ukazuje na pamäť, ktorá už bola uvoľnená volaním `free` alebo `realloc`;
 - ak „`ptr == NULL`“, funkcia nič nespraví.



UKÁŽKA PRÁCE S DYNAMICKOU PAMĚTÍ

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void naplnPole(int n, int pole[]) {
5      for (int i = 0; i < n; i++) {
6          pole[i] = i;
7      }
8  }
9
10 void vypisPole(int n, int pole[]) {
11     for (int i = 0; i < n; i++) {
12         printf("%d ", pole[i]);
13     }
14     printf("\n");
15 }
16
17 int main(int argc, char* argv[]) {
18     int* pole; //pole - "wild pointer"
19
20     pole = calloc(5, sizeof(int));
21     naplnPole(5, pole);
22
23     pole = realloc(pole, sizeof(int[10])); //pri takomto zapise existuju rizika
24     naplnPole(5, pole + 5);
25
26     vypisPole(10, pole);
27     free(pole); //pole - "dangling pointer"
28     pole = NULL; //pole - "NULL pointer"
29
30     return 0;
31 }
```



NAJČASTEJŠIE CHYBY PRI PRÁCI S DYNAMICKOU PAMÄŤOU

- **Prístup na nealokované miesto** – program spravidla spadne alebo si prepíše vlastné dáta.
- **Prístup za alokované miesto** – nemusí sa stať vôbec nič (alokácie po blokoch), program môže spadnúť hneď (neplatná adresa), neskôr (porušené štruktúry voľnej pamäte) alebo si prepísať dáta z inej alokácie.
- **Prístup do uvoľnenej pamäte** – prejav chyby opäť závisí na okolnostiach.
- **Neuvoľnenie pamäte** – dnešné operačné systémy pamäť uvoľnia po ukončení procesu, ale pri alokácii v cykle a dostatočne dlhom behu programu môže nastať nedostatok pamäte.
- **Opakované volanie free** – už uvoľnený ukazovateľ znovu dealokujeme. Môže viesť k pádu programu, ale nemusí sa stať vôbec nič.



3 TYPY POLÍ V JAZYKU C

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char* argv[]) {
5      int pole1[5];
6      int n = 5;
7      int pole2[n];
8      int *pole3 = (int *)malloc(5 * sizeof(int));
9
10     printf("sizeof(pole1) = %u\n", sizeof(pole1));
11     printf("pole1 = %p\n", pole1);
12     printf("&pole1 = %p\n", &pole1);
13     printf("&(*pole1) = %p\n\n", &(*pole1));
14
15     printf("sizeof(pole2) = %u\n", sizeof(pole2));
16     printf("pole2 = %p\n", pole2);
17     printf("&pole2 = %p\n", &pole2);
18     printf("&(*pole2) = %p\n\n", &(*pole2));
19
20     printf("sizeof(pole3) = %u\n", sizeof(pole3));
21     printf("pole3 = %p\n", pole3);
22     printf("&pole3 = %p\n", &pole3);
23     printf("&(*pole3) = %p\n\n", &(*pole3));
24
25     free(pole3);
26     pole3 = NULL;
27     return 0;
28 }
```



3 TYPY POLÍ V JAZYKU C

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char* argv[]) {
5     int pole1[5];
6     int n = 5;
7     int pole2[n];
8     int *pole3 = (int *)malloc(5 * sizeof(int));
9
10    printf("sizeof(pole1) = %u\n", sizeof(pole1));
11    printf("pole1 = %p\n", pole1);
12    printf("&pole1 = %p\n", &pole1);
13    printf("&(*pole1) = %p\n\n", &(*pole1));
14
15    printf("sizeof(pole2) = %u\n", sizeof(pole2));
16    printf("pole2 = %p\n", pole2);
17    printf("&pole2 = %p\n", &pole2);
18    printf("&(*pole2) = %p\n\n", &(*pole2));
19
20    printf("sizeof(pole3) = %u\n", sizeof(pole3));
21    printf("pole3 = %p\n", pole3);
22    printf("&pole3 = %p\n", &pole3);
23    printf("&(*pole3) = %p\n\n", &(*pole3));
24
25    free(pole3);
26    pole3 = NULL;
27    return 0;
28 }
```

```
sizeof(pole1) = 20
pole1 = 0x28cbb0
&pole1 = 0x28cbb0
&(*pole1) = 0x28cbb0

sizeof(pole2) = 20
pole2 = 0x28cb78
&pole2 = 0x28cb78
&(*pole2) = 0x28cb78

sizeof(pole3) = 4
pole3 = 0x80028c80
&pole3 = 0x28cbac
&(*pole3) = 0x80028c80
```

ŠTRUKTÚRA OBSAHUJÚCA POLE NEDEFINOVANEJ VELKOSTI (C99)

- Ak štruktúra obsahuje aspoň jeden pomenovaný člen, jej posledným členom môže byť pole nedefinovanej veľkosti.
- Inicializácia, operátor sizeof a operátor priradenia ignorujú člen, predstavujúci pole nedefinovanej veľkosti.
- Štruktúry s takýmto členom sa nemôžu vyskytnúť ako položky poľa alebo členy iných štruktúr. Môžu sa vyskytnúť len ako posledný člen union-u.



ŠTRUKTÚRA OBSAHUJÚCA POLE NEDEFINOVANEJ VELKOSTI (C99) – UKÁŽKA (1)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct dynArray {
5      int size;
6      int elements[];
7  } DynArray;
8
9  void naplnDynArray(DynArray *dynArray) {
10     for (int i = 0; i < dynArray->size; i++) {
11         dynArray->elements[i] = i;
12     }
13 }
14
15 void vypisDynArray(DynArray *dynArray) {
16     for (int i = 0; i < dynArray->size; i++) {
17         printf("%d ", dynArray->elements[i]);
18     }
19     printf("\n");
20 }
```



ŠTRUKTÚRA OBSAHUJÚCA POLE NEDEFINOVANEJ VELKOSTI (C99) – UKÁŽKA (2)

```
22  □ int main(int argc, char* argv[]) {  
23      DynArray d1 = {0};  
24  
25      printf("Zadaj veľkosť pola:\n");  
26      int n;  
27      scanf("%d", &n);  
28  
29      DynArray *d2 = malloc(sizeof(DynArray) + n * sizeof(int));  
30      (*d2).size = n;  
31      naplnDynArray(d2);  
32  
33      n = n + 10;  
34      DynArray *d3 = malloc(sizeof(DynArray) + n * sizeof(int));  
35      *d3 = *d2;  
36      vypisDynArray(d3);  
37  
38      free(d2);  
39      d2 = NULL;  
40      free(d3);  
41      d3 = NULL;  
42  
43      return 0;  
44  }
```



VIACROZMERNÉ POLIA (1)

- Viacrozmerné pole vytvoríme ako pole, ktorého prvkami sú opäť polia. Napr.:

```
int matica[2][3];
```

- dvojrozmerné pole, ktoré má 2 riadky, pričom každý riadok je poľom 3 premenných typu int
- Pre viacrozmerné pole sa alokuje **súvislý úsek pamäte**, prvý prvok ([0][0]) je umiestnený na najnižšej adrese, posledný prvok ([1][2]) na najvyššej adrese.
- Posledný index sa mení najrýchlejšie – tzv. “row major” ukladanie polí.
- V prípade aplikovania konverzie poľa na smerník je viacrozmerné pole skonvertované **na smerník** ukazujúci na prvý element, t.j. **na riadok s indexom 0**.



VIACROZMERNÉ POLIA (2)

- Predpokladajme:

`int matica[2][3];`

- Potom predchádzajúce tvrdenia majú nasledujúce dôsledky:

- `*matica == matica[0]`
- `**matica == matica[0][0]`
- `matica + 1 == &matica[1]`
- `*(matica + 1) == matica[1]`
- `** (matica + 1) == matica[1][0]`
- `*(*matica + 1) == matica[0][1]`

- `int (*p_1)[3] = matica; //toto je korektné`
- `int *p_2 = *matica; //toto je korektné`

- `int *p_2 = matica; //toto je nekorektné`



VIACROZMERNÉ POLIA – VLA (C99)

- Jazyk C od štandardu C99 podporuje aj viacrozmerné VLA. Napr.:

```
int m = 2;
```

```
int n = 3;
```

```
int matica1[2][n];
```

```
int matica2[m][2];
```

```
int matica3[m][n];
```



VIACROZMERNÉ POLIA – INICIALIZÁCIA

```
int matica[2][3] = { { 0, 1, 2}, {3, 4, 5} };
```

```
int matica[2][3] = { { 0, 1, 2} };
```

```
int matica[2][3] = { [1] = { 0, 1, 2} };
```

```
int matica[2][3] = { [1] = {0,1,2}, [0][2] = 9 };
```

```
int matica[2][3] = { 0 };
```

```
int matica[2][3] = { 0, 1 }; //možný warning
```

```
int matica[][3] = { {0, 1, 2}, {1, [2] = 1} };
```

```
int matica[2][] = { {0, 1, 2}, {1, 1, 1} };
```

```
int matica[2][n] = { 0 };
```

```
int matica[m][3] = { 0 };
```

```
int matica[m][n] = { 0 };
```



VIACROZMERNÉ POLE AKO PARAMETER FUNKCIE

- V dôsledku konverzie poľa na smerník sa do funkcie, ktorá očakáva parameter typu viacrozmerné pole odovzdáva nie pole, ale smerník na prvý prvok poľa t.j. **na riadok s indexom 0**.
- Možné deklarácie funkcie s parametrom typu pole:
 - `int funkcia(int m, int n, int pole2D[m][n]); //C99`
 - `int funkcia(int m, int n, int pole2D[][n]); //C99`
 - **`int funkcia(int m, int pole2D[][3]);`**
 - `int funkcia(int n, int pole[static 2][3]); //C99`
 - ~~`int funkcia(int n, int pole2D[10][]);`~~
 - ~~`int funkcia(int m, int n, int pole2D[][]);`~~



VIACROZMERNÉ POLE AKO PARAMETER FUNKCIE – UKÁŽKA

```
1  #include <stdio.h>
2  #include <string.h>
3
4  void vypisMaticu(int m, int n, int pole2D[m][n]) {
5      for (int i = 0; i < m; i++) {
6          for (int j = 0; j < n; j++) {
7              printf("%d ", pole2D[i][j]);
8          }
9          printf("\n");
10     }
11 }
12
13 void vypisMaticu2(int m, int n, int pole2D[m][n]) {
14     for (int (*p_riadok)[n] = pole2D; p_riadok < pole2D + m; p_riadok++) {
15         for (int* p_element = *p_riadok; p_element < *p_riadok + n; p_element++) {
16             printf("%d ", *p_element);
17         }
18         printf("\n");
19     }
20 }
```



VIACROZMERNÉ POLIA – KONTROLNÉ OTÁZKY

- Majme nasledujúce pole:

```
int pole[2][3][4];
```

- Určte, aké hodnoty budú mať nasledujúce výrazy:

```
sizeof(pole)
```

```
sizeof(pole[0])
```

```
sizeof(*pole)
```

```
sizeof(pole[0][0])
```

- Určte, ktoré tvrdenia sú pravdivé:

```
*pole == pole[0]
```

```
***pole == pole[0][0][0]
```

```
*( (*pole + 1) + 1) == pole[1][1][0]
```

```
*( (*pole + 1) + 1) == pole[0][1][1]
```

```
*( ** (pole + 1) + 1) == pole[1][0][1]
```

- Akými spôsobmi sa dá toto pole inicializovať?
- Ako by ste napísali hlavičku funkcie, ktorá prijíma ako parameter toto pole?



VIACROZMERNÉ POLIA – DYNAMICKÁ PAMÄŤ

- Predpokladajme nasledujúce pole:

```
int matica[2][3];
```

- Toto je staticky alokované pole, ktoré má dva riadky a tri stĺpce a môže sa nachádzať na zásobníku alebo v dátovom segmente.
- Aké sú výhody/nevýhody takéhoto typu poľa?
- Ak chceme pole alokovať dynamicky, počas behu programu, máme viaceré možnosti:
 - VLA (C99) – pole alokované dynamicky na zásobníku
 - pole alokované v halde – v prípade n -rozmerného poľa existuje $2^n - 1$ ako takého pole môžeme vytvoriť (ak neuvažujeme s VLA).



DVOJROZMERNÉ POLIA V DYNAMICKEJ PAMÄTI – POLE SMERNÍKOV

- Tento prístup je založený na myšlienke, že môžem definovať premennú, ktorá je poľom smerníkov.
- Vytvorenie poľa:

```
int* pole[2];  
pole[0] = (int *)malloc(3 * sizeof(int));  
pole[1] = (int *)malloc(3 * sizeof(int));
```
- Práca s poľom:

```
pole[0][1] = 10;
```
- Uvoľnenie:

```
free(pole[0]);  
pole[0] = NULL;  
free(pole[1]);  
pole[1] = NULL;
```
- Takto vytvorené pole môže mať v každom riadku iný počet prvkov (tzv. zubaté polia).
- Ako a kde je toto pole uložené v pamäti?



DVOJROZMERNÉ POLIA V DYNAMICKEJ PAMÄTI – SMERNÍK NA POLE (1)

- Tento prístup je založený na myšlienke, že smerník môže uchovávať adresu celého poľa (a nie len jedného prvku).
- So smerníkom ukazujúcim na pole sa pracuje nasledovne:

```
int pole1[10];  
int pole2[5];  
int (*pPoleA)[10];  
int (*pPoleB)[]; //smerník na pole nešpecifikovanej veľkosti (je to neúplný typ)
```

```
pPoleA = &pole2;  
pPoleA = &pole1;  
pPoleB = &pole1;  
pPoleB = &pole2;
```

```
*pPoleA == pPoleA[0];
```

- Po vykonaní predchádzajúcich príkazov platí:

```
*pPoleA == pPoleA[0] == pole1 == &pole1[0];  
**pPoleA == (*pPoleA)[0] == *(pPoleA[0]) == pPoleA[0][0] == pole1[0];  
sizeof(*pPoleA) == sizeof(pole1);  
sizeof(**pPoleA) == sizeof(*pole1);
```

```
*pPoleB == pPoleB[0] == pole2 == &pole2[0];  
**pPoleB == (*pPoleB)[0] == *(pPoleB[0]) == pPoleB[0][0] == pole2[0];  
sizeof(*pPoleB) == sizeof(pole2);  
sizeof(**pPoleB) == sizeof(*pole2);
```

- Prečiarknuté výrazy vedú k chybe pri kompilácii. **Prečo?**



DVOJROZMERNÉ POLIA V DYNAMICKEJ PAMÄTI – SMERNÍK NA POLE (2)

- Vytvorenie poľa:

```
int (*pole)[3];  
pole = (int(*)[3]) malloc(2 * 3 * sizeof(int));
```

- Práca s poľom:

```
pole[0][1] = 10;
```

- Uvoľnenie:

```
free(pole);  
pole = NULL;
```

- Ako a kde je toto pole uložené v pamäti?



DVOJROZMERNÉ POLIA V DYNAMICKEJ PAMÄTI – SMERNÍK NA SMERNÍK

- Tento prístup je založený na myšlienke, že premenná typu smerník môže uchovávať adresu premennej, ktorá je tiež smerník.
- Vytvorenie poľa:

```
int **pole;  
pole = (int **) malloc(2 * sizeof(int *));  
pole[0] = (int *) malloc(3 * sizeof(int));  
pole[1] = (int *) malloc(3 * sizeof(int));
```
- Práca s poľom:

```
pole[0][1] = 10;
```
- Uvoľnenie (**v takomto poradí**):

```
free(pole[0]);  
free(pole[1]);  
free(pole);  
pole = NULL;
```
- Najvšeobecnejší prístup, máme možnosť meniť oba rozmery poľa počas chodu programu.
- Ako a kde je toto pole uložené v pamäti?



DVOJROZMERNÉ POLIA AKO PARAMETER FUNKCIE

○ 1. možnosť:

```
void funkcia(int pole[][3]);
```

```
void funkcia(int (* pole)[3]);
```

- skutočným parametrom môže byť pole alokované staticky alebo pole alokované ako ukazovateľ na jednorozmerné pole:

```
int pole[2][3];
```

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

○ 2. možnosť:

```
void funkcia(int *pole[]);
```

```
void funkcia(int **pole);
```


- skutočným parametrom môže byť pole smerníkov alebo pole alokované ako smerník na smerník:

```
int **pole;
```

```
int* pole[2];
```



TROJROZMERNÉ POLIA

- Premenná korešpondujúca trojrozmernému poľu môže byť definovaná 2^3 spôsobmi:
 - `int pole3Da[3][4][5];`
 - `int* pole3Db[3][4];`
 - `int (*pole3Dc)[4][5];`
 - `int (*pole3Dd[3])[5];`
 - `int** pole3De[3];`
 - `int* (*pole3Df)[4];`
 - `int (**pole3Dg)[5];`
 - `int*** pole3Dh;`
 - Ako by vyzerala alokácia a dealokácia pre jednotlivé premenné?
- 

POLE REŤAZCOV (1)

- Reťazec je v jazyku C implementovaný ako pole znakov ukončených znakom '\0'.
- Pole reťazcov potom zodpovedá dvojrozmernému poľu znakov:
 - `char retazce1[10][20];`
 - `char retazce2[10][20] = { "ahoj", "ano", "nie" };`
 - `char retazce3[][20] = { "ahoj", "ano", "nie" };`
 - ~~`char retazce4[10][] = { "ahoj", "ano", "nie" };`~~
 - ~~`char retazce5[][] = { "ahoj", "ano", "nie" };`~~
- S využitím smerníkov môžeme pole reťazcov definovať nasledovne:
 - `char *retazce6[10];`
 - `char (*retazce7)[20];`
 - `char **retazce8;`



POLE REŤAZCOV (2)

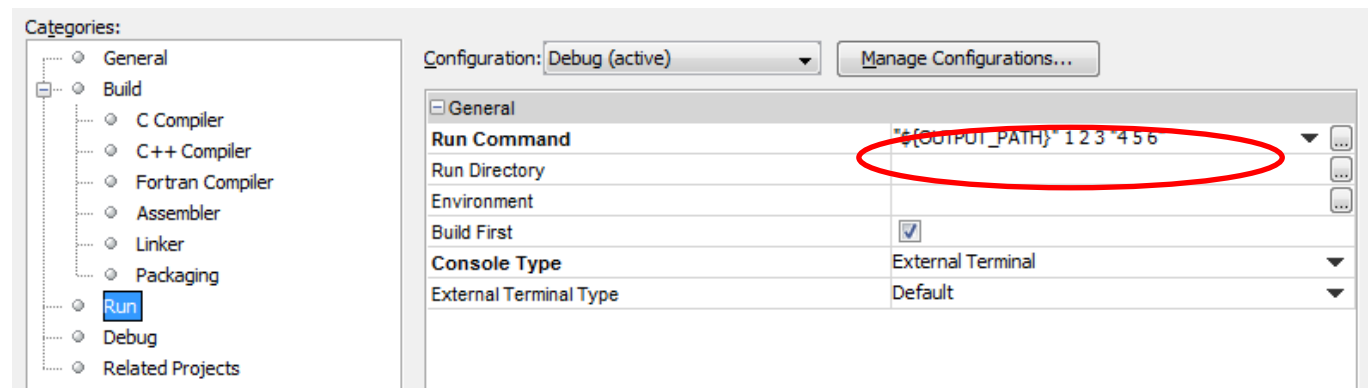
- Určte, v čom sa odlišujú nasledujúce zápisy:
 - `char retazce1[10][20] = { "ahoj", "ano", "nie" };`
 - `char retazce2[][20] = { "ahoj", "ano", "nie" };`
 - `char *retazce3[10] = { "ahoj", "ano", "nie" };`
 - `char *retazce4[] = { "ahoj", "ano", "nie" };`
 - ~~`char (*retazce5)[20] = &"ahoj";`~~
 - `char (*retazce6)[5] = &"ahoj";`
 - `char (*retazce7)[] = &"ahoj";`
 - `char *retazce8 = *retazce1;`
 - `char **retazce9 = retazce3;`



POLE REŤAZCOV (3)

- Typický príklad, kde sa stretávame s poľom reťazcov, je druhý parameter funkcie main.

```
int main (int argc, char** argv) {  
    printf("Program %s bol spustený s nasledujúcimi argumentmi:\n", argv[0]);  
    for (int i = 1; i < argc; i++) {  
        printf("%s\n", argv[i]);  
    }  
  
    return 0;  
}
```



```
Program /cygdrive/d/NetBeans/C_test/dist/Debug/Cygwin_86-Windows/c_test bol spus  
teny s nasledujucimi argumentmi:  
1  
2  
3  
4 5 6
```

FUNKCIE „VRACAJÚCE“ SMERNÍK

- V jazyku C môže funkcia vrátiť dáta buď cez návratovú hodnotu alebo prostredníctvom parametrov.
- V prípade, že chceme aby funkcia vrátila smerník na objekt typu T cez návratovú hodnotu, musí byť deklarovaná nasledovne:
 - `T* funkcia();`
- V prípade, že chceme aby funkcia vrátila smerník cez parameter, musí byť deklarovaná nasledovne:
 - `void funkcia(T **data);`



FUNKCIA, KTORÁ VYMENÍ DVA SMERNÍKY

```
1  #include <stdio.h>
2
3  void swapPointers(int **a, int **b) {
4      int* tmp = *a;
5      *a = *b;
6      *b = tmp;
7  }
8
9  int main(int argc, char* argv[]) {
10     int iA, iB;
11     int *pA = &iA, *pB = &iB;
12
13     printf("Zadaj dve cele cisla:\n");
14     scanf("%d%d", &iA, &iB);
15
16     printf("Pred vymenou smernikov: *pA = %d, *pB = %d\n", *pA, *pB);
17     printf("Pred vymenou smernikov: pA = %p, pB = %p\n", pA, pB);
18     swapPointers(&pA, &pB);
19     printf("Po vymene smernikov: *pA = %d, *pB = %d\n", *pA, *pB);
20     printf("Po vymene smernikov: pA = %p, pB = %p\n", pA, pB);
21
22     return 0;
23 }
```

