Alocarea dinamică a memoriei. Aplicaţii folosind tablouri şi matrice.

Responsabili:

- Darius Neațu (CA 2019-2020) [mailto:neatudarius@gmail.com]
- Dorinel Filip (CA 2019-2020) [mailto:ion_dorinel.filip@cti.pub.ro]
- Laura Vasilescu [mailto:laura.vasilescu@cti.pub.ro]
- George Muraru [mailto:murarugeorgec@gmail.com]

Obiective

În urma parcurgerii acestui laborator studentul va fi capabil:

- să aloce dinamic o zona de memorie;
- să elibereze o zona de memorie;
- să lucreze cu vectori şi matrice alocate dinamic.

Funcţii de alocare şi eliberare a memoriei

Funcțiile standard de alocare și de eliberare a memoriei sunt declarate în fişierul antet `stdlib.h`.

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

Alocarea memoriei

Cele trei funcţii de alocare (`malloc`, `calloc` și `realloc`) au ca rezultat adresa zonei de memorie alocate (de tip `void*`) şi ca argument comun dimensiunea, în octeţi, a zonei de memorie alocate (de tip `size_t`). Dacă cererea de alocare nu poate fi satisfăcută pentru că nu mai există un bloc continuu de dimensiunea solicitată, atunci funcţiile de alocare au rezultat `NULL` (ce reprezintă un pointer de tip `void *` la adresa de memorie 0, care prin convenţie este o adresă nevalidă - nu există date stocate în acea zonă).

Exemplu

```
char *str = malloc(30);            // Aloca memorie pentru 30 de caractere
int *a = malloc(n * sizeof(int)); // Aloca memorie pt. n numere intregi
```

Dimensiunea memoriei luată ca parametru de `malloc()` este specificată în `octeţi`, indiferent de tipul de date care va fi stocat în acea regiune de memorie! Din acest motiv, pentru a aloca suficientă memorie, numărul dorit de elemente trebuie înmulţit cu dimensiunea unui element, atunci când are loc un apel `malloc()`.

Alocarea de memorie pentru un vector şi iniţializarea zonei alocate cu zerouri se poate face cu funcţia `calloc`.

Exemplu

```
int *a = calloc(n, sizeof(int)); // Aloca memorie pentru n numere intregi și inițializează zona cu zero
```

Codul de mai sus este perfect echivalent (dar mai rapid) cu următoarea secvenţă de instrucţiuni:

```
int i;
int *a = malloc(n * sizeof(int));
for (i = 0; i < n; i++) {
    a[i] = 0;
}
```

În timp ce funcţia `malloc()` ia un singur parametru (o dimensiune în octeţi), funcţia `calloc()` primeşte două argumente, o lungime de vector şi o dimensiune a fiecărui element. Astfel, această funcţie este specializată pentru memorie organizată ca un vector, în timp ce `malloc()` nu ţine cont de structura memoriei.

Acest lucru aduce și o măsură de siguranță în plus deoarece înmulțirea dintre numărul de elemente și dimensiunea tipului de date ar putea face overflow, iar dimensiunea memoriei alocate să nu fie în realitatea cea așteptată.

Realocarea unui vector care creşte (sau scade) faţă de dimensiunea estimată anterior se poate face cu funcţia `realloc`, care primeşte adresa veche şi noua dimensiune şi întoarce **noua adresă**:

```
int *aux;
aux = realloc(a, 2 * n * sizeof(int)); // Dublare dimensiune anterioara (n)
if (aux) //daca aux este diferit de NULL
    a = aux;
else
    //prelucrare in caz de eroare
```

În exemplul anterior, noua adresă este memorată tot în variabila pointer `a`, înlocuind vechea adresă (care nu mai este necesară şi nici nu mai trebuie folosită). Funcţia `realloc()` realizează următoarele operaţii:

- alocă o zonă de dimensiunea specificată ca al doilea argument
- copiază la noua adresă datele de la adresa veche (primul argument al funcţiei)
- eliberează memoria de la adresa veche.

In cazul in care nu reuseste sa aloce memorie functia `realloc()` intoarce NULL lasand pointerul initial nemodificat. Din acest motiv verificam daca realocarea a reusit si apoi asignam rezultatul pointerului `a`.

Eliberarea memoriei

Funcţia `free()` are ca argument o adresă (un pointer) şi eliberează zona de la adresa respectivă (alocată prin apelul unei funcţii de tipul `[m|c|re]alloc`). Dimensiunea zonei nu mai trebuie specificată deoarece este ţinută minte de sistemul de alocare de memorie în nişte structuri interne.

Vectori alocaţi dinamic

Structura de vector are avantajul simplităţii şi economiei de memorie faţă de alte structuri de date folosite pentru memorarea unei colectii de informaţii între care există anumite relaţii. Între cerinţa de dimensionare constantă a unui vector şi generalitatea programelor care folosesc astfel de vectori există o contradicţie. De cele mai multe ori programele pot afla (din datele citite) dimensiunile vectorilor cu care lucrează şi deci pot face o alocare dinamică a memoriei pentru aceşti vectori. Aceasta este o solutie mai flexibilă, care foloseşte mai bine memoria disponibilă şi nu impune limitări arbitrare asupra utilizării unor programe. În limbajul C nu există practic nici o diferenţă între utilizarea unui vector cu dimensiune fixă şi utilizarea unui vector alocat dinamic, ceea ce încurajează şi mai mult utilizarea unor vectori cu dimensiune variabilă.

Exemplu

```
int main(void)
{
```

```
  int n,i;
  int *a;                     // Adresa vector alocat dinamic

  printf("n = ");
  scanf("%d", &n);            // Dimensiune vector

  a = calloc(n, sizeof(int)); // Alternativ: a = malloc(n * sizeof(int));
  printf("Componente vector: \n");

  for (i = 0; i < n; i++) {
    scanf("%d", &a[i]);       // Sau scanf (“%d”, a+i);
  }
  for (i = 0; i < n; i++) {    // Afisare vector
    printf("%d ",a[i]);
  }

  free(a);                    // Nu uitam sa eliberam memoria

  return 0;
}
```

Puteti testa codul aici [http://tpcg.io/BZeAR5]. Trebuie introdus in tabul de STDIN inputul.

Există şi cazuri în care datele memorate într-un vector rezultă din anumite prelucrări, iar numărul lor nu poate fi cunoscut de la începutul execuţiei. Un exemplu poate fi un vector cu toate numerele prime mai mici ca o valoare dată. În acest caz se poate recurge la o realocare dinamică a memoriei. În exemplul următor se citeşte un număr necunoscut de valori întregi într-un vector extensibil:

```
#define INCR 100 // cu cat creste vectorul la fiecare realocare

int main(void)
{
  int n, i, m;
  float x, *v, *tmp;                  // v = adresa vector

  n = INCR;
  i = 0;

  v = malloc(n * sizeof(float)); // Dimensiune initiala vector
  if (v == NULL) {
      /* Nu s-a reusit alocarea */
      printf("Could not allocate v\n");
      return 1;
  }

  while (scanf("%f", &x) != EOF) {
    if (i == n) {              // Daca este necesar...
      n = n + INCR;              // ... creste dimensiune vector
      tmp = realloc(v, n * sizeof(float));
      if (tmp != NULL) {
          /* Daca s-a reusit alocarea pentru noua zona de memorie */
          v = tmp;
      } else {
          /* Daca nu s-a reusit alocarea */
          break;
      }
    }

    v[i++] = x;                    // Memorare in vector numar citit
  }

  m = i;

  for (i = 0; i < m; i++) {       // Afisare vector
    printf("%f ", v[i]);
  }
  printf("\n");

  free(v);

  return 0;
}
```

Puteti testa codul aici [http://tpcg.io/KSDiLd]. Trebuie introdus in tabul de STDIN inputul.

Matrice alocate dinamic

Alocarea dinamică pentru o matrice este importantă deoarece:

- foloseşte economic memoria şi evită alocări acoperitoare, estimative.
- permite matrice cu linii de lungimi diferite (denumite uneori `ragged arrays`, datorită formelor "zimţate" din reprezentările grafice)
- reprezintă o soluţie bună la problema argumentelor de funcţii de tip matrice.

Daca programul poate afla numărul efectiv de linii şi de coloane al unei matrice (cu dimensiuni diferite de la o execuţie la alta), atunci se va aloca memorie pentru un vector de pointeri (funcţie de numărul liniilor) şi apoi se va aloca memorie pentru fiecare linie (funcţie de numărul coloanelor) cu memorarea adreselor liniilor în vectorul de pointeri. O astfel de matrice se poate folosi la fel ca o matrice declarată cu dimensiuni constante.

Exemplu

```
int main (void)
{
  int **a;
  int i, j, nl, nc;

  printf("nr. linii = ");
  scanf("%d", &nl);

  printf("nr. coloane = ");
  scanf("%d", &nc);

  /*
   * In cele ce urmeaza presupunem ca toate apelurile de alocare de memorie
   * nu vor esua.
   */
  a = malloc(nl * sizeof(int *));   // Alocare pentru vector de pointeri

  for (i = 0; i < nl; i++) {
    a[i] = calloc(nc, sizeof(int)); // Alocare pentru o linie si initializare la zero
  }

  // Completare diagonala matrice unitate
  for (i = 0; i < nl; i++) {
    a[i][i] = 1;                    // a[i][j]=0 pentru i != j


  // Afisare matrice
  printmat(a, nl, nc);

  for (i = 0; i < nl; i++)
     free(a[i]);
  free(a);                         // Nu uitam sa eliberam!

  return 0;
}
```

Funcţia de afişare a matricei se poate defini astfel:

```
void printmat(int **a, int nl, int nc) {
  for (i = 0; i < nl; i++) {
    for (j = 0; j < nc; j++) {
      printf("%2d", a[i][j]);
    }

    printf("\n");
  }
}
```

Notaţia `a[i][j]` este interpretată astfel pentru o matrice alocată dinamic:

- `a[i]` conţine un pointer (o adresă `b`)
- `b[j]` sau `b+j` conţine întregul din poziţia `j` a vectorului cu adresa `b`.

Astfel, `a[i][j]` este echivalent semantic cu expresia cu pointeri `*(*(a + i) + j)`.

Totuşi, funcţia `printmat()` dată anterior nu poate fi apelată dintr-un program care declară argumentul efectiv ca o matrice cu dimensiuni constante. Exemplul următor este corect sintactic dar nu se execută corect:

```
int main() {
  int x[2][2] = { {1, 2}, {3, 4} }; // O matrice patratica cu 2 linii si 2 coloane
  printmat((int**)x, 2, 2);
  return 0;
}
```

Explicaţia este interpretarea diferită a conţinutului zonei de la adresa aflată în primul argument: funcţia `printmat()` consideră că este adresa unui vector de pointeri (`int *a[]`), iar programul principal consideră că este adresa unui vector de vectori (`int x[][2]`), care este reprezentat liniar in memorie.

Se poate defini şi o funcţie pentru alocarea de memorie la execuţie pentru o matrice.

Exemplu

```
int **newmat(int nl, int nc) { // Rezultat adresa matrice
  int i;
  int **p = malloc(nl * sizeof(int *));

  for (i = 0; i < n; i++) {
    p[i] = calloc(nc, sizeof(int));
  }

  return p;
}
```

Stil de programare

Exemple de programe

`Exemplul 1`: Funcţie echivalentă cu funcţia de bibliotecă `strdup()`:

```
// Alocare memorie si copiere sir
char *mystrdup(char *adr)
{
  int len = strlen(adr);
  char *rez = malloc(len + 1); // len+1, deoarece avem si un '\0' la final

  /* Daca alocarea nu a reusit, intorcem NULL */
  if(rez == NULL)
    return NULL;

  strcpy(rez, adr);

  return rez;
}
```

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

// Utilizare "mystrdup"
```

```
int main(void)
{
  char s[80], *d;

  do {
    if (fgets(s, 80, stdin) == NULL) {
      break;
    }

    d = mystrdup(s);
    if (d != NULL) {
      /* Nu s-a reusit alocarea de memorie */
      fputs(d, stdout);
      free(d);
    } else {
      printf("Nu s-a reusit alocarea\n");
      return 1;
    }
  } while (1);

  return 0;
}
```

Puteti testa codul aici [http://tpcg.io/eaSDUd].

In exemplele urmatoare consideram ca toate alocarile de memorie nu vor esua.

Exemplul 3: Vector realocat dinamic (cu dimensiune necunoscută)

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

#define INCR 4

int main(void)
{
  int n, i, m;
  float x, *v;

  n = INCR;
  i = 0;

  v = malloc(n * sizeof(float);

  while (scanf("%f", &x) != EOF) {
    if (i == n) {
      n = n + INCR;
      v = realloc(v, n * sizeof(float);
    }

    v[i++] = x;
  }

  m = i;

  for (i = 0; i < m; i++) {
    printf("%.2f ", v[i]);
  }
  printf("\n");

  free(v);

  return 0;
}
```

Puteti testa codul aici [http://tpcg.io/q0djWl].

Exemplul 4: Matrice alocată dinamic (cu dimensiuni necunoscute la execuţie)

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

```
int main(void)
{
  int n, i, j;
  int **mat; // Adresa matrice

  // Citire dimensiuni matrice
  printf("n = ");
  scanf("%d", &n);

  // Alocare memorie ptr matrice
  mat = malloc(n * sizeof(int *));

  for (i = 0; i < n; i++) {
    mat[i] = calloc(n, sizeof(int));
  }

  // Completare matrice
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      mat[i][j] = n * i + j + 1;
    }
  }

  // Afisare matrice
  for (i = 0; i < n; i++) {
    for (j = 0;j < n; j++) {
      printf("%6d", mat[i][j]);
    }

    printf("\n");
  }

  return 0;
}
```

Puteti testa codul aici [http://tpcg.io/Z5DeQC].

Exemplul 5: Vector de pointeri la şiruri alocate dinamic

```
/* Creare/afisare vector de pointeri la siruri */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Afisare siruri reunite in vector de pointeri
void printstr(char *vp[], int n)
{
  int i;

  for (i = 0; i < n; i++) {
    printf("%s\n", vp[i]);
  }
}

// Ordonare vector de pointeri la siruri
void sort(char *vp[], int n)
{
  int i, j;
  char *tmp;

  for (j = 1; j < n; j++) {
    for (i = 0; i < n - 1; i++) {
      if (strcmp(vp[i], vp[i+1]) > 0) {
        tmp = vp[i];
        vp[i] = vp[i+1];
        vp[i+1] = tmp;
      }
    }
  }
}
```

```
// Citire siruri si creare vector de pointeri
int readstr (char * vp[])
{
  int n = 0;
  char *p, sir[80];

  while (scanf("%s", sir) == 1) {
    p = malloc(strlen(sir) + 1);
    strcpy(p, sir);
    vp[n] = p;
    ++n;
  }

  return n;
}

int main(void)
{
  int n;
  char *vp[1000]; // vector de pointeri, cu dimensiune fixa

  n = readstr(vp); // citire siruri si creare vector
  sort(vp, n); // ordonare vector
  printstr(vp, n); // afisare siruri

  return 0;
}
```

Puteti testa codul aici [http://tpcg.io/2nn7vJ].

Practici recomandate

Deşi au fost enunţate în momentul în care au fost introduse noţiunile corespunzătoare în cursul acestui material, se pot rezuma câteva reguli importante de folosire a variabilelor de tip pointer:

- Aveţi grijă ca variabilele de tip pointer să indice către adrese de memorie valide înainte de a fi folosite; consecinţele adresării unei zone de memorie aleatoare sau nevalide (NULL) pot fi dintre cele mai imprevizibile.
- Utilizaţi o formatare a codului care să sugereze asocierea operatorului * cu variabila asupra căreia operează; acest lucru este în special valabil pentru declaraţiile de pointeri.
- Nu returnaţi pointeri la variabile sau tablouri definite în cadrul funcţiilor, întrucât valabilitatea acestora încetează odată cu ieşirea din corpul funcţiei.
- Verificaţi rezultatul funcţiilor de alocare a memoriei, chiar dacă dimensiunea pe care doriţi s-o rezervaţi este mică. Atunci când memoria nu poate fi alocată rezultatul este NULL iar programul vostru ar trebui să trateze explicit acest caz (finalizat, de obicei, prin închiderea "curată" a aplicaţiei).
- Nu uitaţi să eliberaţi memoria alocată dinamic, folosind funcţia `free()`. Memoria rămasă neeliberată încetineşte performanţele sistemului şi poate conduce la erori (bug-uri) greu de depistat.

Studiu de caz

Această secţiune este opţională şi nu este necesară pentru rezolvarea exerciţiilor de laborator, însă ajută la înţelegerea aprofundată a modului în care limbajul C lucrează cu variabilele.

Clasa de stocare (memorare) arată când, cum şi unde se alocă memorie pentru o variabilă (vector). Orice variabilă C are o clasă de memorare care rezultă fie dintr-o declaraţie explicită, fie implicit din locul unde este definită variabila. Există trei moduri de alocare a memoriei, dar numai două corespund unor clase de memorare:

- `Static`: memoria este alocată la compilare în segmentul de date din cadrul programului şi nu se mai poate modifica în cursul execuţiei. Variabilele externe, definite în afara funcţiilor, sunt implicit statice, dar pot fi declarate static şi variabile locale, definite în cadrul funcţiilor.
- `Automat`: memoria este alocată automat, la activarea unei funcţii, în zona stivă alocată unui program şi este eliberată automat la terminarea funcţiei. Variabilele locale unui bloc (unei funcţii) şi argumentele formale sunt implicit din clasa `auto`.
- `Dinamic`: memoria se alocă la execuţie în zona `heap` alocată programului, dar numai la cererea explicită a programatorului, prin apelarea unor funcţii de bibliotecă (`malloc`, `calloc`, `realloc`). Memoria este eliberată numai la cerere, prin apelarea funcţiei `free`. Variabilele dinamice nu au nume şi deci nu se pune problema clasei de memorare (atribut al variabilelor cu nume).

Variabilele statice pot fi iniţializate numai cu valori constante (pentru că se face la compilare), dar variabilele auto pot fi iniţializate cu rezultatul unor expresii (pentru că se face la execuţie). Toate variabilele externe (şi statice) sunt automat iniţializate cu valori zero (inclusiv vectorii). Cantitatea de memorie alocată pentru variabilele cu nume rezultă automat din tipul variabilei şi din dimensiunea declarată pentru vectori. Memoria alocată dinamic este specificată explicit ca parametru al funcţiilor de alocare.

O a treia clasă de memorare este clasa `register` pentru variabile cărora, teoretic, li se alocă registre ale procesorului şi nu locaţii de memorie, pentru un timp de acces mai bun. În practică nici un compilator modern nu mai ţine cont de acest cuvânt cheie, folosind automat registre atunci când codul poate fi optimizat în acest fel (de exemplu când observă că nu se accesează niciodata adresa variabilei în program).

SEP Memoria neocupată de datele statice şi de instrucţiunile unui program este împărţită între stivă şi heap. Consumul de memorie pe stivă este mai mare în programele cu funcţii recursive şi număr mare de apeluri recursive, iar consumul de memorie heap este mare în programele cu vectori şi matrice alocate (şi realocate) dinamic.

Exercitii laborator CB/CD

Codul sursa se gaseste aici [http://swarm.cs.pub.ro/~gmuraru/PC/ex.c]

Primul exercitiu presupune modificarea/adaugarea de instructiuni unui cod existent pentru a realiza anumite lucruri. In momentul actual programul citeste o matrice si afiseaza suma elementelor de pe fiecare linie.

- Nu uitati ca trebuie sa utilizam un coding style adecvat atunci cand scriem sursele.

Cerinte:

- Sa se mute elementele de pe o anumita linie, intr-un vector, alocat dinamic:
- Sa se mareasca dimensiunea matricei astfel incat sa aiba o linie in plus, iar pointerul specific ultimei linii sa indice spre vectorul generat anterior.

Următoarele două probleme vă vor fi date de asistent în cadrul laboratorului.

Checker laborator 9 [https://drive.google.com/drive/folders/1qB6EZLGVubKbuTXMtMue06egH_8fo25M]

Exerciţii de Laborator

1. **[2p]** Să se scrie un program care citeşte de la tastatură un număr pozitiv n împreună cu alt număr pozitiv max. Programul va aloca apoi dinamic un vector de întregi de n elemente, pe care îl va iniţializa cu numere aleatoare în intervalul [0..max-1]. Sortaţi vectorul, folosind metoda preferată, afişându-i conţinutul atât înainte, cât şi după ce sortarea a avut loc.

2. **[3p]** Să se scrie un program care citeşte de la tastatură două matrice: una inferior triunghiulară (toate elementele de deasupra diagonalei principale sunt nule), şi cealaltă superior triunghiulară. Ele vor fi reprezentate în memorie cât mai compact cu putinţă (fară a stoca şi zerourile de deasupra, respectiv dedesubtul diagonalei). Se va calcula apoi produsul celor matrice, şi se va afişa.
3. Un număr lung (cu o valoare mult mai mare decât maximul reprezentabil pe un tip de date întreg standard din C), poate fi reprezentat ca un vector char *v de cifre (considerate valori de tip char), în felul următor:
    - v[0] reprezintă numărul de cifre ale numărului lung. Lungimea vectorului în memorie va fi v[0]+1.
    - v[i], unde i este de la 1 la v[0], reprezintă a i-a cifră a numărului, în ordinea crescătoare a semnificativităţii. Astfel v[1] reprezintă cifra unităţilor, v[2] cifra zecilor, etc. O reprezentare eficientă va avea întotdeauna ultima cifră v[v[0]] nenulă (altfel numărul de cifre ar fi putut fi mai mic şi reprezentarea mai compactă).
    - **[1p]** a) Scrieţi o funcţie care construieşte vectorul de cifre asociat unui număr întreg simplu (de tipul int):

      ```
      char *build_number(int value);
      ```

    - **[2p]** b) Scrieţi o funcţie care adună două numere lungi şi întoarce ca rezultat un alt număr lung:

      ```
      char *add_numbers(char *a, char *b);
      ```

    - **[2p]** c) Scrieţi un program care calculează şirul Fibonacci folosind numere lungi. Se cer primii 100 de termeni ai şirului, afişaţi pe câte o linie în parte.

Toate funcţiile cerute vor aloca dinamic memoria necesară reprezentării vectorului întors. Numerele nefolosite vor trebui eliberate, pentru a evita consumarea memoriei. Trataţi tipul de date char ca pe un tip numeric (deci lucraţi cu vectori de numere, nu cu şiruri de caractere ASCII).

Bonus

1. Considerând structura unui număr lung prezentată la punctul precedent, să se rezolve următoarele:
    - **[1p]** a) Scrieţi o funcţie care înmulţeşte două numere lungi şi întoarce ca rezultat un alt număr lung:

      ```
      char *multiply_numbers(char *a, char *b);
      ```

    - **[1p]** b) Scrieţi un program care calculează factorialul numerelor de la 1 la 50, afişând câte un număr pe fiecare linie.
2. **[2p]** Se consideră un paralelipiped tridimensional cu dimensiunile citite de la tastatură, pentru care va trebui să alocaţi memorie. De asemenea, se citeşte apoi un număr pozitiv N, ce reprezintă un număr de bombe care vor fi plasate în paralelipiped. Apoi se citesc N triplete ce reprezintă coordonatele bombelor. Valorile citite vor trebui validate astfel încât să nu depăşească dimensiunile paralelipipedului. Pentru fiecare cub liber se va calcula numărul de bombe din cei maxim 26 de vecini ai săi, şi aceste numere vor fi afişate pe ecran, alături de coordonatele corespunzătoare. La sfârşitul execuţiei programului, memoria alocată va trebui eliberată.