Dynamické přidělování paměti

Petr Šaloun

katedra informatiky FEI VŠB-TU Ostrava

7. listopadu 2011

Dynamické a statické přidělování paměti

- **Statická data** (SD) – velikost určena v okamžiku překladu zdrojového textu. Hodnota SD se během programu může měnit. Velikost SD v programu = součet všech statických datových položek.
- **Dynamická data** (DD) – o paměť žádáme OS až za chodu programu, při překladu jsou vytvořeny pouze proměnné vhodného typu *ukazatel na*, které za chodu slouží jako pevné body pro práci s DD. Požadavek na DD v programu $\leq$ součet všech dynamických datových položek
(znovuvyužití dynamicky přidělené paměti).

Paměť programu za chodu

- **Kódový segment** obsahuje kód vytvořený překladem zdrojového textu, za chodu programu se nemění.
- **Datový segment** je vytvořen překladačem, obsahuje *statická data*, během chodu programu se mění.
- **Zásobník** je nezbytný při předávání argumentů funkcím, návratové hodnoty od funkcí a také pro nalezení návratové adresy po ukončení funkce (zapisuje se při volání funkce). Na zásobníku jsou uloženy *hodnoty (automatických) proměnných*.
- **Halda** (heap) je oblast paměti, kam se umisťují *dynamická data*, zpravidla lze velikost haldy měnit.
  Problémem je *segmentace haldy*. V jazyce C není *garbage collector* – automatický proces procházející dynamickou paměť; objekt bez odkazu uvolní. Programátor – paměť haldy používá, správné uvolňování – běhová podpora.

Správce haldy – (heap manager)

Rozhraní pro dynamickou alokaci paměti jsou `stdlib.h` a `alloc.h`.

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

požadavek o přidělení souvislého bloku paměti o velikosti `size`. Při neúspěchu `NULL`.

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

jako malloc() jen `nitems` položek, každá `size` bajtů, přidělená paměť je vyplněna nulami.

```
void free(void *block)
```

vrácení dříve alokované paměti, na kterou ukazuje `block`.

```
void *realloc(void *block, size_t size)
```

změna velikosti alokované paměti, na kterou ukazuje `block` na novou velikost určenou hodnotou `size`. V případě potřeby (požadavek je větší, než původní blok) je obsah původního bloku překopírován. Vrací ukazatel na nový blok.

Ukázka vytvoření kopie řetězce, uvolnění řetězce

```
char *newstr(char *p)
{
  register char *t;
  t = malloc(strlen(p) + 1);
  strcpy(t, p);
  return t;
}

void freestr(char *p)
{
  free(p);
}
```

Příklad – dynamické pole ukazatelů `arry.c`

Načtěme vstupní řádky textu na haldu, přístup pomocí pole ukazatelů. Načtené řádky program vytiskne ve stejném pořadí, v jakém byly načteny. Hodnoty `START` a `PRIRUSTEK`, jsou úmyslně voleny malé, aby se ukázala možnost realokace pole podle skutečného počtu řádků.

```
int main(void)
{
 int      pocet      = 0,
          alokovano  = START,
          prirustek  = PRIRUSTEK;
 pole_retezcu p_ret = NULL;

 if (alokace(&p_ret, alokovano))
   {
    puts("nedostatek pameti");
    return 1;
   }
 if (cti_a_pridavej(&p_ret, &pocet, &alokovano, prirus
   return 1;
 zobrazuj(p_ret, pocet);

 return 0;
} /* int main(void) */
```

```c
/* soubor str_arry.c v ukázce kráceno */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

const int START = 2;
const int PRIRUSTEK = 1;
const int DELKA_RADKU = 100;

typedef char *     retezec;
typedef retezec * pole_retezcu;
```

```
void uvolni(pole_retezcu *p_r, int pocet) {
 int i;
 for (i = 0; i < pocet; i++)     {
     free((*p_r)[i]);
   }
 free(*p_r);
} /* void uvolni(pole_retezcu *p_r, int pocet) */
```

```
int alokace(pole_retezcu *p_r, int pocet) {
 *p_r = malloc(pocet * sizeof(retezec));
 return (*p_r == NULL) ? -1 : 0;
} /* int alokace(pole_retezcu *p_r, int pocet) */

int re_alokace(pole_retezcu *p_r, int novy_pocet) {
 pole_retezcu pom;
 if (*p_r == NULL)
   if (alokace(p_r, novy_pocet))
     return -1; /* chyba */
 pom = realloc(*p_r, sizeof(retezec) * novy_pocet);
 if (pom == NULL)
     return -1;
 else        {
      *p_r = pom; return 0;
 }
} /* int re_alokace(pole_retezcu *p_r, int novy_pocet)
```

```
int pridej_radek(pole_retezcu *p_r, retezec s, int ind
{
 int delka = strlen(s) + 1;
 if (((*p_r)[index] = malloc(delka)) == NULL)
   return -1;
 strcpy((*p_r)[index], s);
 return 0;
} /* int pridej_radek(pole_retezcu *p_r, retezec s, in
```

```
int cti_a_pridavej(pole_retezcu *p_r, int *pocet,
    int *alokovano, int prir) {
 char radek[DELKA_RADKU], *pom;
 puts("Zadavej retezce, posledni CTRL-Z na novem radku
 do{if ((pom = gets(radek)) != NULL) {
       if (*pocet + 1 > *alokovano)  {
          if (re_alokace(p_r, *alokovano + prir))  {
             puts("nedostatek pameti");
             return -1;}
          *alokovano += prir;
         }
       if (pridej_radek(p_r, radek, *pocet))  {
          puts("nedostatek pameti");
          return 1; }
       (*pocet)++;
      }
   } while (pom != NULL);
 return 0;
```

```
void zobrazuj(pole_retezcu p_r, int pocet)
{
 while (pocet--)
    {
     puts(*p_r++);
    }
} /* void zobrazuj(pole_retezcu p_r, int pocet) */
```