Řetězce

DELTA - Střední škola informatiky a ekonomie, s.r.o.

Ing. Luboš Zápotočný

04.12.2025

CC BY-NC-SA 4.0

Pole

Pole v C

Souvislá oblast paměti pro prvky stejného typu

```c
int arr[5];           // Declaration of array of 5 ints
int arr[] = {1,2,3};  // Initialization (size = 3)
```

Důležité vlastnosti:

- Prvky jsou v paměti **za sebou**
- Index začíná od **0**
- Velikost je **fixní** (nelze měnit)
- Název pole je ve skutečnosti **pointer** na první prvek

Pole v C

```c
int arr[5] = {10, 20, 30, 40, 50};
printf("%d\n", arr[0]);   // 10
printf("%d\n", *arr);     // 10 (same!)
printf("%d\n", arr[2]);   // 30
printf("%d\n", *(arr+2)); // 30 (same!)
```

Předávání pole do funkcí

Klíčové: Pole se do funkce předává jako **pointer**!

Tyto dva zápisy jsou identické:

```
void func(int arr[], int n) {
    // ...
}
```

```
void func(int *arr, int n) {
    // ...
}
```

Předávání pole do funkcí

Důsledky:

- Funkce dostává jen **adresu** prvního prvku
- Neví, jak je pole velké! ⟶ musíme předat velikost
- Změny v poli se projeví i mimo funkci (předává se reference)
- `sizeof(arr)` ve funkci vrací velikost pointeru, ne pole!

Předávání pole do funkcí

```c
void printArray(int *arr, int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

void modifyArray(int *arr, int size) {
    for (int i = 0; i < size; i++) {
        arr[i] *= 2;  // Double each element
    }
}
```

Předávání pole do funkcí

```
int main() {
    int numbers[] = {1, 2, 3, 4, 5};
    int size = 5;

    printArray(numbers, size);  // 1 2 3 4 5
    modifyArray(numbers, size);
    printArray(numbers, size);  // 2 4 6 8 10
}
```

Inicializace pole uvnitř funkce

```c
void initArray(int **a, int *n) {
    *a = (int *) (malloc((*n = 3) * sizeof(int)));
    (*a)[0] = 9; // pay attention to the parentheses - segfault
    (*a)[1] = 8;
    (*a)[2] = 7;
}
int main() {
    int *arr, size;
    initArray(&arr, &size);
    for (int i = 0; i < size; i++) printf("%d\n", arr[i]);
    free(arr);
    return 0;
}
```

Řetězce (strings) v C

Co je to řetězec (string)

V C neexistuje speciální typ `string`!

String = pole znaků zakončené `'\0'`

```
char str[] = "Hello";
// Internally: {'H', 'e', 'l', 'l', 'o', '\0'}
```

Důležité:

- Znak `'\0'` (null terminator, ASCII kód 0) označuje konec stringu
- Funkce jako `printf("%s")` čtou znaky, než narazí na `'\0'`
- Velikost pole pro string musí být **délka + 1** (pro `'\0'`)

Co je to řetězec (string)

```c
char str1[6] = "Hello";  // 5 chars + 1 for '\0'
printf("%s\n", str1);
```

```
Hello
```

```c
char str2[5] = "World";  // missing space for '\0'
printf("%s\n", str2);
```

```
World?3?Hello
```

Deklarace a inicializace řetězců

```c
// 1. Automatic size
char str1[] = "Hello";  // size = 6 (including '\0')

// 2. Explicit size
char str2[10] = "Hello";  // remaining space = '\0'

// 3. By characters
char str3[] = {'H', 'e', 'l', 'l', 'o', '\0'};

// 4. String literal (read-only!)
char *str4 = "Hello";  // cannot be modified!
```

Deklarace a inicializace řetězců

```
str1[0] = 'h';  // OK
str4[0] = 'h';  // Bus error! - cannot write to read-only memory
```

Null terminátor

`'\0'` je klíčový pro práci se stringy

```c
char str[] = "Hello World";
printf("%s\n", str);   // Hello World
```

Co když ho změníme?

```c
str[5] = '\0';
printf("%s\n", str);
```

```
Hello
```

Null terminátor

Ale data jsou stále v paměti:

```c
for (int i = 0; i < 11; i++) {
    printf("%c", str[i]);
}
printf("\n");
for (int i = 0; i < 11; i++) {
    printf("%d ", str[i]); // ASCII values
}
printf("\n");
```

```
HelloWorld
72 101 108 108 111 0 87 111 114 108 100
```

Knihovna `string.h`

Podporné funkce `str...` jsou dostupné v knhovně `string.h` a lze ji imporotvat pomocí

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

Kopírování řetězců - `strcpy()`

Funkce `strcpy()` zkopíruje řetězec

```
char* strcpy(char* destination, const char* source);
```

Použití:

```
char src[] = "Hello World";
char dst[12];  // 11 chars + '\0'

strcpy(dst, src);
printf("%s\n", dst);  // Hello World
```

Kopírování řetězců - `strcpy()`

Nebezpečí - Buffer overflow:

```c
char src[] = "Hello World";  // 11 chars + '\0' = 12
char dst[10];

strcpy(dst, src);
printf("%s\n", src);
printf("%s\n", dst);
```

```
d
Hello World
```

Proč?

Kopírování řetězců - `strcpy()`

```
printf("src = %p\n", src);
printf("dst = %p\n", dst);
```

```
src = 0x7fff2660dd24
dst = 0x7fff2660dd1a
```

Kopírování řetězců - `strcpy()`

Stav před voláním `strcpy()`:

0x1a	0x1b	0x1c	0x1d	0x1e	0x1f	0x20	0x21	0x22	0x23	0x24	0x25	0x26	0x27	...
?	?	?	?	?	?	?	?	?	?	H	e	l	l	...

Stav po volání `strcpy()`:

0x1a	0x1b	0x1c	0x1d	0x1e	0x1f	0x20	0x21	0x22	0x23	0x24	0x25	0x26	0x27	...
H	e	l	l	o		W	o	r	l	d	0	l	l	...

Kopírování řetězců - `strcpy()`

Řešení: Používejte `strncpy()` nebo vždy kontrolujte velikost!

- `n` v `strncpy()` je maximální počet znaků k zkopírování
- `strncpy()` nepřidává null terminátor

```c
char src[] = "Hello World";  // 11 chars + '\0' = 12
char dst[10];
// Safe copying - max 9 chars + manual null terminator
strncpy(dst, src, 10 - 1);
dst[9] = '\0';  // Ensure null terminator

printf("dst = '%s'\n", dst);  // "Hello Wor"
printf("src = '%s'\n", src);  // "Hello World" (not overwritten!)
```

Užitečné string funkce

```c
#include <string.h>

char str1[] = "Hello";
char str2[] = "World";

strlen(str1);           // 5 (length without '\0')

strcmp(str1, str2);     // < 0 (lexicographic comparison, in ASCII)
// 0 if strings are equal
// positive if str1 is greater
// negative if str2 is greater

// ASCII('H') = 72
// ASCII('h') = 104
// 72 < 104 ==> strcmp(str1, str2) < 0
```

Užitečné string funkce

```
strcat(str1, str2);    // Concatenate strings (be careful with size!)
strchr(str1, 'l');     // Find first 'l'
strstr(str1, "ll");    // Find substring "ll"
```

Pozor na `strcat()`:

```
char str[20] = "Hello";
strcat(str, " World");  // OK, fits in
strcat(str, " Very Long String");  // Buffer overflow!
```