Dynamická paměť

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

Ing. Luboš Zápotočný

04.12.2025

Dynamická paměť

Heap

Heap (halda) je oblast paměti pro

Heap

Heap (halda) je oblast paměti pro **dynamická data**

Heap

Heap (halda) je oblast paměti pro **dynamická data**

Kdy potřebujeme Heap?

- Velikost dat není známá při kompilaci

Heap

Heap (halda) je oblast paměti pro **dynamická data**

Kdy potřebujeme Heap?

- Velikost dat není známá při kompilaci
- Data musí „přežít“ funkci

Heap

Heap (halda) je oblast paměti pro **dynamická data**

Kdy potřebujeme Heap?

- Velikost dat není známá při kompilaci
- Data musí „přežít“ funkci
- Velká data (stack je omezený)

Heap

```c
// Should not do this - size is not known at compile time
int n;
scanf("%d", &n);
int arr[n];  // VLA - variable length array (C99)

// Do this - dynamic allocation
int *arr = malloc(n * sizeof(int));
```

Porovnání heapu a stacku

	Stack	Heap
Správa	Automatická	Manuální (`malloc`/`free`)
Velikost	Fixní (malá)	Velká (až do RAM)
Rychlost	Velmi rychlý	Pomalejší
Životnost	Jen v rámci funkce	Až do `free()`
Chyby	Stack overflow	Memory leak, use after free

Porovnání heapu a stacku

Pravidlo: Použij „stack“ když můžeš, „heap“ když musíš!

malloc()

Funkce `malloc()` alokuje paměť na heapu

malloc()

Funkce `malloc()` alokuje paměť na heapu

Syntaxe:

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

malloc()

Funkce `malloc()` alokuje paměť na heapu

Syntaxe:

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

Použití:

```
#include <stdlib.h>

int *ptr = (int *) (sizeof(int));      // 1 int
int *arr = (int *) malloc(10 * sizeof(int)); // array of 10 ints
```

malloc()

Důležité:

- Vrací `void*` (generický pointer), nutné přetypovat na požadovaný typ

malloc()

Důležité:

- Vrací `void*` (generický pointer), nutné přetypovat na požadovaný typ
- Vrací `NULL` pokud alokace selže

malloc()

Důležité:

- Vrací `void*` (generický pointer), nutné přetypovat na požadovaný typ
- Vrací `NULL` pokud alokace selže
- Paměť je **neinicializovaná** (obsahuje smetí)

malloc()

Důležité:

- Vrací `void*` (generický pointer), nutné přetypovat na požadovaný typ
- Vrací `NULL` pokud alokace selže
- Paměť je **neinicializovaná** (obsahuje smetí)
- **Musíme** volat `free()` když jsme hotovi!

malloc()

Důležité:

- Vrací `void*` (generický pointer), nutné přetypovat na požadovaný typ
- Vrací `NULL` pokud alokace selže
- Paměť je **neinicializovaná** (obsahuje smetí)
- **Musíme** volat `free()` když jsme hotovi!

malloc() - bezpečné použití

Vždy kontrolujte, zda `malloc()` uspěl!

```c
int *arr = (int *) malloc(1000 * sizeof(int));
if (arr == NULL) {
    fprintf(stderr, "Error: cannot allocate memory!\n");
    return 1;
}
// Now we can safely use arr
for (int i = 0; i < 1000; i++) {
    arr[i] = i;
}
free(arr);  // Important! Free the memory
```

malloc() - bezpečné použití

Tipy:

- Vždy používejte `sizeof(typ)` místo hardcoded čísel

malloc() - bezpečné použití

Tipy:

- Vždy používejte `sizeof(typ)` místo hardcoded čísel
- Kontrolujte návratovou hodnotu
- Po `free()` nastavte pointer pro jistotu na `NULL`

free()

Co je `free()`?

free()

Co je `free()`?

Funkce která **uvolní** dynamicky alokovanou paměť

free()

Co je `free()`?

Funkce která **uvolní** dynamicky alokovanou paměť

```
int *ptr = (int *) malloc(sizeof(int));
*ptr = 42;

free(ptr);    // Free the memory
ptr = NULL;   // Best practice
```

Časté chyby při práci s dynamickou pamětí

```c
// Memory leak - forgot to free()
int *p1 = (int *) malloc(100);
// ... use p1 ...
// program ends without free(p1)

// Double free - free 2×
free(ptr);
free(ptr);  // ERROR!

// Use after free
free(ptr);
*ptr = 10;  // ERROR! ptr is not valid anymore
```

Kompletní příklad

```c
int main() {
    int n = 5;
    int *arr = (int *) malloc(n * sizeof(int));

    if (arr == NULL) {
        return 1;  // Allocation failed
    }

    // Initialization
    for(int i = 0; i < n; i++) {
        arr[i] = i * 10;
    }
```

Kompletní příklad

```
    // Use
    for(int i = 0; i < n; i++) {
        printf("%d ", arr[i]);  // 0 10 20 30 40
    }

    free(arr);      // Free the memory
    arr = NULL;     // Prevent use-after-free
    return 0;
}
```

realloc()

Co když potřebujeme změnit velikost alokované paměti?

realloc()

Co když potřebujeme změnit velikost alokované paměti?

Funkce `realloc()` změní velikost existujícího bloku

```
void* realloc(void* ptr, size_t new_size);
```

realloc()

Co když potřebujeme změnit velikost alokované paměti?

Funkce `realloc()` změní velikost existujícího bloku

```
void* realloc(void* ptr, size_t new_size);
```

Chování:

- Pokud se vejde na stejné místo, tak se nic neděje
- Pokud ne, tak se zkopírují data na nové místo a staré se uvolní
- Vrací pointer na (možná nové) místo v paměti
- Vrací `NULL` pokud selže (původní paměť zůstane!)

realloc() - bezpečné použití

```c
int size = 10;
int *arr = (int *) malloc(size * sizeof(int));
// ... use arr ...
// We need a bigger array
int new_size = 20;
int *temp = (int *) realloc(arr, new_size * sizeof(int));
if (temp == NULL) {
    // Allocation failed, arr is still valid!
    free(arr);
    return 1;
}

arr = temp;  // Now we can overwrite arr
```

realloc() - bezpečné použití

Pozor: Nikdy nepřepisujte původní pointer dokud nekontrolujete `NULL`!

calloc()

`calloc()` = `malloc()` + inicializace na nulu

```
void* calloc(size_t count, size_t size);
```

calloc()

`calloc()` = `malloc()` + inicializace na nulu

```
void* calloc(size_t count, size_t size);
```

Porovnání:

```
// malloc - not initialized memory
int *arr1 = malloc(5 * sizeof(int));
// arr1[0], arr1[1], ... contains random data

// calloc - initialized memory
int *arr2 = calloc(5, sizeof(int));
// arr2[0] == 0, arr2[1] == 0, ...
```

calloc()

Použití:

- Výhodnější pro velká pole (OS optimalizace)
- Bezpečnější - žádná náhodná data