Wpisy zaliczeń: 18.02.2011, godz. 12:00, sala 503

Język ANSI C

część 11

tablice 2D, alokacja pamięci

Jarosław Gramacki
Instytut Informatyki i Elektroniki

Alokacja pamięci, stos i sterta

- Było:
- Stałe, zmienne statyczne i zewnętrzne mają zarezerwowane miejsce w kodzie wykonywalnym programu (są umieszczane w obszarze danych programu).
- Zmienne lokalne (automatyczne) funkcji są umieszczane na **stosie** w momencie, gdy sterowanie wejdzie do bloku, w którym zostały zdefiniowane. Zmienne tej klasy znikają po wyjściu sterowania z bloku (są usuwane ze stosu).
- Zmienne dynamiczne są umieszczane na **stercie**.
- W języki C niezmiernie często zachodzi potrzeba dynamicznej alokacji pamięci

Deklaracje i opis

```
SKLADNIA
  #include <stdlib.h>

  void *calloc(size_t nmemb, size_t size);
  void *malloc(size_t size);
  void *realloc(void *ptr, size_t size);
  void free(void *ptr);
```

zwróć uwagę na "wskaźnik do czegokolwiek" (wskaźnik na void). W trakcie wywołania funkcji przekażemy właściwy typ

kopia z systemowego manuala

```
OPIS
       calloc()  przydziela  pamięć  dla tablicy zawierającej nmemb elementów,
       każdy o rozmiarze  size  bajtów  i  zwraca  wskaźnik  do  przydzielonej
       pamięci.  Pamięć jest zerowana.

       malloc()  przydziela  size  bajtów  i  zwraca wskaźnik do przydzielonej
       pamięci.  Pamięć nie jest czyszczona.

       free() zwalnia  obszar  pamięci  wskazywany  przez  ptr,  który  został
       wcześniej przydzielony za pomocą malloc(), calloc() lub realloc().
       W przeciwnym przypadku, lub gdy free(ptr) zostało już wcześniej
       wywołane,  funkcja  zachowa  się w sposób nieokreślony.  Jeśli ptr jest
       równe NULL, nie zostanie wykonana żadna operacja.

       realloc() zmienia rozmiar bloku pamięci wskazywanego przez ptr na  size
       bajtów.  Zawartość nie zostanie zmieniona w zakresie poniżej minimum ze
       starego i  nowego  rozmiaru;  nowo  przydzielona  pamięć  nie  zostanie
       zainicjalizowana.  Jeśli  ptr  jest  równe  NULL,  to  wywołanie  jest
       równoważne malloc(size); jeśli size jest równe zeru, to wywołanie  jest
       równoważne  free(ptr).  O  ile  ptr  nie jest równe NULL, musi ono być
       zwrócone przez wcześniejsze wywołanie malloc(), calloc() lub realloc().
```


Deklaracje i opis

```
WARTOŚĆ ZWRACANA
       Dla  calloc()  i  malloc(),  zwracana wartość jest wskaźnikiem do przy-
       dzielonej pamięci, który jest właściwie wyrównany dla dowolnego rodzaju
       zmiennej, lub NULL gdy żądanie zakończyło się niepowodzeniem.

       free() nie zwraca żadnej wartości.

       realloc()  zwraca  wskaźnik  do  nowoprzydzielonej  pamięci, który jest
       właściwie wyrównany dla dowolnego rodzaju zmiennej i może być różny  od
       ptr,  lub  NULL  gdy żądanie zakończy się niepowodzeniem. Jeśli rozmiar
       był równy 0, zwracane jest NULL lub wskaźnik odpowiedni do  przekazania
       go  funkcji  free().  Gdy realloc() zakończy się niepowodzeniem, pier-
       wotny blok zostaje nienaruszony – nie jest on ani zwalniany  ani  prze-
       suwany.
```

– ... tyle teorii

Tablice wskaźników, malloc()

```
// allocates memory using malloc(), checks for error return
#include <stdlib.h>                          // for malloc() and exit()

int main(void) {
   unsigned mem_ints;              // number of ints to store

   int *ptr;                       // pointer to start of memory
   unsigned j;

   printf("\nEnter number of integers to store: "); scanf("%u", &mem_ints);

   ptr = (int*) malloc(mem_ints * sizeof(int) );
   if(ptr == NULL) {
      printf("Can't allocate memory");
      exit(1);
      }
   for(j=0; j<mem_ints; j++)                          // fill memory with data
      ptr[j] = j;
   for(j=0; j<mem_ints; j++)                          // check the data
      if( ptr[j] != j ) {
          printf("Memory error: j=%u, ptr[j]=%u", j, ptr[j] );
          exit(1);
          }
   free(ptr);     // free memory
   printf("Memory test successful");
   return 1;
   }
```

Jeśli masz w systemie 512MB to podaj: 50000000 =~ 200MB

lub `#include <assert.h>` i dalej: `assert( ptr != NULL);`

przećwicz działanie assert doprowadzając do błędu (tu: alokacji pamięci)

Funkcja assert()

```
#define NDEBUG  // wylacza tryb debug
#include <assert.h>

int main(void)
{
  assert(1==2);
}
```

```
#define DEBUG  // wlacza tryb debug
#include <assert.h>

int main(void)
{
  assert(1==2);
}
```

```
C:\Temp>Project1.exe
Assertion failed: 1==2, file main.c, line 6

This application has requested the Runtime to terminate it in an unusual
way. Please contact the application's support team for more information.
```

Tablice wskaźników, realloc()

```
// reallocates memory using realloc()
#include <stdlib.h>
#define CHUNK 10

int main(void) {
   int *ptr = NULL;                 // pointer to memory
   int j, n = 0;                    // number of data items stored
   int total_bytes = 0;             // total bytes allocated

   printf("\n");
   do
      {
      if( n*sizeof(int) >= total_bytes )          // if no more memory,
          {
          ptr = (int*) realloc(ptr, CHUNK);       // get another chunk
          if(ptr == NULL) {
             printf("\nCan't reallocate memory"); exit(1); }
          total_bytes += CHUNK;
          printf("Total bytes allocated = %d\n", total_bytes );
          }
      printf("Enter a number: ");                 // get data from user
      scanf( "%d", &ptr[n] );                     // store in memory
      }
   while(ptr[n++] != 0);                          // user enters 0

   for(j=0; j<n-1; j++)                           // display the data
      printf("%7d ", ptr[j]);   free(ptr); return 1;}
```

W praktyce 10 bajtów wystarczy na 2.5 liczby int. Wiec co dwie wprowadzone liczby trzeba doalokować pamięci

gdyby nie inicjalizacja NULL przy deklaracji to wcześniej trzeba by użyć malloc() lub calloc()

używamy "zwykłego" operatora []

Tablice wskaźników, malloc() + realloc()

- podajemy z klawiatury kolejno napisy. Zakończenie programu po podaniu "quit"
- program za każdym razem doalokowuje tyle pamięci aby zmieścić wszystkie wprowadzone napisy oraz ustalony ich separator (znak '-')

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
char *DodajDo(char *p, char *str);

int main (void)
{
  char buf[30], *ptr = NULL;
  do
  {
    printf("\nPodaj napis: ");
    gets(buf);
    if (ptr = DodajDo(ptr, buf))
      printf("\n %s", ptr);
    else printf("Brak pamięci");
  }
  while (strcmp(buf, "quit"));     // popraw błąd: quit też się wyświetli
  getchar();
  free(ptr);
}
                          c.d.n.
```

tu będzie znajdował się "rosnący" w miarę podawania kolejnych danych napis

będzie ważne przy pierwszym wywołaniu funkcji DodajDo()

Tablice wskaźników, malloc() + realloc()

- podajemy z klawiatury kolejno napisy. Zakończenie programu po podaniu "quit"
- program za każdym razem doalokowuje tyle pamięci aby zmieścić wszystkie wprowadzone napisy oraz ustalony ich separator (znak '-')

```
c.d.

// implementacja funkcji DodajDo()
char *DodajDo(char *p, char *str)
{
  if ( !p )                                  // spr. czy string istnieje
    if ( !(p = (char*)malloc(strlen(str) + 1)))     // +1 na znak '-'
      return NULL;
    else strcpy(p, str);                     // zapełnienie bufora
  else
    if ( !(p = (char*)realloc(p, strlen(p) + strlen(str) + 2)))
    // +2 gdyż separator
      return NULL;
    else
      strcat(strcat(p, "-"), str); // dodanie separat. i nowego napisu
  return p;
}
```

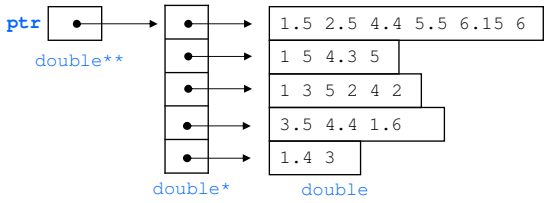
porównaj przykład "mapa pamięci" z wcześniejszego wykładu

zwracamy wskaźnik do alokowanej w funkcji pamięci

zwróć uwagę na kaskadowe wywołania strcat()

Tablice wskaźników, alokacja pamięci dla tablicy 2D

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

int main(void) {
  int i, j,
  r, c;  // rows, columns

  double **ptr;

  printf("Enter # of rows and # of cols: ");
  scanf("%d %d", &r, &c); // how many rows and columns ?
  // 1. etap
  ptr = (double**) malloc ( r * sizeof(double*) );
  assert (ptr != NULL);

  // 2. etap
  for (i=0; i < r; i++)
    ptr[i] = (double*) malloc ( c * sizeof(double) );
    assert (ptr != NULL);

  for (i=0; i < r; i++)
    for (j=0; j < c; j++) {
      printf("Enter elements %d %d: ", i, j);       // read values to a matrix
      scanf("%lf", &ptr[i][j]);
    }
  fun (ptr, r, c);  // argument przekazany "jak zwykle"
                                  c.d.n.
```



proces 2-etapowy

lub indywidualna wartość dla każdego wiersza

używamy "zwykłego" operatora [][]

Tablice wskaźników, alokacja pamięci dla tablicy 2D

```
c.d.

for (i=0; i < r; i++)     // free memory in reverse order
    free(ptr[i]);
  free(ptr);

return 1;
}                         // end main


void fun( double **tab, int r, int c)
{
int i, j;

for (i=0; i < r; i++)
  for (j=0; j < c; j++)
  printf("%f ", tab[i][j];
}
```

używamy "zwykłego" operatora [][]

Tablice wskaźników, alokacja pamięci dla tablicy 2D

– inne sposoby alokacji pamięci dla tablic "typu 2D" - następny wykład