Programarea calculatoarelor

alocare dinamică; funcţiile malloc, free, realloc; vectori de pointeri; alocarea matricilor

În special când lucrăm cu volume mari de date, ale căror dimensiuni individuale variază într-o plajă foarte largă de valori, setarea unor structuri de date astfel încât dimensiunea lor să acopere cazul maximal, nu mai este o opţiune viabilă de implementare. De exemplu, dorim să avem o bază de date cu literatură (romane, articole, poezii, etc), inclusiv conţinutul lor. Aplicând tehnicile învăţate până acum (ex: o matrice cu câte o lucrare pe linie), ar trebui să declarăm pentru fiecare lucrare dimensiunea maximală, de exemplu 500,000 de caractere pentru un roman în două volume. Ţinând cont că o poezie are în medie doar câteva sute de caractere, rezultă că o poezie ocupă practic doar o miime din memoria care a fost alocată unei lucrări, ceea ce este foarte ineficient. În această situaţie baza de date ar trebui să fie destul de mică, fiindcă altfel nu ar încăpea în memorie.

Pentru asemenea situaţii, limbajul C ne pune la dispoziţie un mecanism, numit *alocare dinamică de memorie*, prin intermediul căruia putem să cerem doar atâta memorie de cât avem nevoie, reuşind să maximizăm ocuparea acesteia. Există în C trei tipuri de memorie:

- **memoria statică** - este compusă din variabilele globale şi toate constantele. Deoarece dimensiunea acestei zone se poate calcula încă de la compilarea programului, compilatorul alocă pentru ea o zonă fixă de memorie, de dimensiune constantă, care va fi disponibilă de la începutul până la sfârşitul programului.
- **stiva apelurilor de funcţii** - în momentul în care o funcţie este apelată, se alocă pentru ea în memorie o zonă de dimensiune fixă, pentru argumentele şi variabilele sale locale. La ieşirea din funcţie, această zonă de memorie este eliberată automat.
- **memoria dinamică** - această memorie se compune din blocuri de dimensiuni variabile, care sunt cerute de la sistemul de operare (Linux, Windows) de către programator, prin intermediul unor funcţii specifice. Astfel, programatorul poate cere în orice moment cantitatea exactă de memorie de care are nevoie.

Funcţiile pentru alocarea dinamică a memorie sunt declarate în antetul *stdlib.h* (*standard library*). Dintre ele, trei sunt mai importante:

- **malloc** - alocă un bloc de memorie
- **calloc** - alocă un bloc de memorie, exact ca *malloc* şi îl iniţializează cu 0 (pune toţi octeţii din acel bloc de memorie pe 0)
- **free** - eliberează un bloc de memorie, atunci când nu mai este nevoie de el
- **realloc** - redimensionează un bloc de memorie deja alocat

Funcţiile *malloc* şi *free*

Pentru a aloca un bloc de memorie, se foloseşte funcţia *malloc* (*memory alloc*), cu următoarea declaraţie:

```
void *malloc(size_t nrOcteti)
```

malloc primeşte ca **argument dimensiunea în octeţi** a blocului de memorie de alocat. Pentru a se afla această dimensiune, se poate folosi operatorul *sizeof*, care primeşte ca argument un tip de date sau o expresie şi returnează dimensiunea sa în octeţi.

Funcţia calloc (clear alloc) are următoarea declaraţie:

```
void *calloc(size_t nrElemente, size_t nrOcteti)
```

Este de preferat utilizarea lui malloc deoarece este mai rapid, calloc utilizând timp preţios pentru a face “clear” la blocul de memorie pe care îl alocă.

Dacă avem n elemente şi un vector cu elemente întregi v pe care dorim să îl alocăm dinamic, putem utiliza:

```
int v[10],n;
printf("n=");
scanf("%d",&n);
v=(int*)malloc(n*sizeof(int) ); // varianta cu malloc
v=(int*)calloc(n, sizeof(int) ); // varianta cu calloc, daca dorim vectorul sa fie si initializat cu 0
```

size_t este definit ca fiind un tip întreg fără semn (*unsigned*), cu proprietatea că prin intermediul unei variabile index de tip *size_t* se poate accesa orice element al celui mai mare vector posibil pentru un program C. Ţinând cont că vectorul cu cele mai multe elemente posibile este cel de tip *char[]* (fiindcă fiecare celulă va fi doar un octet de memorie), *size_t* de obicei este tipul întreg fără semn prin intermediul căruia se pot număra toate locaţiile de memorie. De exemplu, pentru un sistem de operare pe 32 de biţi, *size_t* are 32 de biţi, iar pentru un sistem de operare pe 64 de biţi, *size_t* are 64 de biţi.

Dacă *malloc* reuşeşte să aloce memoria cerută, se va returna un pointer către începutul blocului de memorie alocat. În C, **void*** înseamnă pointer generic, către orice tip de date. *malloc* returnează acest tip de date (*void**), deoarece ea, fiind o funcţie, nu-şi poate adapta tipul pointerului returnat în funcţie de ce tip de date s-a cerut. Pointerii generici trebuie convertiţi înainte de a fi folosiţi la tipuri specifice de pointeri, de exemplu *char**.

Dacă *malloc* nu reuşeşte să aloce memoria cerută (de exemplu dacă nu este suficientă memorie disponibilă), ea va returna **NULL**. Reamintim că NULL este un pointer care pointează la adresa 0 de memorie şi, prin convenţie, se consideră că nu pointează la nimic.

Când nu mai este nevoie de un anumit bloc de memorie, programatorul trebuie să-l elibereze, folosind funcţia *free*. Aceasta are următoarea declaraţie:

void free(void *ptr)

free primeşte ca parametru un pointer la începutul unui bloc de memorie alocat cu *malloc* şi eliberează acest bloc. Se constată şi aici folosirea pointerilor generici (*void**), pentru a permite ca argumentul să fie orice tip de pointer. Dacă *ptr==NULL*, *free* nu are niciun efect.

La terminarea unui program, sistemul de operare eliberează automat toată memoria şi alte resurse ocupate de acel program, deci în principiu nu mai trebuie să eliberăm memoria manual, dacă ştim că următoarea operaţie este ieşirea din program. Chiar şi în această situaţie, este bine totuşi să eliberăm noi memoria, pentru a ne crea obiceiul de a ţine cont ce se petrece cu fiecare bloc de memorie alocat dinamic. Un alt avantaj al eliberării manuale în această situaţie, este faptul că, dacă pe viitor mai adăugăm ceva la program şi acesta nu se va mai termina în punctul respectiv, noi suntem asiguraţi că nu avem memorie neeliberată.

Exemplu: Se cere un număr n, iar apoi n numere întregi. Se cere să se sorteze crescător numerele. Programul va utiliza doar strictul necesar de memorie.

```
#include <stdio.h>
#include <stdlib.h>                 // pentru functiile malloc, free, exit

int main()
{
   int i,j,n;
   int *v;                          // vector alocat dinamic, cu elemente de tip int
   int e;
   printf("n=");scanf("%d",&n);
```

```
   // aloca dinamic un bloc de memorie pentru n elemente de tip int
   if((v=(int*)malloc(n*sizeof(int)))==NULL){
      // daca nu s-a reusit alocarea, afiseaza un mesaj
      // si iese din program returnand sistemului de operare un cod de eroare
      printf("memorie insuficienta\n");
      exit(EXIT_FAILURE);
      }

   for(i=0;i<n;i++){
      printf("v[%d]=",i);scanf("%d",&v[i]);
      }

   do{                              // bubble sort
      e=0;                          // daca au avut loc interschimbari
      for(i=1;i<n;i++){
         if(v[i-1]>v[i]){
            int tmp=v[i-1];
            v[i-1]=v[i];
            v[i]=tmp;
            e=1;
            }
         }
      }while(e);

   for(i=0;i<n;i++)printf("%d\n",v[i]);

   free(v);                         // elibereaza memoria alocata pentru v

   return 0;
}
```

În exemplul de mai sus, dacă utilizatorul doreşte să introducă *n* numere, va fi nevoie de un bloc de memorie cu suficient spaţiu pentru aceste numere. Ţinând cont că dimensiunea în octeţi a unui *int* este *sizeof(int)*, atunci pentru *n* numere întregi vom avea nevoie de *n*sizeof(int)* octeţi. Toate aceste elemente vor fi accesate prin pointerul *v*, care va pointa la începutul blocului de memorie. Deoarece un pointer este în acelaşi timp un vector care începe la adresa pointată de pointer, putem accesa prin intermediul lui *v* orice celulă de memorie din blocul alocat, ca şi când acesta ar fi fost un vector. Singura cerinţă este ca *v* să pointeze la tipul dorit pentru celule, adică *int*.

Valoarea returnată de *malloc*, care este de tipul *pointer generic* (*void**), se converteşte la tipul de pointer specific dorit (*int**), folosind operatorul cast: ...*(int*)malloc*…şi astfel putem să atribuim această valoare lui *v*. După ce s-a atribuit lui *v* valoarea returnată de *malloc*, trebuie verificat dacă *malloc* a reuşit să aloce memoria cerută. De obicei este mai simplu să testăm cazul în care *malloc* nu a reuşit alocarea de memorie, fiindcă atunci nu mai avem ce face şi programul va trebui terminat. În exemplu, se poate constata că dacă *malloc* a returnat NULL, se va afişa un mesaj de eroare şi apoi, prin intermediul funcţiei *exit(cod)*, se iese din program. Funcţia *exit* este declarată tot în *stdlib.h*. Ea termină în mod necondiţionat programul curent, de oriunde din program ar fi apelată. Funcţia exit returnează sistemului de operare una dintre cele 2 valori predefinite: EXIT_SUCCESS, dacă programul s-a încheiat cu succes sau EXIT_FAILURE, dacă a avut loc o eroare la execuţia programului.

Datorită faptului că un pointer poate fi oricând considerat un vector, folosirea zonei de memorie alocată dinamic este identică cu situaţia în care *v* ar fi fost declarat ca vector de întregi. Astfel, citirea datelor, sortarea şi afişarea

lor sunt identice cu implementările lor de la laboratorul despre vectori şi este invizibil faptul că de fapt *v* este un pointer.

După ce s-au terminat operaţiile cu blocul de memorie alocat dinamic, se foloseşte funcţia *free* pentru a elibera această zonă de memorie. Se poate constata că *free* a primit ca argument chiar începutul blocului de memorie alocat cu *malloc*, început care este pointat de *v*.

Aplicaţia 11.1: Să se scrie o funcţie *citire(n)*, care primeşte ca argument un număr *n* şi alocă dinamic un vector de *n* numere întregi, pe care îl iniţializează cu valori citite de la tastatură şi îl returnează. În programul principal se citesc două numere, *m* şi *n*, iar apoi, folosind funcţia *citire*, se citesc elementele a doi vectori, primul de *m* elemente iar al doilea de *n* elemente. Să se afişeze toate elementele din primul vector care se regăsesc şi în al doilea vector. Programul va utiliza doar strictul necesar de memorie.

Funcţia *realloc*

Funcţia *realloc* redimensionează un bloc de memorie alocat dinamic, putându-i mări sau reduce dimensiunea. realloc are următoarea declaraţie:

```
void *realloc(void *ptr, size_t nOcteti)
```

realloc are două argumente:

- *ptr* - un pointer la o bloc alocat dinamic
- *nOcteți* - noua dimensiune pe care va trebui să o aibă blocul de memorie

Dacă *ptr==NULL*, *realloc* se comportă *malloc*, alocând un bloc de memorie de dimensiunea cerută.

realloc nu este obligat să realoce memoria începând tot cu aceeaşi adresă, deci adresa blocului realocat poate fi diferită de cea a blocului originar (*ptr*). Dacă modifică adresa blocului, *realloc* va copia automat conţinutul vechiului bloc la noua adresă.

realloc returnează adresa de memorie a blocului alocat/redimensionat sau NULL, dacă operaţia nu a reuşit. În acest caz, pointerul *ptr* rămâne nemodificat.

Exemplu: Se citesc de la tastatură numere până la introducerea valorii 0. Să se afişeze dacă toate numerele sunt pare, sau nu. Programul va utiliza doar strictul necesar de memorie.

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

int main()
{
   int n=0;                          // cate numere sunt deja in vector
   // initializam v==NULL, deoarece aceasta va fi valoare pasata lui realloc
   // si daca v==NULL, realloc se va comporta ca malloc
   int *v=NULL;
   int *v2;                          // variabila auxiliara pentru realloc
   for(;;){                          // bucla infinita pentru a citi toate numerele
      int k;
      printf("v[%d]=",n);scanf("%d",&k);
      if(!k)break;                   // daca s-a introdus 0, se iese din bucla
      // redimensioneaza v, pentru a avea loc noul element, k
      n++;
      if((v2=(int*)realloc(v,n*sizeof(int)))==NULL){
         printf("memorie insuficienta\n");
         free(v);
```

```
        exit(EXIT_FAILURE);
        }
    v=v2;
    v[n-1]=k;                       // insereaza noul element la ultima pozitie in vectorul redimensionat
    }

  int i;
  int pare=1;                       // 1 daca toate numerele sunt pare, altfel 0
  for(i=0;i<n;i++){
    if(v[i]%2!=0){                  // daca s-a gasit un numar impar, se reseteaza "pare" si se iese din bucla
        pare=0;
        break;
        }
    }
  if(pare)printf("toate numerele sunt pare\n");
    else printf("exista si numere impare\n");

  free(v);                          // elibereaza memoria alocata pentru v

  return 0;
}
```

Numerele vor fi accesate prin intermediul pointerului *v*. În acest caz, iniţializarea ”*v=NULL*” este importantă, deoarece primul apel al lui *realloc* se va transforma astfel într-o alocare iniţială, ca şi când ar fi fost *malloc*. Numărul de numere începe de la ”*n=0*”, iar înainte de fiecare realocare se incrementează, pentru a face loc noului element.

Deoarece *realloc* poate returna o altă adresă de memorie decât cea iniţială, valoarea returnată se memorează într-o variabilă temporară *v2*. Ulterior, doar dacă alocarea a reuşit, setăm *v* cu noua valoare din *v2*. Altfel, dacă am fi memorat valoarea returnată de *realloc* direct în *v* şi realocarea nu ar fi reuşit, *v* ar fi devenit NULL şi vechea valoare (care nu s-ar fi modificat), ar fi devenit inaccesibilă (fiindcă nu mai avem niciun pointer care să pointeze la ea), deci nu am mai fi putut elibera acea memorie.

În caz că *realloc* returnează NULL, înainte de a ieşi din program eliberăm blocul de memorie alocat anterior. Reamintim că *free(NULL)* nu are niciun efect.

Vectori de pointeri

Dacă dorim să avem un vector în care să memorăm mai multe şiruri de caractere, ţinând cont că un şir de caractere are tipul *char**, vectorul va fi declarat de forma:

```
char *siruri[100];
```

Această declaraţie se citeşte: *siruri este un vector de 100 de elemente, fiecare element având tipul pointer la char*. În acest vector, la fiecare index se va afla un pointer către blocul de memorie în care se află un şir de caractere.

Am putea iniţializa o asemenea structură de date în felul următor:

```
char *siruri[3]={“Ion”, “Ana”, “Maria”};
```

Compilatorul îşi va alege el unde să stocheze în memorie cele 3 şiruri de caractere (inclusiv terminatorul de şir care se pune automat). Cele 3 şiruri este posibil să nu fie dispuse consecutiv în memorie. Dacă folosim un index în vectorul *şiruri*, vom obține pointerul de la acel index:

printf(“%s”,siruri[2]); // Maria (pointer[2] are tipul char*)

Dacă dorim ca *siruri* să fie alocat dinamic, astfel încât să ocupe doar strictul necesar de memorie, ne vom folosi de echivalența vector <-> pointer și îl vom declara în felul următor:

char **siruri:

În acest caz, *siruri* este un pointer la celule de memorie care ele însele sunt pointeri la *char*. Diferența esențială față de declararea cu vector, este faptul că acum, având un pointer, putem modifica adresa de memorie la care se află *siruri* în memorie, deci putem folosi alocarea dinamică.

Exemplu: Se citesc linii de caractere, de maxim 200 de caractere fiecare, până la întâlnirea liniei vide. Ulterior se cere un șir, terminat și el cu \n. Să se afișeze toate liniile care conțin șirul dat. Programul va utiliza doar strictul necesar de memorie.

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

char **linii=NULL;              // vector de linii
int nrLinii=0;                  // nr de linii din linii

void eliberare()
{
   int i;
   for(i=0;i<nrLinii;i++)free(linii[i]);          // elibereaza fiecare linie din linii
   free(linii);                                   // elibereaza vectorul in sine
}

// citeste o linie de max 200 caractere si depune caracterele citite intr-un bloc alocat dinamic
// returneaza adresa blocului
char *linie()
{
   char buf[201];
   char *p;
   size_t n;
   fgets(buf,201,stdin);
   buf[strcspn(buf,"\n")]='\0';                   // sterge posibilul \n final
   n=strlen(buf)+1;                               // nr de caractere total, inclusiv terminatorul
   if((p=(char*)malloc(n*sizeof(char)))==NULL){
      eliberare();
      printf("memorie insuficienta\n");
      exit(EXIT_FAILURE);
      }
   strcpy(p,buf);
   return p;
}

int main()
{
   char **linii2;
   char *p;
   int i;
   for(;;){                                       // citeste pana la linia vida
      p=linie();
```

```
        if(strlen(p)==0){
            free(p);                        // elibereaza separat linia vida, deoarece ea nu se depune in linii
            break;
            }
        //realoca linii pentru o noua linie
        nrLinii++;
        if((linii2=(char**)realloc(linii,nrLinii*sizeof(char*)))==NULL){
            eliberare();
            printf("memorie insuficienta\n");
            exit(EXIT_FAILURE);
            }
        linii=linii2;
        linii[nrLinii-1]=p;
        }
    printf("introduceti sirul de cautat:");
    p=linie();
    // afiseaza toate liniile care contin sirul dat
    for(i=0;i<nrLinii;i++){
        if(strstr(linii[i],p))printf("%s\n",linii[i]);
        }
    free(p);
    eliberare();
    return 0;
}
```

Variabilele *linii* şi *nrLinii* au fost declarate globale, pentru a fi vizibile atât din *main* cât şi din *eliberare*. Strict vorbind, iniţializările lor nu sunt necesare, deoarece fiind variabile statice, sunt automat iniţializate pe 0, dar aceste iniţializări adaugă un plus de lizibilitate programului. Funcţia *eliberare* eliberează memoria alocată dinamic: fiecare linie din vectorul de linii şi vectorul în sine. Ea se apelează atât la sfârşitul programului, cât şi de fiecare dată când ieşim din program cu *exit*, pentru a se elibera memoria alocată până atunci.

Funcţia *linie* citeşte maxim 200 de caractere şi pe baza lor alocă un bloc de dimensiunea necesară, pe care-l returnează. Astfel nu există memorie suplimentară, deoarece la ieşirea din funcţie, toate variabilele ei locale, incluzând *buf*, sunt şterse din memorie.

Deoarece *linii* este un pointer la elemente de tip *char**, înseamnă că fiecare element are dimensiunea acestui tip de date. Astfel, pentru *nrLinii* pointeri, avem nevoie de *nrLinii*sizeof(char*)* octeţi.

Căutarea unui subşir într-un şir se face cu funcţia *strstr(sir,subsir)*. Dacă subşirul a fost găsit, *strstr* returnează adresa lui de început în *sir*, altfel returnează NULL.

Singura limitare din exemplul anterior este faptul că lungimea unei linii a fost limitată artificial la maxim 200 de caractere. Pentru a elimina şi această limitare, ar trebui ca funcţia linie() să citească câte un caracter, iar buf să fie alocat şi el dinamic, astfel încât să crească pe măsură ce se citesc caractere. Exemplul de mai jos prezintă această modificare:

```
char *linie(){
    char *buf=NULL;
    char *buf2;
    char ch;
    size_t n=0;
    for(;;){
        n++;
        if((buf2=(char*)realloc(buf,n))==NULL){
            free(buf);
```

```
            eliberare();
            printf("memorie insuficienta\n");
            exit(EXIT_FAILURE);
            }
        buf=buf2;
        ch=getchar();
        if(ch=='\n')break;
        buf[n-1]=ch;
        }
    buf[n-1]='\0';
    return buf;
}
```

În această variantă, *buf* este alocat dinamic şi dimensiunea sa creşte pe măsură ce citim noi caractere. Citirea se face caracter cu caracter, folosind *getchar*, până când se întâlneşte \n. În caz că apare o eroare, pe lângă memoria eliberată de *eliberare*, trebuie eliberat şi *buf*.

Realocarea se face înainte de citirea unui caracter, deoarece astfel ne asigurăm că avem mereu o celulă liberă la sfârşit. Dacă se citeşte \n, în această celulă se va depune terminatorul, altfel se depune caracterul citit.

Aplicaţia 11.2: Se citesc linii de caractere, până la întâlnirea liniei vide. Să se sorteze aceste linii în ordine alfabetică şi să se afişeze. Programul va utiliza doar strictul necesar de memorie.

Alocarea dinamică a matricilor

Matricile se pot aloca dinamic în două feluri, fiecare cu avantajele şi dezavantajele sale.

În prima variantă, alocăm matricea ca pe un vector care conţine toate elementele, la fel cum face şi compilatorul. Elementele vor fi stocate în memorie linie după linie.

Exemplu: Se citesc m şi n. Folosind doar cantitatea necesară de memorie, se citesc elementele unei matrici a[m][n]. Se cere să se afişeze maximul fiecărei coloane.

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

int main()
{
    int m,n,i,j;
    int *a;                                 // matricea este implementata ca un vector ce contine toate elementele
    printf("m=");scanf("%d",&m);
    printf("n=");scanf("%d",&n);
    // alocare matrice
    if((a=(int*)malloc(m*n*sizeof(int)))==NULL){
        printf("memorie insuficienta\n");
        exit(EXIT_FAILURE);
        }
    for(i=0;i<m;i++){
        for(j=0;j<n;j++){
            printf("a[%d][%d]=",i,j);scanf("%d",&a[i*n+j]);
            }
        }
    // cauta maximul fiecarei coloane
    // parcurgea se face in ordinea coloanelor
    for(j=0;j<n;j++){
```

```
        int max=a[j];                           // primul element de pe coloana j, din linia 0, adica: 0*n+j
        // parcurge fiecare element din coloana j, incepand cu a doua linie
        for(i=1;i<m;i++){
            int k=a[i*n+j];
            if(k>max)max=k;
            }
        printf("maximul coloanei %d este: %d\n",j,max);
        }
    // elibereaza memoria alocata pentru matrice
    free(a);
    return 0;
}
```

Deoarece matricea este implementată ca un vector, alocarea şi eliberarea ei se fac simplu, folosind doar câte o instrucţiune *malloc* sau *free*. În schimb, la fiecare acces al unui element al matricii, programatorul trebuie să folosească formula de acces a unui element, adică *a[i*nr_coloane+j]*, echivalent cu **(a+i*nr_coloane+j)*.

A doua metodă de alocare a unei matrici este să folosim un vector de pointeri, fiecare pointer pointând la câte o linie din matrice. În acest caz liniile vor fi alocate separat, ca blocuri de memorie independente. Matricea va fi definită ca *int **a*, adică *a* este un pointer/vector la pointeri/vectori cu elemente de tip *int*. În acest caz, *a[i]* înseamnă pointerul (*int**) de la indexul *i* din *a*, adică adresa memoriei care conţine acea linie. *a[i][j]* înseamnă elementul *j* pointat de pointerul stocat la indexul *i* în *a*.

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

int main()
{
    int m,n,i,j;
    int **a;                                // matricea este implementata ca un vector de pointeri la liniile ei
    printf("m=");scanf("%d",&m);
    printf("n=");scanf("%d",&n);
    // alocare vector de pointeri la linii
    if((a=(int**)malloc(m*sizeof(int*)))==NULL){
        printf("memorie insuficienta\n");
        exit(EXIT_FAILURE);
        }
    // alocare linii
    for(i=0;i<m;i++){
        if((a[i]=(int*)malloc(n*sizeof(int)))==NULL){
            for(i--;i>=0;i--)free(a[i]);                    // elibereaza liniile alocate anterior
            free(a);                                        // elibereaza vectorul de pointeri
            printf("memorie insuficienta\n");
            exit(EXIT_FAILURE);
            }
        }
    for(i=0;i<m;i++){
        for(j=0;j<n;j++){
            printf("a[%d][%d]=",i,j);scanf("%d",&a[i][j]);
            }
        }
    // cauta maximul fiecarei coloane
    // parcurgea se face in ordinea coloanelor
```

```
    for(j=0;j<n;j++){
        int max=a[0][j];
        // parcurge fiecare element din coloana j, incepand cu a doua linie
        for(i=1;i<m;i++){
            int k=a[i][j];
            if(k>max)max=k;
            }
        printf("maximul coloanei %d este: %d\n",j,max);
        }

    for(i=0;i<m;i++)free(a[i]);              // elibereaza fiecare linie
    free(a);                                 // elibereaza vectorul de linii
    return 0;
}
```

Se constată că în această variantă, alocarea şi eliberarea memoriei sunt mai complexe, pentru că fiecare linie se alocă separat. În schimb, accesarea elementelor matricii se face exact ca în cazul în care ea ar fi fost declarată ca atare (int a[M][N]), ceea ce simplifică toate operaţiile cu ea.

Erorile de alocare/eliberare a memoriei dinamice pot fi împărţite în două categorii:

- Memoria alocată nu este eliberată folosind *free.* Apar astfel *scurgeri de memorie* (*memory leaks*). În acest caz, se tot adună blocuri de memorie alocată, care nu sunt folosite în niciun fel, până când se va epuiza toată memoria calculatorului. Aceste erori sunt în special periculoase pentru programele care trebuie să ruleze încontinuu, un timp îndelungat (ex: servere), deoarece în acest caz, chiar dacă la o alocare fără eliberare se pierd doar câţiva octeţi, în timp aceste zone se tot adună.
  Exemplu: într-o buclă alocăm la fiecare iteraţie un bloc de memorie în aceeaşi variabilă, dar nu eliberăm decât la finalul buclei ultimul bloc alocat. În acest caz toate blocurile alocate intermediar vor rămâne alocate şi vor constitui scurgeri de memorie.
- Folosirea unor pointeri la blocuri de memorie care deja au fost eliberate (*dangling pointers*) - În acest caz, deşi am eliberat un bloc de memorie, continuăm să îl folosim prin intermediul unui pointer care continuă să pointeze la vechea adresă alocată. Tot de acest gen sunt erorile în care încercăm să eliberăm de mai multe ori acelaşi bloc de memorie. Din aceste situaţii pot rezulta erori de tipul *segmentation fault* (când memoria eliberată nu mai este accesibilă din program) sau de corupere a altor date din program (când în acea memorie am alocat un nou bloc pe care-l folosim la altceva şi deci vom suprascrie noile date prin intermediul vechiului pointer). Pentru a se evita aceste situaţii, este bine ca pointerii eliberaţi să fie setaţi pe NULL, pentru a marca astfel faptul că nu sunt folosiţi.

Aplicaţii propuse

Aplicaţia 11.3: Să se scrie o funcţie care primeşte două şiruri de caractere şi le concatenează cu un spaţiu între ele, folosind un bloc de memorie alocat dinamic pentru rezultat. Funcţia va returna acest bloc de memorie. Să se verifice funcţia cu două şiruri de caractere introduse de la tastatură, fiecare de maxim 100 de caractere.

Aplicaţia 11.4: Se citesc numere până la întâlnirea numărului 0. Să se afişeze aceste numere în ordine inversă. Programul va folosi doar minimul necesar de memorie.

Aplicaţia 11.5: Jocul fazan: se citesc de la tastatură cuvinte separate prin spații albe (spațiu, TAB sau ENTER), până la întâlnirea cuvântului vid. Pornind de la cuvintele citite, se construieşte un şir de caractere după următoarele reguli:
- Primul cuvânt citit se adaugă la şirul de caractere
- Fiecare din următoarele cuvinte citite se adaugă la şirul de caractere dacă ultimele două litere ale şirului coincid cu primele două litere ale cuvântului (nu se face distincție între literele mici şi cele mari);

Cuvintele adăugate la şirul de caractere sunt despărțite de caracterul '-'. Să se afişeze şirul astfel construit.

Programul va folosi alocare dinamică astfel încât spațiul de memorie utilizat să fie minim.
Spre exemplu, pentru intrarea: Fazan Antic Noe icoana Egipt naftalina nimic Narcisa trei altceva santier iarba Caine Pisica erudit
se afişează: Fazan-Antic-icoana-naftalina-Narcisa-santier-erudit