Considerații teoretice

Formatori:
Tutor: Stângaciu Valentin
Tutor: Belu Claudiu-Marcel
+3

Data de începere a cursului:
25.09.2023
Utilizatori înscriși
Calendar
Note

Considerații teoretice

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 opțiune viabilă de implementare. De exemplu, dorim să avem o bază de date cu literatură (romane, articole, poezii, etc), inclusiv conținutul lor. Aplicând tehnicile învățate 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. Ținând 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 situație baza de date ar trebui să fie destul de mică, fiindcă altfel nu ar încăpea în memorie.
Pentru asemenea situații, limbajul C ne pune la dispoziție un mecanism, numit *alocare dinamică de memorie*, prin intermediul căruia putem să cerem doar atâta memorie de cât avem nevoie, reușind să maximizăm ocuparea acesteia. Există în C trei tipuri de memorie:

memoria statică - este compusă din variabilele globale și 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 sfârșitul programului.

stiva apelurilor de funcții - în momentul în care o funcție este apelată, se alocă pentru ea în memorie o zonă de dimensiune fixă, pentru argumentele și variabilele sale locale. La ieșirea din funcție, 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 funcții specifice. Astfel, programatorul poate cere în orice moment cantitatea exactă de memorie de care are nevoie.

Funcțiile 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* și îl inițializează cu 0 (pune toți octeții 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

Funcțiile *malloc* și *free*

Pentru a aloca un bloc de memorie, se folosește funcția *malloc* (*memory alloc*), cu următoarea declarație:

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

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

Funcția *calloc* (*clear alloc*) are următoarea declarație:

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

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

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

```
int *v,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. Ținând 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 locațiile de memorie. De exemplu, pentru un sistem de operare pe 32 de biți, *size_t* are 32 de biți, iar pentru un sistem de operare pe 64 de biți, *size_t* are 64 de biți.

Dacă *malloc* reușește 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 funcție, nu-și poate adapta tipul pointerului returnat în funcție de ce tip de date s-a cerut. Pointerii generici trebuie convertiți înainte de a fi folosiți la tipuri specifice de pointeri, de exemplu *char**.

Dacă *malloc* nu reușește 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 și, prin convenție, se consideră că nu pointează la nimic.

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

```
void free(void *ptr)
```

free primește ca parametru un pointer la începutul unui bloc de memorie alocat cu *malloc* și eliberează acest bloc. Se constată și 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 și alte resurse ocupate de acel program, deci în principiu nu mai trebuie să eliberăm memoria manual, dacă știm că următoarea operație este ieșirea din program. Chiar și în această situație, este bine totuși să eliberăm noi memoria, pentru a ne crea obiceiul de a ține cont ce se petrece cu fiecare bloc de memorie alocat dinamic. Un alt avantaj al eliberării manuale în această situație, este faptul că, dacă pe viitor mai adăugăm ceva la program și acesta nu se va mai termina în punctul respectiv, noi suntem asigurați 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.

```c
#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 dorește să introducă *n* numere, va fi nevoie de un bloc de memorie cu suficient spațiu pentru aceste numere. Ținând cont că dimensiunea în octeți a unui *int* este *sizeof(int)*, atunci pentru *n* numere întregi vom avea nevoie de *n*sizeof(int)* octeți. Toate aceste elemente vor fi accesate prin pointerul *v*, care va pointa la începutul blocului de memorie. Deoarece un pointer este în același timp un vector care începe la adresa pointată de pointer, putem accesa prin intermediul lui *v* orice celulă de memorie din blocul alocat, ca și când acesta ar fi fost un vector. Singura cerință este ca *v* să pointeze la tipul dorit pentru celule, adică *int*.

Valoarea returnată de *malloc*, care este de tipul *pointer generic* (*void**), se convertește la tipul de pointer specific dorit (*int**), folosind operatorul cast: *...(int*)malloc...*și astfel putem să atribuim această valoare lui *v*. După ce s-a atribuit lui *v* valoarea returnată de *malloc*, trebuie verificat dacă *malloc* a reușit să aloce memoria cerută. De obicei este mai simplu să testăm cazul în care *malloc* nu a reușit alocarea de memorie, fiindcă atunci nu mai avem ce face și programul va trebui terminat. În exemplu, se poate constata că dacă *malloc* a returnat NULL, se va afișa un mesaj de eroare și apoi, prin intermediul funcției *exit(cod)*, se iese din program. Funcția *exit* este declarată tot în *stdlib.h*. Ea termină în mod necondiționat programul curent, de oriunde din program ar fi apelată. Funcția 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 execuția programului.

Datorită faptului că un pointer poate fi oricând considerat un vector, folosirea zonei de memorie alocată dinamic este identică cu situația în care *v* ar fi fost declarat ca vector de întregi. Astfel, citirea datelor, sortarea și afișarea lor sunt identice cu implementările lor de la laboratorul despre vectori și este invizibil faptul că de fapt *v* este un pointer.

După ce s-au terminat operațiile cu blocul de memorie alocat dinamic, se folosește funcția *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*.

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

Funcția *realloc*

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

```
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 aceeași adresă, deci adresa blocului realocat poate fi diferită de cea a blocului originar (*ptr*). Dacă modifică adresa blocului, *realloc* va copia automat conținutul vechiului bloc la noua adresă.

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

Exemplu: Se citesc de la tastatură numere până la introducerea valorii 0. Să se afișeze 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, inițializarea "*v=NULL*" este importantă, deoarece primul apel al lui *realloc* se va transforma astfel într-o alocare inițială, ca și 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 inițială, valoarea returnată se memorează într-o variabilă temporară *v2*. Ulterior, doar dacă alocarea a reușit, setăm *v* cu noua valoare din *v2*. Altfel, dacă am fi memorat valoarea returnată de *realloc* direct în *v* și realocarea nu ar fi reușit, *v* ar fi devenit NULL și 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 ieși 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 șiruri de caractere, ținând cont că un șir de caractere are tipul *char**, vectorul va fi declarat de forma:

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

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

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

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

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

```
printf(“%s”,siruri[2]); // Maria (siruri[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.

```c
#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* și *nrLinii* au fost declarate globale, pentru a fi vizibile atât din *main* cât și din *eliberare*. Strict vorbind, inițializările lor nu sunt necesare, deoarece fiind variabile statice, sunt automat inițializate pe 0, dar aceste inițializări adaugă un plus de lizibilitate programului. Funcția *eliberare* eliberează memoria alocată dinamic: fiecare linie din vectorul de linii și vectorul în sine. Ea se apelează atât la sfârșitul programului, cât și de fiecare dată când ieșim din program cu *exit*, pentru a se elibera memoria alocată până atunci.

Funcția *linie* citește maxim 200 de caractere și pe baza lor alocă un bloc de dimensiunea necesară, pe care-l returnează. Astfel nu există memorie suplimentară, deoarece la ieșirea din funcție, toate variabilele ei locale, incluzând *buf*, sunt șterse 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*)* octeți.

Căutarea unui subșir într-un șir se face cu funcția *strstr(sir,subsir)*. Dacă subșirul 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 și această limitare, ar trebui ca funcția linie() să citească câte un caracter, iar buf să fie alocat și 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 și dimensiunea sa crește pe măsură ce citim noi caractere. Citirea se face caracter cu caracter, folosind *getchar*, până când se întâlnește \n. În caz că apare o eroare, pe lângă memoria eliberată de *eliberare*, trebuie eliberat și *buf*.

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

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

Erorile de alocare/eliberare a memoriei dinamice pot fi împărțite 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 câțiva octeți, în timp aceste zone se tot adună.
Exemplu: într-o buclă alocăm la fiecare iterație un bloc de memorie în aceeași variabilă, dar nu eliberăm decât la finalul buclei ultimul bloc alocat. În acest caz toate blocurile alocate intermediar vor rămâne alocate și vor constitui scurgeri de memorie.

Folosirea unor pointeri la blocuri de memorie care deja au fost eliberate (*dangling pointers*) - În acest caz, deși 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 același bloc de memorie. Din aceste situații 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 și deci vom suprascrie noile date prin intermediul vechiului pointer). Pentru a se evita aceste situații, este bine ca pointerii eliberați să fie setați pe NULL, pentru a marca astfel faptul că nu sunt folosiți.

Alocarea dinamică a structurilor

Putem aloca dinamic structuri, la fel ca pe orice alt tip de date, așa cum s-a discutat la alocarea dinamică. Această facilitate ne permite să avem baze de date care ocupă doar atâta memorie cât este necesară.

Exemplul 12.5: Un angajat este definit prin nume, funcție și salariu. Să se implementeze o bază de date cu angajați, care să permită adăugarea, ștergerea și listarea înregistrărilor, astfel încât memoria folosită să fie minimă.

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

typedef struct{
 char nume[30];
 char functie[30];
 float salariu;
}Angajat;

Angajat *angajati=NULL; // vector alocat dinamic de angajati
int nAngajati=0;

// afiseaza un text, iar apoi citeste un sir de caractere in destinatia specificata
void citireSir(const char *text,char *dst,size_t max)
{
 printf("%s: ",text);
 fgets(dst,max,stdin);
 dst[strcspn(dst, "\n")]='\0';
}

int main()
{
 int i,j,op;
 Angajat *v2;
 char buf[30];
 for(;;){
 printf("1. Adaugare\n");
 printf("2. Stergere\n");
 printf("3. Listare\n");
 printf("4. Iesire\n");
 printf("Optiune: ");
 scanf("%d",&op);
 switch(op){
 case 1:
 v2=(Angajat*)realloc(angajati,(nAngajati+1)*sizeof(Angajat));
 if(!v2){
 printf("memorie insuficienta\n");
 free(angajati);
 exit(EXIT_FAILURE);
 }
 angajati=v2;
 getchar();
 citireSir("Nume",angajati[nAngajati].nume,30);
 citireSir("Functie",angajati[nAngajati].functie,30);
 printf("Salariu: ");
 scanf("%g",&angajati[nAngajati].salariu);
 nAngajati++;
 break;
 case 2:
 getchar();
 citireSir("Nume",buf,30);
 // cauta si sterge toti angajatii cu numele dat
 for(i=0;i<nAngajati;i++){
 if(!strcmp(buf,angajati[i].nume)){
 for(j=i;j<nAngajati-1;j++) angajati[j]=angajati[j+1];
 i--;
 nAngajati--;
 }
 }
 // pentru aceasta realocare nu este necesar sa testam rezultatul,
 // deoarece ea actiuneaza doar in sensul descresterii blocului de memorie alocat,
 // deci intotdeauna va fi memorie suficienta
 angajati=(Angajat*)realloc(angajati,nAngajati*sizeof(Angajat));
 break;
 case 3:
 for(i=0;i<nAngajati;i++){
 printf("%s\t%s\t%g\n",angajati[i].nume,angajati[i].functie,angajati[i].salariu);
 }
 break;
 case 4:
 free(angajati);
 return 0;
```

```
 default:printf("optiune necunoscuta\n");
 }
 }
}
```

În această implementare, baza de date este conținută de vectorul *angajați*, realocat dinamic atât la adăugarea, cât și la ștergerea unui angajat, astfel încât să ocupe doar atâta memorie cât este necesar. Se poate însă constata că memoria nu este folosită optimal, deoarece fiecare nume și funcție ocupă o dimensiune fixă de 30 de caractere, chiar dacă informația utilă are mai puține caractere. Pentru a remedia aceasta, vom aloca dinamic și aceste două câmpuri, rezultând astfel o nouă versiune de program:

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

typedef struct{
 char *nume;  // numele si functia vor fi alocate dinamic
  char *functie;
  float salariu;
 }Angajat;

Angajat *angajati=NULL; // vector alocat dinamic de angajati
int nAngajati=0;

// elibereaza campurile alocate dinamic ale angajatului cu indexul dat
void eliberareAngajat(int idx)
{
 free(angajati[idx].nume);
 free(angajati[idx].functie);
}

void eliberare()
{
 int i;
 for(i=0;i<nAngajati;i++)eliberareAngajat(i);
 free(angajati);
}

// afiseaza un text, iar apoi citeste un sir de caractere de orice lungime
// intr-o zona de memorie alocata dinamic
// citirea se termina la \n
char *citireSir(const char *text)
{
 char ch, *dst=NULL, *dst2=NULL;
 size_t n=0;
 printf("%s: ",text);
  for(;;){
 n++;
 if((dst2=(char*)realloc(dst,n*sizeof(char)))==NULL){
 printf("memorie insuficienta");
 free(dst);
 eliberare();
 exit(EXIT_FAILURE);
 }
 dst=dst2;
 ch=getchar();
 if(ch=='\n'){
 dst[n-1]='\0';
 return dst;
 }
 dst[n-1]=ch;
 }
}

int main()
{
 int i,j,op;
 Angajat *v2;
 char *buf;
 for(;;){
 printf("1. Adaugare\n");
 printf("2. Stergere\n");
 printf("3. Listare\n");
 printf("4. Iesire\n");
 printf("Optiune: ");
 scanf("%d",&op);
 switch(op){
  case 1:
 v2=(Angajat*)realloc(angajati,(nAngajati+1)*sizeof(Angajat));
 if(!v2){
 printf("memorie insuficienta\n");
 eliberare();
 exit(EXIT_FAILURE);
 }
```

```
 angajati=v2;
getchar();
angajati[nAngajati].nume=citireSir("Nume");
 angajati[nAngajati].functie=citireSir("Functie");
printf("Salariu: ");
scanf("%g",&angajati[nAngajati].salariu);
nAngajati++;
 break;
 case 2:
getchar();
buf=citireSir("Nume");
// cauta si sterge toti angajatii cu numele dat
for(i=0;i<nAngajati;i++){
if(!strcmp(buf,angajati[i].nume)){
eliberareAngajat(i);
for(j=i;j<nAngajati-1;j++)angajati[j]=angajati[j+1];
i--;
nAngajati--;
}
}
free(buf); // continutul lui buf nu mai este necesar
// pentru aceasta realocare nu este necesar sa testam rezultatul,
// deoarece ea actioneaza doar in sensul descresterii blocului de memorie alocat,
// deci intotdeauna va fi memorie suficienta
 angajati=(Angajat*)realloc(angajati,nAngajati*sizeof(Angajat));
break;
 case 3:
for(i=0;i<nAngajati;i++){
printf("%s\t%s\t%g\n",angajati[i].nume,angajati[i].functie,angajati[i].salariu);
}
 break;
 case 4:
eliberare();
 return 0;
default:printf("optiune necunoscuta\n");
}
}
}
```

În această versiune, câmpurile *nume* și *funcție* nu mai sunt vectori ci pointeri care vor fi alocați dinamic. Pentru aceasta, funcția *citireSir* a fost modificată astfel încât să citească oricâte caractere într-un bufer alocat dinamic, până la întâlnirea \n, iar apoi să returneze buferul cu caracterele citite. Deoarece eliberarea memoriei a devenit mai complexă, iar ea se folosește în mai multe situații (ștergere, ieșire din program, eroare), pentru a se evita duplicarea de cod, s-au introdus două funcții auxiliare: *eliberareAngajat* și *eliberare*.

Acum memoria folosită este optimală, atât pentru stocarea vectorului de angajați, cât și pentru stocarea fiecărui angajat în parte. Mai putem aduce totuși o îmbunătățire, care ține de eficiența programului: în această implementare, angajații sunt stocați ca elemente ale vectorului *angajati*. Dacă fiecare angajat ocupă o dimensiune mare de memorie (în acest exemplu nu este cazul, dar pot fi structuri care au dimensiuni foarte mari) și avem mulți angajați, atunci la fiecare adăugare sau ștergere va trebui să deplasăm blocuri mari de memorie. De exemplu, dacă o structură ar ocupa 1KB și am avea 10000 de structuri, dacă ștergem prima structură din vector, va trebui să deplasăm 10MB de memorie. Pentru a îmbunătăți acest aspect, în vector nu vom păstra structuri ci pointeri către ele, structurile în sine fiind alocate dinamic. Astfel, la orice deplasare de date în vector, pentru fiecare înregistrare va trebui să mutăm doar un pointer (4 sau 8 octeți), structura în sine rămânând la poziția din memorie care i-a fost alocată dinamic. Obținem noua versiune de program:

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

typedef struct{
 char *nume;  // numele si functia vor fi alocate dinamic
  char *functie;
  float salariu;
 }Angajat;

Angajat **angajati=NULL; // vector alocat dinamic de pointeri la angajati
int nAngajati=0;

// elibereaza un angajat din memorie
void eliberareAngajat(Angajat *a)
{
 free(a->nume);
 free(a->functie);
 free(a);
}

void eliberare()
{
 int i;
 for(i=0;i<nAngajati;i++)eliberareAngajat(angajati[i]);
 free(angajati);
}

// afiseaza un text, iar apoi citeste un sir de caractere de orice lungime
// intr-o zona de memorie alocata dinamic
// citirea se termina la \n
char *citireSir(const char *text)
{
 char ch, *dst=NULL, *dst2=NULL;
 size_t n=0;
 printf("%s: ",text);
  for(;;){
 n++;
 if((dst2=(char*)realloc(dst,n*sizeof(char)))==NULL){
 printf("memorie insuficienta");
 free(dst);
 eliberare();
 exit(EXIT_FAILURE);
 }
 dst=dst2;
 ch=getchar();
 if(ch=='\n'){
  dst[n-1]='\0';
 return dst;
 }
 dst[n-1]=ch;
 }
}

int main()
{
 int i,j,op;
 Angajat **v2;
 Angajat *a;
 char *buf;
  for(;;){
 printf("1. Adaugare\n");
 printf("2. Stergere\n");
 printf("3. Listare\n");
 printf("4. Iesire\n");
 printf("Optiune: ");
 scanf("%d",&op);
 switch(op){
  case 1:
 v2=(Angajat**)realloc(angajati,(nAngajati+1)*sizeof(Angajat*));
 if(!v2){
 printf("memorie insuficienta\n");
 eliberare();
```

```
exit(EXIT_FAILURE);
}
// alocare memorie in mod individual, pentru fiecare angajat
if((a=(Angajat*)malloc(sizeof(Angajat)))==NULL){
printf("memorie insuficienta\n");
eliberare();
exit(EXIT_FAILURE);
}
 a->nume=NULL; // seteaza pe NULL campurile care se vor aloca dinamic
a->functie=NULL; // a.i. daca apare o eroare la citireSir, ele sa fie eliberate corect din memorie
angajati=v2;
 angajati[nAngajati]=a;
nAngajati++;
getchar();
a->nume=citireSir("Nume");
 a->functie=citireSir("Functie");
printf("Salariu: ");
scanf("%g",&a->salariu);
break;
 case 2:
getchar();
buf=citireSir("Nume");
// cauta si sterge toti angajatii cu numele dat
for(i=0;i<nAngajati;i++){
if(!strcmp(buf,angajati[i]->nume)){
eliberareAngajat(angajati[i]);
for(j=i;j<nAngajati-1;j++)angajati[j]=angajati[j+1];
 i--;
nAngajati--;
}
}
free(buf); // continutul lui buf nu mai este necesar
// pentru aceasta realocare nu este necesar sa testam rezultatul,
// deoarece ea actioneaza doar in sensul descresterii blocului de memorie alocat,
// deci intotdeauna va fi memorie suficienta
angajati=(Angajat**)realloc(angajati,nAngajati*sizeof(Angajat*));
break;
 case 3:
for(i=0;i<nAngajati;i++){
printf("%s\t%s\t%g\n",angajati[i]->nume,angajati[i]->functie,angajati[i]->salariu);
}
 break;
 case 4:
eliberare();
 return 0;
default:printf("optiune necunoscuta\n");
}
}
}
```

În această versiune, adăugarea unui nou angajat se face alocând memorie individual pentru acesta. În vectorul *angajați* se păstrează doar un câte un pointer la memoria alocată fiecărui angajat. Din acest motiv, *angajați* are tipul *Angajat***, deoarece el este un vector de pointeri. Funcția *eliberareAngajat* a fost modificată, astfel încât să primească un pointer la un angajat și să elibereze toată memoria ocupată de acesta. Se poate constata acum folosirea operatorului **->** (săgeată) în loc de punct, deoarece acum accesăm câmpurile structurilor prin intermediul unor pointeri la structuri. De exemplu, *angajati[i]* este un pointer la o structură (are tipul *Angajat**), astfel încât pentru a accesa câmpul *nume*, folosim *angajati[i]->nume*.

◄ Test evaluare 2

Sari la...

Teme și aplicații ►