Pole a základní alokace paměti

Tomáš Faltejsek, Luboš Zápotočný, Michal Havelka

2022

Obsah

Struct

- Klíčové slovo pro uživatelem definované vytváření typů
- **Seskupení** různých datových typů do jednoho

Deklarace:

Struct

- Klíčové slovo pro uživatelem definované vytváření typů
- **Seskupení** různých datových typů do jednoho

Deklarace:

```c
#include <stdio.h>
#include <stdlib.h>

// Deklarujeme pouze strukturu 'Person'
struct Person {
    char name[20];
    int age;
};

// Deklarujeme promennou 'point1'
// a strukturu 'Point' zaroven
struct Point {
    int x, y;
} point1; // promenna point1 je nyni globalne dostupna
```

Struct - inicializace

```
#include <stdio.h>
#include <stdlib.h>

struct Point
{
    // COMPILER ERROR:  cannot initialize members here
    int x = 0;
    // COMPILER ERROR:  cannot initialize members here
    int y = 0;
};

struct Point
{
    int x, y;
};

int main()
{
    // Korektne provedena inicializace
    // p1.x je nastaveno na 0
    // p1.y je nastaveno na 1
    struct Point p1 = {0, 1};
}
```

Nenainicializované hodnoty

```c
#include <stdio.h>

typedef struct {
    long weight;
    char countOfDoors;
} Car;

int main() {
    long a;
    printf("%ld\n", a);

    Car c;
    printf("%ld␣%d\n", c.weight, c.countOfDoors);
}
```

Output

```
1
4636248195 19
```

Klíčové slovo Static v jazyce C

- Static proměnná zachovávají hodnotu i mimo jejich scope
- Modifikace přístupu pouze na modul, ve kterém je vytvořena (vs globální proměnná dostupná napříč moduly)
- Static proměnná je alokována na *data* segmentu, zůstává v paměti během běhu programu

```c
#include<stdio.h>
int fun()
{
  static int count = 0;
  count++;
  return count;
}

int main()
{
  printf("%d\n", fun());
  printf("%d\n", fun());
  return 0;
}
```

Pointer

- Adresa paměti kde začíná referencovaná hodnota
- Nemá datový typ

Pointer - zápis

```
// Obecna syntaxe
datatype *var_name;

// An example pointer "ptr" that holds
// Pointer drzici adresu integer promene
// Ktera muze byt prectena srkze "ptr"
int *ptr;
```

Pointer madness

```c
#include <stdio.h>

int main() {
    int a = 1;

    int *pa = &a;
    int **ppa = &pa;
    int ***pppa = &ppa;

    printf("%d, %d, %d, %d\n", a, *pa, **ppa, ***pppa);
}
```

Pointer madness

```c
#include <stdio.h>

int main() {
    int a = 1;

    int *pa = &a;
    int **ppa = &pa;
    int ***pppa = &ppa;

    printf("%d, %d, %d, %d\n", a, *pa, **ppa, ***pppa);
}
```

Otázka

Co bude výstupem?

Pointer aritmetika

```c
#include <stdio.h>

int main()
{
    char str[] = "Hello";

    for (int i = 0; i < 5; ++i) {
        printf("%c", str[i]);
    }
    printf("\n");

    for (int i = 0; i < 5; ++i) {
        printf("%c", *(str + i));
    }
    printf("\n");
}
```

Pointer aritmetika se strukturami

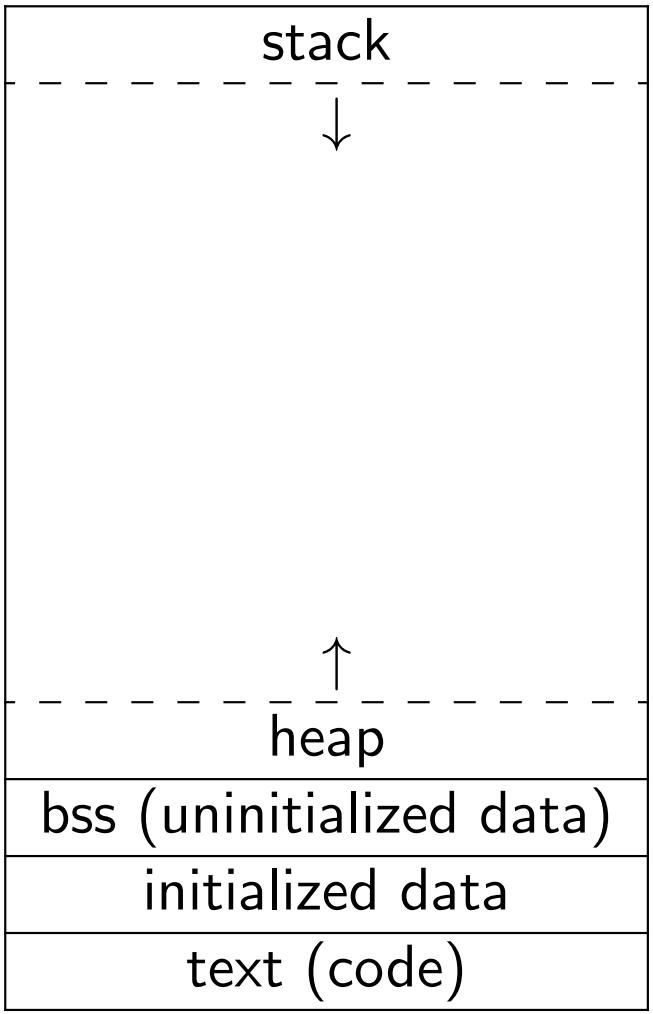
```c
#include <stdio.h>

typedef struct {
    long weight;
    char countOfDoors;
} Car;

int main() {
    Car arr[] = {
            {.weight = 1000, .countOfDoors = 5},
            {.weight = 2000, .countOfDoors = 3},
    };

    for (int i = 0; i < 2; ++i) {
        printf("%ld␣%hhd\n",
               arr[i].weight,
               (arr + i)->countOfDoors);
    }
}
```

Paměť v programech C - segmentace paměti



- *stack*: fixně rezervovaná paměť, volání funkcí, lokální proměnné
- *heap*: dynamická paměť (v režii programátora)
- *bss*: před spuštěním programu, statické a globální proměnné inicializováné na 0 či bez explicitní inicializace
- *initialized data*: statické a globální proměnné (celý lifecycle programu)
- *text*: executable instrukce

Segmentace paměti - ukázka

```c
#include <stdio.h>

int sum;

int sqr(int num) {
    return num*num;
}

int squaredSum(int a, int b) {
    int sum = sqr(a + b);

    return sum;
}

int main() {
    int x = 5, y = 4;

    sum = squaredSum(x, y);

    printf("SUM = %d\n", sum);
}
```

Stack

- LIFO
- **Fixně** přidělená paměť která se v průběhu programe **nemění**
- Alokuje paměť při deklaraci lokální proměnné
- Alokuje paměť při každém volání funkce
- Ve chvíli kdy funkce vrátí hodnotu (return) dealokuje jí (a její lokálné proměnné)
- **Alokace a dealokace** na stacku se děje automaticky (*viz lekce 06*)
- (Většinou) Začíná na "high address"
- Proměnné na stacku se nazývají *automatické* proměnné

Stack

- LIFO
- **Fixně** přidělená paměť která se v průběhu programe **nemění**
- Alokuje paměť při deklaraci lokální proměnné
- Alokuje paměť při každém volání funkce
- Ve chvíli kdy funkce vrátí hodnotu (return) dealokuje jí (a její lokálné proměnné)
- **Alokace a dealokace** na stacku se děje automaticky (*viz lekce 06*)
- (Většinou) Začíná na "high address"
- Proměnné na stacku se nazývají *automatické* proměnné

Otázka

Je dobrý nápad ukládat pointer na automatickou proměnnou?

Pointer na automatickou proměnnou

```c
#include <stdio.h>
#include <stdlib.h>
struct Person {
    char name[20];
    int age;
};
struct Person *createJohn() {
    struct Person p = {"John", 20};
    return &p;
}
int main() {
    struct Person *john = createJohn();

    printf("%s \n", john->name);
    return 0;
}
```

Output

struct.c:12:13: warning: address of stack memory associated with local variable 'c' returned [-Wreturn-stack-address]

Stack - stack overflow

```c
#include <stdio.h>

int main() {
    int arrayThatWillCauseStackOverflow[100000000000];

    return 0;
}
```

Stack - stack overflow

```c
#include <stdio.h>

int main() {
    int arrayThatWillCauseStackOverflow[100000000000];

    return 0;
}
```

Output

```
[1] 97844 segmentation fault ./a.out
```

Dynamická paměť v C - Heap

- paměť v heapu má v režii programátor – **neprobíhá** zde automatické uvolnění paměti
- **Nejedná** se o datovou strukturu heap
- Není thread-safe

Dynamická paměť - pole na heapu

Syntaxe:

```
ptr = (cast-type*) malloc(byte-size)
```

Základní použití:

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int length = 20;
    int *arr = (int*) malloc(length * sizeof(int));

    for(int i = 0; i < length; i++) {
        arr[i] = i + 1;
    }

    for(int i = 0; i < length; i++) {
        printf("%d: %d\n", i, arr[i]);
    }

    // free(arr) -- memory leak
    return 0;
}
```

Dynamická paměť - realokace

```c
char *string = (char *) malloc(size);


// nova velikost stringu
char *newString = (char *) realloc(string, size + 1);

if (newString == NULL) { // realokace selhala
    free(string); // musime uvolnit manualne
    return 0;
}

string = newString;
```

Předávání pole do funkce - hranaté závorky

```c
#include <stdio.h>

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

int main() {
    int arr[] = {9, 8, 7, 6, 5};
    printArray(arr, 5);
    return 0;
}
```

Předávání pole do funkce - pointer

```c
#include <stdio.h>

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

int main() {
    int arr[] = {9, 8, 7, 6, 5};
    printArray(arr, 5);
    return 0;
}
```

Inicializace pole uvnitř funkce

```c
#include <stdio.h>
#include <stdlib.h>
void initArray(int **a, int *n) {
    *a = (int *) malloc((*n = 3) * sizeof(int));
    (*a)[0] = 9;
    (*a)[1] = 8; // pozor na zavorky - segfault
    (*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;
}
```

Zkrácení řetězce bez tvorby nového řetězce

```c
#include <stdio.h>

int main () {
    char str[] = "Hello World";
    printf("%s\n", str);    // Hello World

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

    for (int i = 0; i < 11; i++)
        printf("%c", str[i]);
    printf("\n");           // HelloWorld

    return 0;
}
```

Kopírování řetězce

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

int main () {
    char src[] = "Hello World";
    char dst[12];  // 11 charu + 1 null char

    strcpy(dst, src);

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

    return 0;
}
```

Kopírování řetězce - nebezpečné

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

int main () {
    char src[] = "Hello World";
    char dst[11];  // 11 charu [bez null charu]

    strcpy(dst, src); // platform specific, error

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

    return 0;
}
```

Segmentation fault

Porušení ochrany paměti (též chyba paměťové ochrany, anglicky segmentation fault) je obecně snaha přistoupit k paměti počítače, kterou procesor nemůže fyzicky adresovat. Nastává v případě, kdy hardware upozorní operační systém o nepovoleném přístupu k paměti. Jádro operačního systému na tuto událost obvykle zareaguje nápravným krokem. [Wikipedia]

Segmentation fault

Porušení ochrany paměti (též chyba paměťové ochrany, anglicky segmentation fault) je obecně snaha přistoupit k paměti počítače, kterou procesor nemůže fyzicky adresovat. Nastává v případě, kdy hardware upozorní operační systém o nepovoleném přístupu k paměti. Jádro operačního systému na tuto událost obvykle zareaguje nápravným krokem. [Wikipedia]

A segmentation fault occurs when a program attempts to access a memory location that it is not allowed to access, or attempts to access a memory location in a way that is not allowed. [Wikipedia]

Segmentation fault - bad memory access

```c
#include <stdio.h>

int main()
{
    int *ptr = NULL; // NULL = ((void *)0)
    printf("%d", *ptr);
}
```

Segmentation fault - memory bus error

```
int main()
{
    char *str = "hello";
    *(str + 1) = 'H';
    // bus error - write to read only memory
}
```

Segmentation fault - malloc přiřazuje více, než potřebujeme

```c
#include <stdlib.h>
#include <stdio.h>
int main()
{
    char *str = (char *) malloc(3 * sizeof(char));
    for (int i = 0; i < 1000000; ++i) {
        *(str + i) = 0;
        printf("%d\n", i);
    }

    // ...
    // 134494
    // 134495
    // Segmentation fault
}
```

Binární vyhledávání (v seřazeném poli)

Díváme se na prostřední prvek v seřazeném poli a dle porovnání s hledanou hodnotou dostávám informaci, zdali jsme prvek našli a pokud ne, tak v jaké polovině pole máme daný prvek hledat

Binární vyhledávání

Given a sorted array arr[] of n elements, write a function to search a given element x in arr[].

- Lineární vyhledávání: $O(n)$
- Binární vyhledávání: $O(logn)$ (*viz lekce 04*)

Kroky algoritmu:

1. Prohledáváný interval pokrývá rozsah celého pole, interval je ohraničen levou a pravou mezí
2. Nalezení prostředního elementu
3. Pokud je hodnota hledaného elementu x nižší, než hodnota prostředního, zužíme prohledávaný interval pouze na levou část (*binární půlení*) v opačném případě zužíme na pravou část
4. Opakuji proceduru do chvíle kdy je nalezen element x, nebo je interval prázdný (element x v poli neexistuje)

Hledáme číslo 10

0	1	2	3	4	**5**	6	7	8	9	10	11
1	2	3	4	5	6	7	8	9	10	11	12

Hledáme číslo 10

0	1	2	3	4	**5**	6	7	8	9	10	11
1	2	3	4	5	6	7	8	9	10	11	12

6	7	**8**	9	10	11
7	8	9	10	11	12

Hledáme číslo 10

0	1	2	3	4	**5**	6	7	8	9	10	11
1	2	3	4	5	6	7	8	9	10	11	12

6	7	**8**	9	10	11
7	8	9	10	11	12

9	**10**	11
10	11	12

Hledáme číslo 10

9	**10**	11
10	11	12

Hledáme číslo 10

9	**10**	11
10	11	12

9
10

Číslo 10 bylo nalezeno na indexu 9
Bylo za potřebí pouze $\mathcal{O}(\log n)$ operací pro vyhledání kteréhokoli prvku v takto seřazeném poli.
Oproti linárnímu vyhledávání, které by muselo zkontrolovat každý prvek, tedy $\mathcal{O}(n)$ operací

Binární vyhledávání - iterativní zápis v C

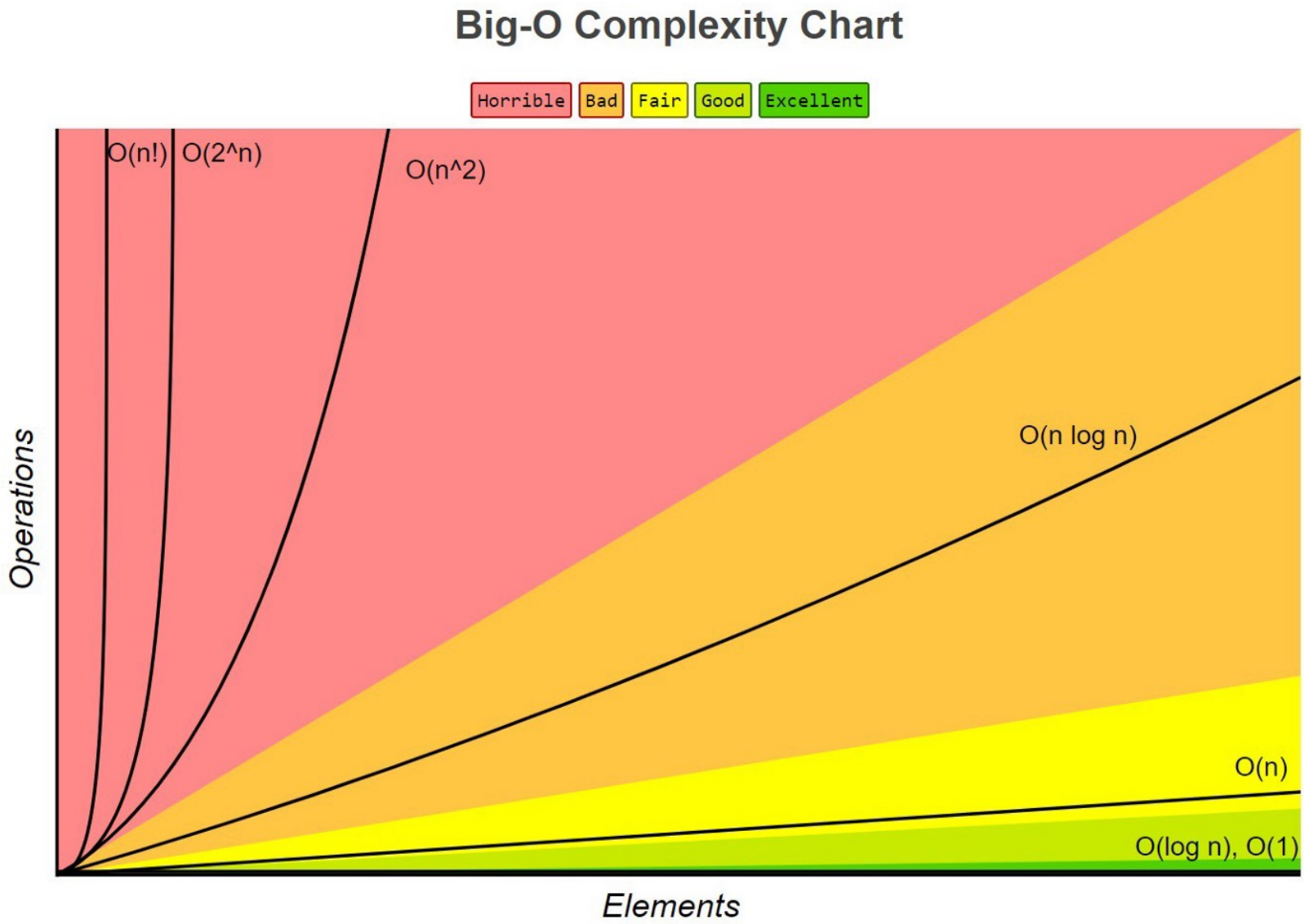
```
int binarySearch(int arr[], int l, int r, int x)
{
    while (l <= r) {
        int m = l + (r - l) / 2;

        // Check if x is present at mid
        if (arr[m] == x)
            return m;

        // If x greater, ignore left half
        if (arr[m] < x)
            l = m + 1;

        // If x is smaller, ignore right half
        else
            r = m - 1;
    }

    // if we reach here, then element was
    // not present
    return -1;
}
```



[https://towardsdatascience.com/understanding-time-complexity-with-python-examples-2bda6e8158a7]