Základy programovania (IZP)
Dynamická alokácia (7. cvičenie)

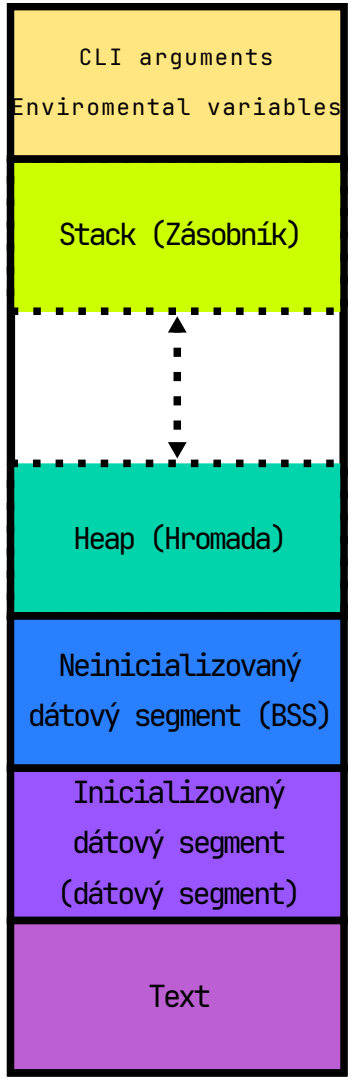
Ing. Pavol Dubovec

Vysoké Učení Technické v Brně, Fakulta informačních technologií
Božetěchova 1/2. 612 66 Brno- Královo Pole
idubovec@fit.vutbr.cz

Ako program v jazyku C spravuje pamäť
pre rôzne typy premenných a funkcií?

Kde sú tieto premenné uložené počas behu programu?

Stack (Zásobník): Spravuje volania funkcií a lokálne premenné. Každé volanie funkcie vytvára nový stack frame, ktorý obsahuje lokálne premenné a informácie potrebné na návrat z funkcie. Je rýchly, ale obmedzený veľkosťou.
Heap (Hromada): Používa sa na dynamickú alokáciu pamäte. Je alokovaná a dealokovaná počas behu programu pomocou funkcií ako malloc, calloc, realloc a free.
Text (kódový segment): Obsahuje zdrojový kód programu. Zvyčajne slúži iba na čítanie, aby sa zabránilo náhodným alebo škodlivým zmenám kódu počas behu programu.




```c
int global_var = 10; // Inicializovany datovy segment
int global_uninit;   // Neinicializovany datovy segment (BSS)

int main(int argc, char *argv[]) { // Argumenty prikazoveho riadka
    int local_var = 5; // Lokalna premenna na stacku
    int *heap_var = (int*)malloc(sizeof(int)); // Dynamicka alokacia pamate na heap
    free(heap_var); // Uvolnenie pamate
    return 0;
}
```

Akú veľkosť zaberajú rôzne dátové typy v pamäti počítača?

Teoretické opakovanie – veľkosť dátových typov a pole

- Rôzne dátové typy zaberajú rôzne množstvo miesta v pamäti počítača:
  - Veľkosť je určená príkazom sizeof, ktorý vracia veľkosť v bajtoch.
- Veľkosť dátových typov nie je pevne stanovená a závisí od operačného systému

```
printf("Velkost char: %zu bajtov\n", sizeof(char)); // Velkost char: 1 bajtov
printf("Velkost short: %zu bajtov\n", sizeof(short)); // Velkost short: 2 bajtov
printf("Velkost int: %zu bajtov\n", sizeof(int)); // Velkost int: 4 bajtov
printf("Velkost long: %zu bajtov\n", sizeof(long)); // Velkost long: 8 bajtov
printf("Velkost float: %zu bajtov\n", sizeof(float)); // Velkost float: 4 bajtov
printf("Velkost double: %zu bajtov\n", sizeof(double)); // Velkost double: 8
bajtov
```

- Položky poľa sú uložené v pamäti ako jeden súvislý blok dát, preto je veľkosť poľa násobkom počtu a veľkosti jeho položiek – `malloc(dlzka * sizeof (int));`
- Pomocou príkazu `sizeof` nie je možné zistiť veľkosť dynamicky alokovanej pamäte. Namiesto toho tento príkaz vracia veľkosť ukazovateľa (adresy), ktorá zostáva konštantná bez ohľadu na veľkosť pamäte, na ktorú ukazuje.
- V prípade, že pole neobsahuje žiadne prvky, jeho adrese sa priradí hodnota NULL.

- Položky štruktúry sú automaticky zarovnané v pamäti na veľkosť slova, preto veľkosť štruktúry môže byť väčšia ako súčet veľkostí jej položiek.
- Jazyk C definuje minimálne veľkosti pre dátové typy, ale konkrétne implementácie môžu byť rôzne v závislosti od kompilátora a operačného systému!

```c
// Dynamická alokácia pamäte pre pole
int dlzka = 10;
int *pole = (int*)malloc(dlzka * sizeof(int)); // Alokuje pamat pre pole 10 integerov
if (pole == NULL) {
printf("Alokacia pamate zlyhala\n"); // Alokacia pamate zlyhala
return 1;
}

typedef struct struct_unaligned_tag {
    char c;
    double d;
    int s;
} struct_unaligned_t;

typedef struct struct_aligned_tag {
    double d;
    int s;
    char c;
} struct_aligned_t;
```

```c
printf("sizeof(structc_t) = %llu\n", sizeof(struct_unaligned_t)); // Vysledok 24
printf("sizeof(structd_t) = %llu\n", sizeof(struct_aligned_t)); // Vysledok 16
```

Kde sú uložené lokálne premenné a parametre funkcií počas behu programu?

Čo sa stane s pamäťou lokálnych premenných po ukončení funkcie?

Teoretické opakovanie – automatická alokácia pamäte (stack)

- Používa sa na automatickú alokáciu pamäte pre lokálne premenné a parametre funkcií.
- Lokálne premenné sú automaticky alokované pri vstupe do funkcie a dealokované pri jej ukončení.
- Každé volanie funkcie vytvára nový stack frame, ktorý obsahuje lokálne premenné a informácie potrebné na návrat z funkcie.
- Výhody:
  - ⊕ Rýchla alokácia a dealokácia.
  - ⊕ Jednoduchá správa pamäte.
- Nevýhody:
  - ⊖ Obmedzené veľkosťou stacku.
  - ⊖ Možnosť stack overflow pri nadmernom využití.

Ako by ste alokovali pamäť pre pole, ktorého veľkosť nie je známa v čase kompilácie? Aké funkcie použijeme?

Teoretické opakovanie – dynamická alokácia pamäte (heap)

- Používa sa na dynamickú alokáciu pamäte počas behu programu.
- Funkcie: `malloc, calloc, realloc`
- Výhody:
  - ⊕ Flexibilita v alokácii pamäte.
  - ⊕ Možnosť alokovať veľké bloky pamäte.
- Nevýhody:
  - ⊖ Pomalšia alokácia a dealokácia v porovnaní so stackom.
  - ⊖ Nutnosť manuálneho uvoľnenia pamäte pomocou free.

```c
/**
 * @brief Štruktúra reprezentujúca dynamické pole.
 */
typedef struct {
    int *items;          // Ukazovateľ na pole položiek.
    unsigned int size;   // Aktuálny počet položiek v poli.
    unsigned int capacity; // Kapacita poľa (maximálny počet položiek, ktoré môže pole obsahovať).
} dynamic_array_t;
```

Prečo je dôležité uvoľňovať dynamicky alokovanú pamäť?

Čo sa môže stať, ak zabudnete uvoľniť dynamicky alokovanú pamäť?

- Pri neuvoľnení pamäte vznikajú memory leaks, keď program alokuje pamäť, ale nikdy ju neuvoľní. To môže viesť k postupnému znižovaniu dostupnej pamäte a nakoniec k pádu programu.
- Funkcia `free(pointer);`
- Používa sa na uvoľnenie dynamicky alokovanej pamäte na heap.
  - Po uvoľnení pamäte je dobrým zvykom nastaviť ukazovateľ na NULL, aby sa predišlo výskytu pointerov s nedefinovaným chovaním.
  - Uvoľnenie pamäte dvakrát môže viesť k nepredvídateľnému správaniu programu.
- Dôsledky neuvoľnenia pamäte
  - Zníženiu výkonu programu – systém má menej dostupnej pamäte na ďalšie operácie.
  - Spotreba všetkej dostupnej pamäte môže viesť k pádu programu alebo k zlyhaniu celého systému !

Praktické cvičenie – dynamická pamäť

1. **Zadanie:** Napíšte funkciu mul, ktorá alokuje nové pole celých čísel a naplní ho násobkom dvoch vstupných polí rovnakej dĺžky.
2. **Zadanie:** Napíšte funkciu concatenate, ktorá alokuje nové pole znakov a naplní ho konkatenáciou dvoch reťazcov bez použitia funkcie strcat.
3. **Zadanie:** Napíšte funkciu concatenate_and_iterate, ktorá alokuje nové pole znakov a naplní ho konkatenáciou dvoch reťazcov bez použitia funkcie strcat, a následne tento reťazec iteruje x-krát.
4. **Zadanie:** Implementujte dynamické pole (vektor) v jazyku C, ktoré bude schopné dynamicky meniť svoju veľkosť pri pridávaní nových prvkov. Použite funkcie `malloc`, `realloc` a `free` na správu pamäte.
   - Vytvor štruktúru dynamického poľa a následne napíšte funkciu create_array, ktorá alokuje nové dynamické pole s počiatočnou kapacitou, taktiež napíšte funkciu `free_array`, ktorá uvoľní pamäť dynamického poľa a samotného poľa.
   - Napíšte funkciu `add_item`, ktorá pridá novú položku do dynamického poľa. Ak je pole plné, zdvojnásobí jeho kapacitu.
   - Napíšte funkciu `remove_item`, ktorá odstráni položku z dynamického poľa na zadanom indexe a posunie zvyšné položky.
   - Napíšte funkciu `find_item`, ktorá vyhľadá položku v dynamickom poli a vráti jej index. Ak položka nie je nájdená, vráti `-1`.