B0B99PRPA – Procedurální programování

Pole a ukazatele, textové řetězce

Stanislav Vítek

Katedra radioelektroniky
Fakulta elektrotechnická
České vysoké učení v Praze

Přehled témat

- Část 1 – Pole a ukazatele

  Pole a ukazatele

  Vícerozměrná pole

- Část 2 – Textové řetězce

  Textové řetězce

- Část 3 – Alokace dynamické paměti

  Alokace dynamické paměti

Část I

Pole a ukazatele

I. Pole a ukazatele

Pole a ukazatele

- Ukazatel ukazuje na vyhrazenou část paměti proměnné

```
int *p;      // ukazatel (adresa)
sizeof(p);   // velikost promenne
```

- Pole je označení souvislého bloku paměti

```
int a[10];   // souvisly blok pameti
sizeof(a);   // 10*sizeof(int)
```

- Obě proměnné odkazují na paměť, kompilátor s nimi však pracuje rozdílně
  - Proměnná typu pole je symbolické jméno pro místo v paměti, kde jsou uloženy hodnoty prvků pole

    Kompilátor nahrazuje jméno přímo paměťovým místem.
  - Ukazatel obsahuje adresu, na které je příslušná hodnota (nepřímé adresování)
- Při předávání pole jako parametru funkce je předáváno pole jako ukazatel.

Pole a ukazatele

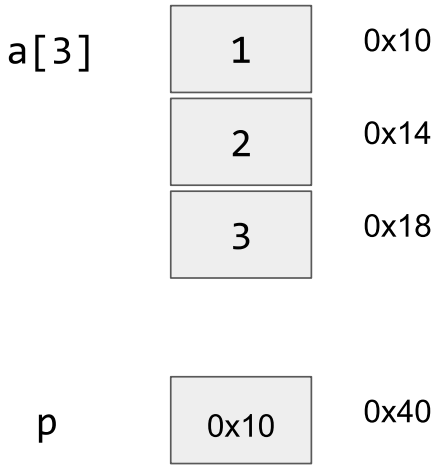
- Proměnná pole int `a[3] = 1,2,3;`

  Odkazuje na adresu prvního prvku pole.

- Proměnná ukazatel `int *p = a;`

  ukazatel p obsahuje adresu prvního prvku pole

- Hodnota `a[0]` přímo reprezentuje hodnotu na adrese `0x10`.



- Hodnota `p` je adresa `0x10`, kde je uložena hodnota 1. prvku pole.
- Přirazení `p = a` je možné.
- Kompilátor zajistí přirazení adresy prvního prvku do ukazatele.
- Prístup k 2. prvku lze použít jak `a[1]` tak `p[1]`.
- Obema přístupy se dostaneme na příslušné prvky pole, zpusob je však odlišný

Ukazatele využívají tzv. pointerovou aritmetiku.

Ukazatelová artitmetika

- S ukazateli lze provádět aritmetické operace + a -, tj. přičítat nebo odčítat celé číslo
  - ukazatel = ukazatel stejného typu + (nebo -) a celé číslo (int)
  - Nebo lze používat zkrácený zápis např. ukazatel += 1 a unární operátory např. ukazatel++
- Aritmetické operace jsou užitečné pokud ukazatel odkazuje na více položek daného typu (souvislý blok pameti)
  - Např. pole položek příslušného typu
  - Dynamicky alokovaný souvislý blok paměti
- Přičtením hodnoty celého čísla k ukazateli posouváme hodnotu ukazatele na další prvek, např.

```
int a[10];
int *p = a;
int i = *(p+2); //odkazuje na hodnotu a[2]
```

Pole a ukazatele – příklad

```
1  #include<stdio.h>
2
3  int main()
4  {
5      int *p;   // ukazatel na integer
6      int (*ptr)[5];  // ukazatel na pole peti prvku int.
7      int arr[5];
8
9      p = arr;    // ukazatel na prvni prvek pole
10     ptr = &arr;   // ukazatel na cele pole
11
12     printf("p = %p, ptr = %p\n", p, ptr);
13
14     p++;
15     ptr++;
16
17     printf("p = %p, ptr = %p\n", p, ptr);
18
19     return 0;
20 }
```

Funkce s polem jako argumentem

```
void fce(int A[]) {
  int B[] = { 2, 4, 6 };
  printf("[A] = %lu, [B] = %lu\n", sizeof(B), sizeof(B));
  for (int i = 0; i < 3; ++i) {
    printf("A[%i]=%i B[%i]=%i\n", i, A[i], i, B[i]);
  }
}
...
int array[] = { 1, 2, 3 };
fce(array);
```

- Po překladu (`gcc -std=c99`) na **amd64**
  - `sizeof(A)` vrátí velikost 8 bajtů (64-bitová adresa)
  - `sizeof(B)` vrátí velikost 12 bajtů (3×4 bajty – `int`)
- Pole se funkcím předává jako ukazatel na první prvek

I. Pole a ukazatele

Vícerozměrná pole

Pole lze definovat jako vícerozměrná, napr. 2D matice

```
int m[3][3] = {
  { 1, 2, 3 },
  { 4, 5, 6 },
  { 7, 8, 9 }
};
printf("Size of m: %lu == %lu\n", sizeof(m), 3*3*
    sizeof(int));

for (int r = 0; r < 3; ++r) {
  for (int c = 0; c < 3; ++c) {
    printf("%3i", m[r][c]);
  }
  printf("\n");
}
```

Vnitřní reprezentace vícerozměrných polí

- Vícerozměrné pole je vždy souvislý blok paměti

```
int *pm = (int *)m; // ukazatel na souvislou oblast
printf("m[0][0]=%i m[1][0]=%i\n", m[0][0], m[1][0]);
printf("pm[0]=%i pm[3]=%i\n", pm[0], pm[3]);
```

- Dvourozměrné pole lze také definovat jako ukazatel na ukazatele (pole ukazatelů) na hodnoty konkrétního typu, např.
  - `int **a;` – ukazatel na ukazatele
  - V obecném prípade však takový ukazatel nemusí odkazovat na souvislou oblast, kde jsou alokovány jednotlivé prvky.
  - Při přístupu jako do jednorozměrného pole `int *b = (int *)a;` tedy nelze garantovat přístup do druhého řádku jako v přechozím příkladě.

Vícerozměrná pole jako parametr funkce

- Parametr funkce je ukazatel na pole, např. typu `int`

```
int (*p)[3] = m; // pointer to array of int
printf("Size of p: %lu\n", sizeof(p));
printf("Size of *p: %lu\n", sizeof(*p)); // 3 *
  sizeof(int) = 12
```

- Funkci nelze deklarovat s argumentem typu `[][]` např.

```
int fce(int a[][]);
```

  - kompilátor nemůže správně spočítat index, pro přístup na `a[i][j]` se používá adresová aritmetika jinak

    Pro int m[row][col] totiž m[i][j] odpovídá hodnote na adrese *(m + col * i + j)

- Je však možné funkci deklarovat například jako
  - `int g(int a[]);` což odpovídá deklaraci `int g(int *a);`
  - `int fce(int a[][13]);` – je znám počet sloupců
  - nebo `int fce(int a[3][3]);`

Část II

Textové řetězce

II. Textové řetězce

Textové řetězce

Řetězcové literály

- Formát – posloupnost znaků a řídicích znaků uzavřená v uvozovkách

  `"Sud kulatý, rys tu pije!\n"`

  - Řetězcové konstanty oddělené oddělovači (white spaces) se sloučí

    `"Tu je kára," " ten to ryje!\n"`

    se sloučí do

    `"Tu je kára, ten to ryje!\n"`
- Typ – pole typu char zakončené znakem `'\0'`
  - Pole pro uložení řetězce musí být o jeden prvek větší než délka samotného řetězcového literálu

    ```
    char text[10]; // text o delce 9 znaku
    ```

Textový řetězec

- Textový řetězec můžeme inicializovat jako pole znaků `char[]`

```
printf("Size of str %lu\n", sizeof(str));
printf("Size of s %lu\n", sizeof(s));
printf("str '%s'\n", str);
printf(" s '%s'\n", s);
```

- Na textový řetězec lze odkazovat ukazatelem na znak `char*`

```
char *sp = "ABC";
printf("Size of ps %lu\n", sizeof(sp));
printf(" ps '%s'\n", sp);
```

  - Velikost ukazatele je 8 bytů (pro 64-bit OS)
  - Textový retezec musí být zakončen znakem `'\0'`

Načtení textového řetězce funkcí `scanf()`

- Použitím `%s` může dojít k přepisu paměti

```
char str0[4] = "PRP";
char str1[5];
printf("String str0 = '%s'\n", str0);
printf("Enter 4 chars: ");
scanf("%s", str1);
printf("You entered string '%s'\n", str1);
printf("String str0 = '%s'\n", str0);
```

- Načtení maximálně 4 znaků zajistíme řídicím řetězcem `%4s`

```
char str0[4] = "PRP";
char str1[5];
scanf("%4s", str1);
printf("You entered string '%s'\n", str1);
printf("String str0 = '%s'\n", str0);
```

Zjištění délky textového řetězce

- Textový řetězec v C je pole (`char[]`) nebo ukazatel (`char*`) odkazující na část paměti, kde je uložena příslušná posloupnost znaků.
- Textový řetězec je zakončen znakem `'\0'`
- Délku textového řetězce lze zjistit sekvenčním procházením znak po znaku až k `'\0'`

```
int delka(char *str) {
  int ret = 0;
  while (str && (*str++) != '\0') {
    ret += 1;
  }
  return ret;
}
...
char *text = "Hello PRP!\n";
printf("%i %lu\n", delka(text), strlen(text));
```

Práce s textovými řetězci

- Základní operace jsou definovány v knihovne `<string.h>`

```
int strlen (char *s);
char* strcpy(char *dst, char *src);
int strcmp(const char *s1, const char *s2);
```

  - Funkce předpokládají dostatečný rozsah alokovaných polí
  - Funkce s explicitním limitem na maximální délku retezcu:

```
char* strncpy(char *dst, char *src, size_t len);
int strncmp(const char *s1, const char *s2, size_t
len);
```

- Převod řetězce na číslo – `<stdlib.h>`

`atoi()`, `atof()` – převod celého a necelého čísla

```
long strtol(const char *nptr, char **endptr, int
base);
double strtod(const char *nptr, char **restrict
endptr);
```

Část III

Alokace dynamické paměti

III. Alokace dynamické paměti

Alokace dynamické paměti

Alokace dynamické paměti

- Přidělení bloku paměti velikosti `size` lze realizovat funkcí

```
void* malloc(size);
```

z knihovny `<stdlib.h>`

  - Velikost alokované pameti je uložena ve správci pameti
  - Velikost není soucástí ukazatele
  - Návratová hodnota je typu `void*` – pretypování nutné
- Příklad alokace paměti pro 10 proměnných typu int

```
int *int_array;
int_array = (int*)malloc(10 * sizeof(int));
```

- Operace s více hodnotami v pamětovém bloku je podobná poli
  - Používáme pointerovou aritmetiku
- Uvolnění paměti

```
void* free(pointer);
```

  - Správce paměti uvolní pamět asociovanou k ukazateli
  - Hodnotu ukazatele však nemění!

Stále obsahuje predešlou adresu, která však již není platná.

Příklad alokace dynamické paměti 1/3

- Alokace se nemusí nutne povést – testujeme návratovou hodnotu funkce `malloc()`
- Pro vyplnění adresy alokované paměti předáváme proměnnou jako ukazatel na proměnnou typu ukazatel na int

```
void* allocate_memory(int size, void **ptr)
{
  // ukazatel **ptr k uchovani ukazatele na nove alokovanou
  // pamet (t.j.  adresu mista, kde je uchovana adresa)

  *ptr = malloc(size);

  if (*ptr == NULL) {
    fprintf(stderr, "Error: allocation fail");
    exit(-1); /* alokace se nepovedla, program konci */
  }
  return *ptr;
}
```

Příklad alokace dynamické paměti

- Pro vyplnění hodnot pole alokovaného v dynamické paměti stačí předávat hodnotu adresy paměti pole

```
void fill_array(int* array, int size) {
  for (int i = 0; i < size; ++i) {
    *(array++) = random();
  }
}
```

- Po uvolnění paměti odkazuje ukazatel stále na původní adresu, proto lze explicitne nulovat.
- Předání ukazatele na ukazatele je nutné, jinak nulovat nelze.

```
void deallocate_memory(void **ptr) {
  if (ptr != NULL && *ptr != NULL) {
    free(*ptr);
    *ptr = NULL;
  }
}
```

Příklad alokace dynamické paměti

```
1  int main(int argc, char *argv[])
2  {
3    int *int_array;
4    const int size = 4;
5
6    allocate_memory(sizeof(int) * size, (void**)&
       int_array);
7    fill_array(int_array, size);
8    int *cur = int_array;
9
10   for(int i = 0; i < size; ++i, cur++) {
11     printf("Array[%d] = %d\n", i, *cur);
12   }
13   deallocate_memory((void**)&int_array);
14   return 0;
15 }
```