Vícerozměrná pole

Úvod do programování 2

Tomáš Kühr

Organizační záležitosti

- Konzultace
  - Osobně – pracovna 5.043
    - Pondělí 12.00 – 13.00
    - Úterý 12.30 – 13.00
    - Čtvrtek 8.30 – 9.30
    - **Pátek 12.00 - 14.00**
  - Emailem tomas.kuhr@upol.cz
- Web předmětu: http://www.inf.upol.cz/kombinovane-studium/uvod-do-programovani-2
- Sbírka úloh: http://jazykc.inf.upol.cz/

Literatura

- Pavel Herout: Učebnice Jazyka C. Kopp, 2007.
- Brian W. Kernighan, Dennis M. Ritchie: Programovací jazyk C, Computer Press, 2006.
- Reek Kenneth: Pointers on C. Addison Wesley, 1997.
- Robert Sedgewick: Algorithms in C. Addison-Wesley Professional, 2001.
- Jeri R. Hanly, Elliot B. Koffman: Problem Solving and Program Design in C. Addison Wesley, 2006.
- Eric S. Roberts: Programming Abstractions in C. Addison Wesley, 1997.
- Eric S. Roberts: The Art and Science of C. Addison Wesley, 1994.
- a další…

Zápočet

- Za vyřešení úlohy Maticová kalkulačka
- Nebo za vyřešení libovolných dvou úloh z trojice:
  - Hledání nejdelších slov,
  - Medián čísel v souboru,
  - Vyhodnocování algebraických výrazů.
- Termín – konec srpna 2018

Jednorozměrná pole – připomenutí

- Definice:
  ```
  int pole[5] = { 1, 2, 3, 4, 5 };
  ```
- Přístup k prvkům:
  ```
  pole[4] = 3;
  printf("%i", pole[2]);
  ```
- Vztah ukazatelů a polí:
  `*(pole + i)` zcela odpovídá `pole[i]`
- Datový typ prvků v poli může být libovolný
- Nabízí se tedy otázky:
  - Lze vytvořit pole, jehož prvky budou pole?
  - Jak s tímto „vícerozměrným” polem pracovat?

Vytvoření vícerozměrného pole

- Bez inicializace:
  typ identifikátor[*velikost1*]...[*velikostN*];
- Příklady:

```
int moje_matice[3][4];
float trojrozmerne[3][4][10];
```

- S inicializací:
  typ identifikátor[*v1*]...[*vN*] = {*bloky hodnot*};
- Příklady:

```
int matice[2][3] = {{1, 2, 3}, {4, 5, 6}};
int m[2][3][4] = {
{{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}},
{{13, 14, 15, 16}, {17, 18, 19, 20}, {21, 22, 23, 24}}};
```

Práce s polem

- K prvků přistupujeme pomocí několika operátorů indexu []
- Příklad:

```
#define RADKY 3
#define SLOUPCE 4

int pole[RADKY][SLOUPCE];
for (int i = 0; i < RADKY; i++)
    for (int k = 0; k < SLOUPCE; k++)
        pole[i][k] = 1 + k + i * SLOUPCE;

for (int i = 0; i < RADKY; i++){
    for (int k = 0; k < SLOUPCE; k++)
        printf("%i\t", pole[i][k]);
    printf("\n"); }
```

Uložení pole v paměti

- Prvky pole `int x[2][3]` jsou v paměti uloženy v pořadí: `x[0][0]`, `x[0][1]`, `x[0][2]`, `x[1][0]`, `x[1][1]`, `x[1][2]`.
- Na dvourozměrné pole se lze dívat
  - Jako na jednorozměrné pole jednorozměrných polí daného typu
  - I jako na ukazatel na jednorozměrné pole daného typu
- Výraz `x` se vyhodnotí na adresu dvourozměrné pole
- Tato adresa je typu `int[2][3]` nebo `int(*)[3]`
- Výraz `x[0]` se vyhodnotí adresa prvního řádku
- Výraz `x[1]` pak bude adresa druhého řádku
- Obě výše uvedené adresy jsou typu `int[3]` neboli `int*`
- Přestože `x` a `x[0]` mají stejnou hodnotu
- Jsou `x+1` a `x[0]+1` jsou různé adresy

Pole jako parametr funkce

- Jednorozměrné:

```
int maximum(int cisla[], int pocet){ ... }
int maximum(int *cisla, int pocet){ ... }
...
int data[10] = {1, 45, 21, 5, 7, 2, 3, 35, 47, 4};
int max = maximum(data, 10);
```

- Dvourozměrné:

```
int maximum(int cisla[][3], int radku){ ... }
int maximum(int(*cisla)[3], int radku){ ... }
...
int data[2][3] = {{ 1, 45, 21 }, { 5, 7, 2 }};
int max = maximum(data, 2);
```

Pole jako parametr funkce

- Třírozměrné:

```
int maximum(int cisla[][3][4], int prvni){ ... }
int maximum(int(*cisla)[3][4], int prvni){ ... }
...
int data[2][3][4] = {
{{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}},
{{13, 14, 15, 16}, {17, 18, 19, 20}, {21, 22, 23, 24}}};
int max = maximum(data, 2);
```

- Parametrem lze zadat vždy jen první rozměr vstupního pole
- Proto se takto tato vícerozměrná pole obvykle nepoužívají

Statická vícerozměrná pole

- Příklad definice:
  ```
  int poleA[2][3];
  ```
- Nejjednodušší způsob vytvoření pole
- Pole je statické – oba rozměry se zadávají konstantou
- Rozměry pole jsou dány již při kompilaci
- Jednotlivé řádky mají vždy stejnou délku
- Prvky jsou v paměti uloženy za sebou po jednotlivých řádcích

Dvourozměrné pole jako ukazatel na pole

- Příklad vytvoření:
```
int(*poleB)[3];
poleB = (int(*)[3])malloc(2 * 3 * sizeof(int));
```
- Přístup k prvkům je stejný jako u statického pole
- Při kompilaci je nutné znát pouze druhý rozměr pole
- První rozměr může být vypočítán až za běhu programu
- Jednotlivé řádky mají vždy stejnou délku
- Řádky jsou v paměti uloženy za sebou jako u statického pole
- Pole je ale uloženo v dynamicky alokované paměti

Dvourozměrné pole jako pole ukazatelů

- Příklad vytvoření:

```
int *poleC[2];
poleC[0] = (int *)malloc(3 * sizeof(int));
poleC[1] = (int *)malloc(3 * sizeof(int));
```

- Přístup k prvkům je stejný jako u předchozích typů polí
- Při kompilaci je nutné znát pouze první rozměr pole
- Druhý rozměr pole je možné určit až za běhu programu
- Velikosti jednotlivých řádků nemusí být stejné
- Řádky jsou v paměti uloženy samostatně v dynamicky alokované paměti
- Pole ukazatelů na řádky je ale statické

Dvourozměrné pole jako ukazatel na ukazatel

- Příklad vytvoření:

```
int **poleD;
poleD = (int **)malloc(2 * sizeof(int *));
poleD[0] = (int *)malloc(3 * sizeof(int));
poleD[1] = (int *)malloc(3 * sizeof(int));
```

- Přístup k prvkům je stejný jako u předchozích typů polí
- Oba rozměry mohou být určeny až za běhu programu
- Řádky jsou uloženy samostatně a nemusí mít stejnou délku
- Celé pole je uloženo v dynamicky alokované paměti
- Díky své flexibilitě jeden z nejpoužívanějších způsobů vytvoření vícerozměrného pole
- Řádky lze samozřejmě vytvářet pomocí cyklu

Shrnutí – výhody a nevýhody

- Paměťové nároky:
  - `poleA` je paměťově nejvýhodnější – žádné pomocné ukazatele
  - Ostatní pole vyžadují paměť na uložení ukazatelů
  - Navíc dynamická alokace paměti obvykle spotřebuje další paměť na svou „administrativu”
- Rychlost přístupu k prvkům:
  - Obecně je přístup k běžným proměnným na zásobníku je rychlejší než přístup k dynamicky alokované paměti
  - Přístup k prvkům `poleA` je tedy nejrychlejší
  - Naopak nejpomalejší bude pravděpodobně přístup k prvkům `poleD`, kde se jde přes 2 ukazatele do dyn. alokované paměti

Pole textových řetězců

- Používané relativně často v programech, které pracují s textem
- Obvykle dvourozměrné pole typu `poleC` nebo `poleD`
- Příklad:

```
char *texty[3];
texty[0] = "Jedna";
texty[1] = strdup("Dva");
texty[2] = (char *)malloc(4 * sizeof(char));
strcpy(texty[2], "Tri");
```

- Pokud použijeme za identifikátorem pole pouze jeden index, pracujeme s celým řetězcem

```
printf("%s\n", texty[1]);
```

Jak číst (a psát) složitější deklarace?

- Příklad: `int *(*x)[3];`
- Najdeme identifikátor `x` a čteme: *„x je“*
- Od identifikátoru čteme doprava, dokud nenarazíme na `)` nebo `;`
- `)` nás vrací na odpovídající `(`
- Od ní čteme doprava až po již přečtený text: *„ukazatel na“*
- Přeskočíme již zpracovaný text a pokračujeme
- Opět dokud nenarazíme na `)` nebo `;` tedy: *„pole 3 prvků typu“*
- Pokud narazíme na `;`, přesuneme se na začátek již přečteného textu a čteme doleva: *„ukazatel na int“*
- Celkově: *„x je ukazatel na pole 3 prvků typu ukazatel na int“*

Pár příkladů navíc

- `int *pole[2];`
- `int (*pole)[3];`
- `long double *f(int, double);`
- `long double *(*f)();`
- `double *(*f[3])(double);`

A pro jistotu i řešení

- `int *poleX[2];`
  „poleX je pole dvou prvků typu ukazatel na int”
- `int (*poleY)[3];`
  „poleY je ukazatel na pole tří prvků typu int”
- `double *f(int, double);`
  „f je funkce s parametry typu int a double vracející ukazatel na double”
- `long double *(*f)();`
  „f je ukzatel na funkci bez parametrů vracející ukazatel na long double”
- `double *(*f[3])(double);`
  „f je pole tří prvků typu ukazatel na funkci s parametrem typu double vracející ukazatel na double”