Dynamická alokace paměti

IZP-cv07

Ing. Jakub Husa Ph.D.

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

6. listopadu 2024

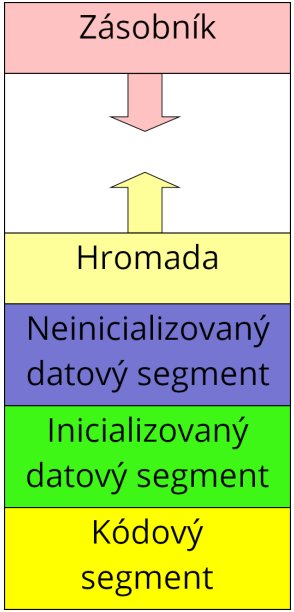
Organizace paměti

Paměťový prostor programu se skládá z pěti segmentů:

- Zásobník (stack) obsahuje automaticky alokované proměnné vytvářené uvnitř funkcí a parametry předávané při jejich volání, roste směrem shora dolů.
- Hromada (halda, heap) obsahuje dynamicky alokované proměnné a roste směrem zdola nahoru.
- Neinicializovaný datový segment obsahuje globální proměnné bez počáteční hodnoty.
- Inicializovaný datový segment obsahuje globální proměnné s počáteční hodnotou.
- Kódový segment obsahuje a zdrojový kód programu, a za běhu programu do něj nelze zapisovat.

Operační paměť počítače má omezenou velikost:

- Když se zásobník s hromadou potkají program havaruje!
- O ostatních segmentech se dozvíte více až v předmětu ISU.



Velikost datových typů

Různé datové typy zabírají v paměti počítače různé množství místa:

- Velikost zjistíme příkazem – `sizeof` – který vrací velikost v bajtech (byte).
- Velikost datových typů **NENÍ** pevně definována a závisí na operačním systému.

```
1 printf("int      = %i\n", sizeof(int));     //cele cislo            //4
2 printf("float    = %i\n", sizeof(float));   //desetinne cislo       //4
3 printf("double   = %i\n", sizeof(double));  //dvojnasobna presnost  //8
4 printf("char     = %i\n", sizeof(char));    //znak                  //1
```

Velikost ukazatelů (délka adres) je dána počtem bitů operačního systému:

```
5 printf("int*     = %i\n", sizeof(int));     //ukazatel na int       //8
6 printf("float*   = %i\n", sizeof(float));   //ukazatel na float     //8
7 printf("double*  = %i\n", sizeof(double));  //ukazatel na double    //8
8 printf("char*    = %i\n", sizeof(char));    //ukazatel na char      //8
```

Automatická a dynamická alokace paměti

Automaticky alokované proměnné na konci funkce přestávají existovat:

- Přístup k neexistující proměnné způsobí havárii na neplatný přístup do paměti.

Problém vyřešíme dynamickou alokací voláním funkce malloc z knihovny stdlib.h:

- Vstupem je velikost alokované paměti, výstupem je její adresa.
- Pokud alokace paměti selhala funkce vrátí konstantu NULL.

- Automatická alokace na zásobníku.

```
1  int *foo()     //"a" je automaticky
2  {              //alokovane cislo
3      int a;
4      a = 10;
5      return &a;//vracime adresu
6  }              //promenne "a"
7
8  int main()     //vypisujeme hodnotu
9  {              //smazane promenne
10     int *x = foo();
11     printf("%i\n", *x); //HAVARIE
12     return 0;
13 }
```

- Dynamická alokace na hromadě.

```
14 int *bar()     //"b" je ukazatel na
15 {              //alokovanou pamet
16     int* b = malloc(sizeof(int));
17     *b = 10;
18     return b; //vracime adresu
19 }              //alokovane pameti
20
21 int main()     //vypisujeme hodnotu
22 {              //z alokovane pameti
23     int *x = bar();
24     printf("%i\n", *x); //OK
25     return 0;
26 }
```

Uvolňování paměti

Dynamicky alokované proměnné budou v paměti existovat tak dlouho dokud je neuvolníme voláním funkce free z knihovny stdlib.h:

- Vstupem funkce je adresa nějaké dynamicky alokované paměti.
- Veškerou alokovanou paměť **MUSÍME** před koncem programu také uvolnit.
- Pokud paměť neuvolníme, nastává únik paměti.

```
1  int* x;                          //ukazatel na cele cislo
2  x = malloc(sizeof(int));         //alokujeme misto pro jedno cele cislo
3
4  if (x != NULL)                   //pokud alokace uspela (osetreni)
5  {
6      printf("Alokace pameti uspela\n"); //vypis
7      free (x);                    //uvolnujeme alokovanou pamet
8      return 0;                    //program konci bez chyby
9  }
10 else                             //jinak
11 {
12     fprintf(stderr, "Alokace pameti selhala\n"); //chybovy vypis
13     return 1;                    //program konci s chybou
14 }
```

Alokace polí

Velikost pole je násobkem počtu a velikosti jeho položek, protože položky pole jsou v paměti uloženy jako jeden souvislý blok dat:

- Hodnota pole (bez hranatých závorek) je adresou jeho prvního prvku.
- S ukazatelem tedy můžeme pracovat stejně jako by to bylo pole.

```
1  int main()                                  //zacatek programu
2  {
3      int* arr = malloc(3 * sizeof(int)); //alokujeme pole cisel
4      if (arr == NULL)                        //pokud alokace selhala
5          return 1;                           //program konci s chybou
6
7      for (int i = 0; i < 3; i++)             //pro vsechny prvky pole
8          arr[i] = 10 * (i+1);                //nastavujeme hodnoty 10,20,30
9
10     for (int i = 0; i < 3; i++)             //pro vsechny prvky pole
11         printf("arr[%i] = %i\n", i, arr[i]);  //vypis
12
13     free(arr);                              //uvolnujeme alokovanou pamet
14     return 0;                               //program konci bez chyby
15 }
```

Vyzkoušejte si:

- Napište funkci která dynamicky alokuje pole celých čísel, a naplní ho vektorovým součtem dvou polí stejné délky.
- Napište funkci která dynamicky alokuje celé číslo, a spočítá skalární součin dvou polí stejné délky.
- Napište funkci která dynamicky alokuje pole znaků, a (bez použití funkce strcat) provede konkatenaci dvou řetězců.

```
1 int* vektorovySoucet(int delka, int arr1[], int arr2[]);
2 int* skalarniSoucin(int delka, int arr1[], int arr2[]);
3 char* konkatenaceRetezcu(char str1[], char str2[]);
```

Například:

- (10 20 30, 40 50 60) => Soucet: 50 70 90
  => Soucin: 3200
- (”Hello”, ”Ahoj”) => Konkatenace: HelloAhoj

```c
int main() {      // ve VSCode formatovani spravime zkratkou "Shift + Alt + F"
    int arr1[3] = {10, 20, 30};                  // prvni pole
    int arr2[3] = {40, 50, 60};                  // druhe pole
    char str1[6] = "Hello";                      // prvni retezec
    char str2[5] = "Ahoj";                       // druhy retezec

    int* arr3 =vektorovySoucet(3, arr1, arr2);   // volame funkci soucet
    if (arr3 != NULL) {                          // pokud alokace uspela
        printf("Soucet: ");
        for (int i = 0; i < 3; i++)              // pro vsechny prvky pole
            printf("%i ", arr3[i]);              // vypisujeme soucty
        printf("\n");
    }

    int* soucin =skalarniSoucin(3, arr1, arr2);        // volame funkci
    if (soucin != NULL) printf("Soucin: %i\n", *soucin); // vypisujeme soucin

    char *str3 =konkatenaceRetezcu(str1, str2);        // volame funkci
    if (str3 != NULL) printf("Konkatenace: %s\n", str3); // vypisujeme konkatenaci

    if (arr3!= NULL) free(arr3);                 // uvolnujeme soucet
    if (soucin != NULL) free(soucin);            // uvolnujeme soucin
    if (str3!= NULL) free(str3);                 // uvolnujeme konkatenaci
    return 0;// konec programu
}
```

Alokace struktur

Velikost dynamicky alokované paměti nelze zjistit příkazem – `sizeof` –, protože příkaz vrátí velikost adresy, ne paměti na kterou ukazuje.

```
1  int arr1[10];                              //automaticky alokovane pole
2  int* arr2 = malloc(10 * sizeof(int));      //dynamicky   alokovane pole
3  printf("arr1 = %i\n", sizeof(arr1));       //velikost pole       //10 * 4 = 40
4  printf("arr2 = %i\n", sizeof(arr2));       //velikost ukazatele //8
```

Pole a jeho velikost (počet položek) můžeme spojit do jedné struktury:

```
5  typedef struct Smnozina //deklarujeme datovy typ pro strukturu
6  {                       //jmenem "Smnozina" se dvema polozkami
7      int delka;          //pocet polozek
8      int* pole;          //dynamicky alokovane pole
9  } mnozina;              //jmeno tohoto typu je "mnozina"
```

Velikost struktury může být větší než je součet velikostí jejích položek, protože položky struktury jsou v paměti automaticky zarovnávány na velikost slova:

```
10 typedef struct Styp {char x; int i;} typ; //struktura obsahujici
11                                           //znak (1) a cele cislo (4)
12 printf("typ  = %i\n", sizeof(typ));        // 1 + 4 + zarovnani = 8
```

Na procvičení (1/2)

Vyzkoušejte si:

- Implementujte následující funkce pro práci s množinami:

```
1 mnozina* alokujMnozinu(int delka);
2 void nactiMnozinu(mnozina* A);
3 void vypisMnozinu(mnozina* A);
4 bool jeMnozina(mnozina* A);
5 mnozina* konkatenaceMnozin(mnozina* A, mnozina* B);
6 void uvolniMnozinu(mnozina* A);
7 mnozina* prunikMnozin(mnozina* A, mnozina* B);
```

- Funkce alokujMnozinu která alokuje množinu, nastaví její délku, a alokuje pole.
- Funkce nactiMnozinu ze vstupu načte hodnoty a uloží je do pole množiny.
- Funkce vypisMnozinu hodnoty z pole množiny vypíše na výstup.
- Funkce jeMnozina ověří že se hodnoty v poli množiny neopakují.
- Funkce konkatenaceMnozin vytvoří novou množinu která bude konkatenací (ne sjednocením) dvou množin.
- Funkce uvolniMnozinu uvolní pole množiny a množinu.
- Funkce prunikMnozin vytvoří novu množinu která bude průnikem dvou množin.

Na procvičení (2/2)

Například:

- (10 20 30 40, 10 10 10) => Mnozina A:
  => 10 20 30 40
  => Mnozina B:
  => 10 10 10
  => Zadana pole nejsou mnozinami

- (10 20 30 40, 20 40 60) => Mnozina A:
  => 10 20 30 40
  => Mnozina B:
  => 20 40 60
  => Zadana pole jsou mnozinami
  => Konkatenace:
  => 10 20 30 40 20 40 60
  => Prunik:
  => 20 40

```c
int main() {   // ve VSCode formatovani spravime zkratkou "Shift + Alt + F"
    mnozina *A =alokujMnozinu(4);          // alokujeme mnozinu A (a jeji pole)
    mnozina *B =alokujMnozinu(3);          // alokujeme mnozinu B (a jeji pole)
    if (A ==NULL ||A->pole ==NULL ||B ==NULL ||B->pole ==NULL) {// osetreni alokace
        printf("Alokace selhala\n"); return 1;}

    nactiMnozinu(A);                       // do mnoziny A nacitame vstup
    nactiMnozinu(B);                       // do mnoziny B nacitame vstup
    printf("Mnozina A:\n"); vypisMnozinu(A); // vypisujeme mnozinu A
    printf("Mnozina B:\n"); vypisMnozinu(B); // vypisujeme mnozinu A

    if (!jeMnozina(A) ||!jeMnozina(B)) {   // osetreni vstupu
        printf("Zadana pole nejsou mnozinami\n"); return 2;}
    printf("Zadana pole jsou mnozinami\n");

    mnozina *C =konkatenaceMnozin(A, B);   // C je konkatenaci A a B
    if (C ==NULL ||C->pole ==NULL) {       // osetreni alokace
        printf("Alokace konkatenace selhala\n"); return 3;}
    printf("Konkatenace:\n"); vypisMnozinu(C); // vypisujeme mnozinu C
    uvolniMnozinu(C);                      // uvolnujeme mnozinu C

    C = prunikMnozin(A, B);                // C je prunikem A a B
    if (C ==NULL ||C->pole ==NULL) {       // osetreni alokace
        printf("Alokace pruniku selhala\n"); return 4;}
    printf("Prunik:\n"); vypisMnozinu(C); // vypisujeme mnozinu C

    uvolniMnozinu(C); uvolniMnozinu(B); uvolniMnozinu(A); // uvolnujeme mnoziny
    return 0;
}
```