Programarea calculatoarelor

structuri; vectori de structuri; alocarea dinamică a structurilor şi vectori de pointeri la structuri

În multe cazuri o entitate este definită prin intermediul mai multor atribute. De exemplu, un punct în plan este definit de coordonatele sale (x,y), o persoană poate fi definită prin nume, dată de naştere şi e-mail, o carte prin titlu, autor, editură, domeniu. În toate aceste situaţii, procesarea unei entităţi poate implica operaţii asupra tuturor componentelor sale. De exemplu, dacă avem o listă de persoane şi dorim să o sortăm după nume, atunci când interschimbăm două persoane în listă, trebuie să interschimbăm toate atributele celor două persoane.

În limbajul C, gruparea mai multor atribute într-o entitate de sine stătătoare este realizată folosind structuri:

```
struct Produs{
   char nume[200];
   float pret;
   int stoc;
};
```

Cuvântul cheie **struct** defineşte o nouă structură de date având numele dat şi conţinând între acolade (**{}**) toate definiţiile componentelor sale. În interiorul unei structuri se pot defini doar variabile, nu şi funcţii. La sfârşitul structurii se pune punct-virgulă(**;**). Structurile se pot defini oriunde într-un program, inclusiv în interiorul funcţiilor, dar, în general, se definesc în exteriorul lor, pentru a fi accesibile de oriunde din program. La fel ca în cazul variabilelor şi a funcţiilor, structurile trebuie mai întâi definite şi apoi folosite. Componentele unei structuri pot fi oricât de complexe: vectori, matrici, alte structuri, etc.

Definirea unei structuri creează un nou tip de date, pe care îl putem folosi la fel ca pe tipurile de date predefinite (ex: **int**, **float**, **char**). Când definim o variabilă de tipul unei structuri, numele structurii trebuie întotdeauna prefixat de cuvântul cheie **struct**, altfel se va genera o eroare:

```
struct Produs p1, p2;          // declară p1 şi p2 ca fiind variabile de tip Produs
Produs perr;                   // EROARE de compilare fiindcă lipseşte struct
p1 = p2;                       // copiaza tot conţinutul lui p2 în p1
p2.pret = 7.5;                 // atribuie campului pret al lui p1 valoarea data
```

În acest exemplu, *p1* şi *p2* sunt variabile de tipul structurii *Produs*. Fiecare dintre aceste variabile va conţine propriul său set de atribute (numite şi *câmpuri* sau *componente*) care sunt definite în structură. Variabilele de tip structură se pot folosi în mod unitar, ca un întreg, sau se poate opera asupra componentelor lor specifice.

Pentru folosire unitară, se foloseşte numele variabilei. În exemplul anterior, *p1=p2;* copiază tot conţinutul lui *p2* (toate câmpurile sale) în *p1*. Dacă dorim să accesăm individual componentele unei structuri, se va pune după numele variabilei de tip structură punct (**.**) şi apoi numele câmpului pe care dorim să-l accesăm.

```
fgets(p1.nume,200,stdin);      // citeste câmpul „nume” al variabilei p1
printf("%d", p2.stoc);         // afişează câmpul „stoc” al variabilei p2
```

La fel ca la orice tip de date, putem avea vectori de structuri, în care fiecare element al vectorului este o structură. Pentru a accesa un câmp al unui element din vector, punctul se pune după parantezele drepte,

deoarece mai întâi se izolează elementul respectiv din vector (prin indexare) și apoi se accesează în acesta câmpul cerut:

```
struct Produs produse[10];          // vector de 10 elemente, fiecare avand tipul Produs
produse[1] = produse[0];            // copiaza produsul de la poziţia 0 la pozitia 1
printf("%g", produse[5].pret);      //afisează pretul produsului de la pozitia 5
```

Structurile pot fi inițializate punând între acolade, în ordinea de la definirea structurii, valorile dorite pentru fiecare câmp:

```
struct Produs p = {”Nectar”, 7.5, 21};
 //vector de structuri: acoladele exterioare sunt pentru vector, iar fiecare element este o inițializare de structură
struct Produs produse[10] = { {”mere”, 5, 100}, {”pere”, 6.2, 70} };
```

Exemplul 12.1: Se definește o structură *Dreptunghi* care conține lățimea și lungimea unui dreptunghi. Se cere n<=10 și apoi *n* dreptunghiuri. Să se afișeze dimensiunile dreptunghiului de arie maximă.

```
#include <stdio.h>

struct Dreptunghi{
   int latime, lungime;
};

int main()
{
   struct Dreptunghi v[10];
   int i, n;
   int imax;                          // indexul dreptunghiului de arie maxima
   printf("n: ");
   scanf("%d", &n);
   for (i = 0; i < n; i++){           // citire dreptunghiuri
      printf("latime %d: ", i);
      scanf("%d", &v[i].latime);
      printf("inaltime %d: ", i);
      scanf("%d", &v[i].lungime);
   }
   imax = 0;
   for (i = 1; i < n; i++){           // cauta dreptunghiul de arie maxima
      if (v[imax].latime * v[imax].lungime < v[i].latime * v[i].lungime){
         imax = i;
      }
   }
   printf("dreptunghi de arie maxima: %dx%d\n", v[imax].latime, v[imax].lungime);
   return 0;
}
```

Se poate constata că putem folosi câmpurile unei structuri la fel ca pe orice altă variabilă. Putem să cerem adresa lor (pentru *scanf*), să facem operații aritmetice cu ele, etc.

Atribuirea unor nume tipurilor de date, folosind typedef

typedef asociază un nume unei definiții de tip:

typedef tip nume;

După acest *typedef*, *nume* va putea fi folosit pentru a substitui tipul specificat:

```
typedef char *String;          // String este un nou nume pentru tipul char*
String s1;                     // s1 are tipul char*
String v[3];                   // v este un vector cu elemente de tipul char*
```

După ce s-a folosit *typedef* pentru a se defini *String* ca fiind tipul *char**, s-a putut folosi *String* peste tot în program în locul lui *char**. În general *typedef* se foloseşte în două situaţii:

- Când avem de-a face cu tipuri mai complexe, pentru a nu trebui de fiecare dată să scriem tot tipul
- Când dorim să descriem mai bine intenţia cu care folosim un anumit tip de date

În exemplul de mai sus, numele *String* exprimă mai clar intenţia de a folosi tipul *char** pentru a stoca un şir de caractere. Pentru acelaşi tip putem avea mai multe *typedef*, fiecare exprimând o anumită intenţie. De exemplu, putem defini încă un nume pentru tipul *char**, pentru situaţiile în care folosim variabile de acest tip ca iteratori în şiruri de caractere:

```
typedef char *StrIter;         // StrIter este un nou nume pentru tipul char*
StrIter i1=v[0];               // i1 are tipul char*, şi prin numele dat stim ca va fi folosit ca iterator in siruri
```

Când definim o structură de date, numele acesteia nu este obligatoriu, ci poate să lipsească, având astfel *structuri anonime*. Aceste structuri anonime sunt tipuri de date cu care putem defini variabile ale căror tip va fi unic în tot programul:

```
struct {int x,y;} pt1,pt2;
```

În acest exemplu, *pt1* şi *pt2* sunt două variabile care sunt definite cu ajutorul unei structuri anonime. Putem astfel să ne dăm seama mai uşor că de fapt o structură este un tip de date, care începe de la cuvântul *struct* şi se încheie la acolada închisă. Din acest motiv, putem folosi *typedef* şi la structuri, pentru a da un nou nume tipului definit de ele. Structura *Dreptunghi* din exemplul anterior poate fi definită astfel, folosind *typedef*:

```
typedef struct{
   int latime, inaltime;
} Dreptunghi;

Dreptunghi v[10];
```

Am folosit o structură anonimă şi tipul de date definit de ea a primit numele *Dreptunghi*. Din acest moment, vom putea folosi numele *Dreptunghi* ca pe orice alt tip de date, fără să-i mai punem *struct* în faţă.

Aplicaţia 12.1: Se defineşte o structură *Punct* cu membrii *x* şi *y* reali. Se cere un n<=10 şi apoi *n* puncte. Să se calculeze distanţa dintre cele mai depărtate două puncte şi să se afişeze.
Notă: în antetul *<math.h>* este definită funcţia *sqrt* (square root), care se poate folosi pentru extragerea radicalului.

Structurile se pot folosi ca argumente pentru funcţii, la fel ca şi variabilele obişnuite. Ca argumente, structurile se transmit prin valoare, la fel ca tipurile de date simple (*int*, *float*, …), deci orice modificare a argumentului este locală funcţiei. O funcţie poate de asemenea să returneze valori de tip structură.

În general folosirea directă a structurilor ca parametri pentru funcţii sau ca valori returnate nu se foloseşte din cauză că, de obicei, structurile ocupă mai multă memorie şi copierea repetată a unor zone mari de memorie este

o operație destul de lentă. Din acest motiv, de obicei structurile se transmit funcțiilor prin intermediul pointerilor, rezultând astfel transfer prin adresă.

De multe ori structurile se folosesc pentru a se implementa baze de date. O bază de date simplă este constituită dintr-un vector cu elemente de tipul necesar și permite diverse operații: introducere, căutare, sortare, salvare/încărcare, etc.

Exemplul 12.2: Să se implementeze o bază de date cu produse definite prin câmpurile *nume* și *preț*. Operațiile necesare sunt *introducere, afișare și ieșire,* iar ele se vor cere de la tastatură, utilizatorului fiindu-i prezentat un meniu de unde poate alege operația dorită.

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

typedef struct{
   char nume[50];
   float pret;
} Produs;

int main()
{
   Produs produse[100];
   int i, j, op, n = 0;                    // initial 0 produse
   for(;;){
      printf("1. Introducere\n");          // afisare meniu
      printf("2. Afisare\n");
      printf("0. Iesire\n");
      printf("operatia: ");
      scanf("%d", &op);                    // cere operatia
      switch (op){
         case 1:                           // introducere produs nou
            getchar();                     // consuma \n ramas de la citirea codului de operatie
            printf("nume: ");
            fgets(produse[n].nume, 50, stdin);
            produse[n].nume[strcspn(produse[n].nume, "\n")] = '\0';
            printf("pret: ");
            scanf("%g", &produse[n].pret);
            n++;
            break;
         case 2:                           // afisare produse
            for (i = 0; i < n; i++){
               printf("%s %g\n", produse[i].nume, produse[i].pret);
            }
            break;
         case 0:
            return 0;                      // iesire din program
         default:
            printf("operatie necunscuta\n");
      }
   }
}
```

Baza de date, iniţial, nu are niciun produs (n=0). Un produs nou este adăugat pe prima poziţie liberă din vector (poziţia *n*) şi apoi se incrementează *n* pentru a se marca faptul că a crescut numărul de produse.

Aplicaţia 12.2: Să se extindă exemplul 2 cu operaţia de căutare de produs după *nume*: se cere un nume de la tastatură şi apoi se caută în baza de date. Dacă s-a găsit, se va afişa preţul său. Dacă nu s-a găsit, se va afişa textul „produs inexistent”.

Aşa cum s-a discutat anterior, deoarece structurile pot fi manevrate şi global (tot conţinutul lor deodată), la fel ca variabilele obişnuite, operaţiile de copiere ale lor pot fi scrise exact la fel cum le-am scrie pentru variabilele simple.

Exemplul 12.3: Să se modifice exemplul 2 astfel încât să se poată cere un nume de produs şi să se şteargă din baza de date toate produsele având acel nume (va fi redată numai partea specifică):

```
case 3:                                  // stergere produse (consideram ca fiind punctul 3 din meniu)
   getchar();
   char nume[50];
   printf("nume: ");
   fgets(nume, 50, stdin);
   nume[strcspn(nume, "\n")] = '\0';
   for (i = 0; i < n; i++){              // itereaza toate produsele
      if (!strcmp(nume, produse[i].nume)){   // nume identice => sterge
         for (j = i + 1; j < n; j++){        // muta la stanga produsele de dupa cel de sters
            produse[j - 1] = produse[j];
         }
         n--;
         i--;
      }
   }
   break;
```

Aplicaţia 12.3: Să se extindă exemplul 2 cu operaţia de sortare a produselor după *nume*.

Aplicaţia 12.4: Să se modifice exemplul 2 astfel încât, la adăugarea unui produs, dacă numele respectiv există deja în baza de date, acesta să nu mai fie adăugat ci să fie schimbat preţul vechi cu cel nou introdus.

Pointeri la structuri

Dacă avem un pointer *p* la o structură, un câmp *x* al structurii se poate accesa cu **„(*p).x**”, care se citeşte: **câmpul x al structurii de la adresa pointată de pointerul p**. Parantezele sunt necesare, fiindcă operaţia de selectare de câmpuri (punctul) are precedenţă mai mare decât operaţia de indirectare (steluţa). Dacă s-ar fi scris *”*p.x”*, atunci construcţia ar fi fost echivalentă cu ”*(p.x)” şi s-ar fi citit: *valoarea pointată de pointerul x, care este un câmp al structurii p.*

Deoarece accesul unui membru prin intermediul unui pointer la structură este o situaţie des întâlnită, în C există operatorul „**->**”, care combină operaţia de indirectare a pointerului la structură cu cea de selecţie a câmpului dorit. Folosind „**->**”, putem scrie „**p->x**” şi este echivalent cu ”(*p).x”

Exemplul 12.4: Se consideră o structură *Produs* care conţine un câmp *nume* şi altul *pret*. Să se scrie o funcţie care primeşte ca parametru un produs şi un procent reprezentând o reducere de preţ. Funcţia va modifica preţul produsului conform cu reducerea dată. În programul principal se vor introduce un număr *n* de produse, cu *n* citit de la tastatură. Folosind funcţia definită anterior, să se modifice preţurile produselor şi să se afişeze.

```c
#include <stdio.h>
typedef struct{
   char nume[100];
   float pret;
} Produs;

void reduce(Produs *p, float reducere)      // transmitere prin adresa a structurii, a.i. sa se poata modifica pretul
{
   p->pret *= 1 - reducere / 100;           // transforma din procente in coeficient si aplica reducerea
}

int main()
{
   int n, i;
   float r;
   Produs produse[50];
   printf("reducere: ");
   scanf("%f", &r);
   printf("n: ");
   scanf("%d", &n);
   for (i = 0; i < n; i++){                 // citire produse
      getchar();                            // citeste \n ramas de la scanf anterior
      printf("nume %d: ", i);
      fgets(produse[i].nume,100,stdin);
      produse[i].nume[strcspn(produse[i].nume, "\n")] = '\0';
      printf("pret %d: ", i);
      scanf("%g", &produse[i].pret);
   }
   for (i = 0; i < n; i++){                 // reducere
      reduce(&produse[i], r);               // transmite adresa unui element din vector si reducerea
   }
   for (i = 0; i < n; i++){                 // afisare
      printf("%s %g\n", produse[i].nume, produse[i].pret);
   }
   return 0;
}
```

Alocarea dinamică a structurilor

Putem aloca dinamic structuri, la fel ca pe orice alt tip de date, aşa 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, funcţie şi salariu. Să se implementeze o bază de date cu angajaţi, care să permită adăugarea, ştergerea şi listarea înregistrărilor, astfel încât memoria folosită să fie minimă.

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

```c
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 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:
            free(angajati);
            return 0;
        default:printf("optiune necunoscuta\n");
    }
  }
}
```

În această implementare, baza de date este conţinută de vectorul *angajaţi*, realocat dinamic atât la adăugarea, cât şi la ştergerea 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 şi funcţie ocupă o dimensiune fixă de 30 de caractere, chiar dacă informaţia utilă are mai puţine caractere. Pentru a remedia aceasta, vom aloca dinamic şi aceste două câmpuri, rezultând astfel o nouă 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 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* şi *funcţie* nu mai sunt vectori ci pointeri care vor fi alocaţi dinamic. Pentru aceasta, funcţia *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 foloseşte în mai multe situaţii (ştergere, ieşire din program, eroare), pentru a se evita duplicarea de cod, s-au introdus două funcţii auxiliare: *eliberareAngajat* şi *eliberare*.

Acum memoria folosită este optimală, atât pentru stocarea vectorului de angajaţi, cât şi pentru stocarea fiecărui angajat în parte. Mai putem aduce totuşi o îmbunătăţire, care ţine de eficienţa programului: în această implementare, angajaţii sunt stocaţi 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) şi avem mulţi angajaţi, atunci la fiecare adăugare sau ştergere va trebui să deplasăm blocuri mari de memorie. De exemplu, dacă o structură ar ocupa 1KB şi am avea 10000 de structuri, dacă ştergem prima structură din vector, va trebui să deplasăm 10MB de memorie. Pentru a îmbunătăţi 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 octeţi), structura în sine rămânând la poziția din memorie care i-a fost alocată dinamic. Obţinem 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. Funcţia *eliberareAngajat* a fost modificată, astfel încât să primească un pointer la un angajat şi 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.*

Aplicaţii propuse

Aplicaţia 12.5: O structură conţine ora (întreg) la care s-a măsurat o anumită temperatură şi valoarea acestei temperaturi (real). Se cere n<=10 şi apoi *n* temperaturi. Se cere apoi o oră de început şi una de sfârşit. Să se afişeze media temperaturilor care au fost măsurate în acel interval orar, inclusiv în capetele acestuia.

Aplicaţia 12.6: Se cere un text, citit până la \n. Să se afişeze de câte ori apare fiecare cuvânt din textul respectiv. Un cuvânt este o succesiune constituită doar din litere.

Aplicaţia 12.7: Se consideră o structură *Persoana* care are un câmp *nume* şi altul *varsta*. Să se scrie o funcţie care primeşte ca parametru o persoană şi îi modifică numele astfel încât prima literă să fie mare iar restul mici. Să se testeze funcţia cu o persoană citită de la tastatură.

Aplicaţia 12.8: Se citeşte un *n* oricât de mare şi apoi *n* puncte în plan, definite prin coordonatele lor (x,y). Să se afişeze toate punctele, grupate în seturi de puncte care sunt pe aceeaşi linie orizontală (au acelaşi y). Memoria folosită va fi minimă.