

Tehnici de programare

*structuri; vectori de structuri; alocarea dinamică a structurilor; **structuri cu câmpuri pe biți***

Structuri – recapitulare de la PC

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 naștere și e-mail, o carte prin titlu, autor, editură, domeniu. În toate aceste situații, procesarea unei entități poate implica operații asupra tuturor componentelor sale. De exemplu, dacă avem o listă de persoane și 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** definește o nouă structură de date având numele dat și conținând între acolade ({}), toate definițiile componentelor sale. În interiorul unei structuri se pot defini doar variabile, nu și funcții. La sfârșitul structurii se pune punct-virgulă(;). Structurile se pot defini oriunde într-un program, inclusiv în interiorul funcțiilor, dar, în general, se definesc în exteriorul lor, pentru a fi accesibile de oriunde din program. La fel ca în cazul variabilelor și a funcțiilor, structurile trebuie mai întâi definite și 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 și p2 ca fiind variabile de tip Produs
Produs perr;                   // EROARE de compilare fiindcă lipsește struct
p1 = p2;                       // copiaza tot conținutul lui p2 în p1
p2.pret = 7.5;                  // atribuie câmpului pret al lui p1 valoarea data
```

În acest exemplu, *p1* și *p2* sunt variabile de tipul structurii *Produs*. Fiecare dintre aceste variabile va conține propriul său set de atribute (numite și *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 folosește numele variabilei. În exemplul anterior, *p1=p2*; copiază tot conținutul 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 (.) și apoi numele câmpului pe care dorim să-l accesăm.

```
fgets(p1.nume,200,stdin);      // citește câmpul „nume” al variabilei p1
printf("%d", p2.stoc);         // afișează 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 pozitia 0 la pozitia 1
printf("%g", produse[5].pret);       // afiseaza 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 1: Se definește o structură *Dreptunghi* care conține lățimea și lungimea unui dreptunghi. Se cere $n \leq 10$ 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 folosește în două situații:

- 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 intenția cu care folosim un anumit tip de date

În exemplul de mai sus, numele *String* exprimă mai clar intenția de a folosi tipul *char** pentru a stoca un șir de caractere. Pentru același tip putem avea mai multe *typedef*, fiecare exprimând o anumită intenție. De exemplu, putem defini încă un nume pentru tipul *char**, pentru situațiile în care folosim variabile de acest tip ca iteratori în șiruri de caractere:

```
typedef char *StrIter;         // StrIter este un nou nume pentru tipul char*
StrIter i1=v[0];              // i1 are tipul char*, și prin numele dat stim ca va fi folosit ca iterator în 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* și *pt2* sunt două variabile care sunt definite cu ajutorul unei structuri anonime. Putem astfel să ne dăm seama mai ușor că de fapt o structură este un tip de date, care începe de la cuvântul *struct* și se încheie la acolada închisă. Din acest motiv, putem folosi *typedef* și 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 și 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 față.

Aplicația 1: Se definește o structură *Punct* cu membrii *x* și *y* reali. Se cere un $n \leq 10$ și apoi *n* puncte. Să se calculeze distanța dintre cele mai depărtate două puncte și să se afișeze.

Notă: în antetul *<math.h>* este definită funcția *sqrt* (square root), care se poate folosi pentru extragerea radicalului.

Structurile se pot folosi ca argumente pentru funcții, la fel ca și variabilele obișnuite. Ca argumente, structurile se transmit prin valoare, la fel ca tipurile de date simple (*int*, *float*, ...), deci orice modificare a argumentului este locală funcției. O funcție poate de asemenea să returneze valori de tip structură.

În general folosirea directă a structurilor ca parametri pentru funcții sau ca valori returnate nu se folosește din cauză că, de obicei, structurile ocupă mai multă memorie și 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 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. Afișare\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, inițial, nu are niciun produs ($n=0$). Un produs nou este adăugat pe prima poziție liberă din vector (poziția n) și apoi se incrementează n pentru a se marca faptul că a crescut numărul de produse.

Aplicația 2: Să se extindă exemplul 2 cu operația de căutare de produs după *nume*: se cere un nume de la tastatură și apoi se caută în baza de date. Dacă s-a găsit, se va afișa prețul său. Dacă nu s-a găsit, se va afișa textul „produs inexistent”.

Așa cum s-a discutat anterior, deoarece structurile pot fi manevrate și global (tot conținutul lor deodată), la fel ca variabilele obișnuite, operațiile de copiere ale lor pot fi scrise exact la fel cum le-am scrie pentru variabilele simple.

Exemplul 3: Să se modifice exemplul 2 astfel încât să se poată cere un nume de produs și să se șteargă 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;
```

Aplicația 3: Să se extindă exemplul 2 cu operația de sortare a produselor după *nume*.

Aplicația 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 prețul 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 citește: **câmpul x al structurii de la adresa pointată de pointerul p** . Parantezele sunt necesare, fiindcă operația de selectare de câmpuri (punctul) are precedență mai mare decât operația de indirectare (steluța). Dacă s-ar fi scris „ $*p.x$ ”, atunci construcția ar fi fost echivalentă cu „ $*(p.x)$ ” și 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 situație des întâlnită, în C există operatorul „ **$->$** ”, care combină operația de indirectare a pointerului la structură cu cea de selecție a câmpului dorit. Folosind „ **$->$** ”, putem scrie „ **$p->x$** ” și este echivalent cu „ $(*p).x$ ”

Exemplul 4: Se consideră o structură *Produs* care conține un câmp *nume* și altul *pret*. Să se scrie o funcție care primește ca parametru un produs și un procent reprezentând o reducere de preț. Funcția va modifica prețul produsului conform cu reducerea dată. În programul principal se vor introduce un număr n de produse, cu n citit de la tastatură. Folosind funcția definită anterior, să se modifice prețurile produselor și să se afișeze.

```

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

Exemplul 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.

```

#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 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 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 alocă dinamic și 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++)elibereAngajat(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--;
                break;
            case 3:
                listare();
                break;
            case 4:
                return 0;
        }
    }
}

```

```

        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 *functie* 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`.

Structuri cu câmpuri pe biți

Să considerăm o structură pentru memorarea unei date calendaristice:

```
struct Data{
    int zi;
    int luna;
    int an;
};
```

În această formă structura ocupă în memorie 12 octeți (3 câmpuri de tip *int* înseamnă $3 \times 4 = 12$ octeți).

Dacă analizăm mai atent lucrurile, observăm că fiecare dintre cele trei câmpuri ar putea fi memorat pe mai puțin de patru octeți. Spre exemplu, câmpul `zi` va lua valori între 1 și 31, deci ar putea fi memorat pe 5 biți (cea mai mică putere a lui 2 mai mare decât 31 este $32 = 2^5$). Câmpul `luna` va lua valori între 1 și 12, deci ar putea fi memorat pe 4 biți (cea mai mică putere a lui 2 mai mare decât 12 este $16 = 2^4$). Câmpul `an` să presupunem că va lua valori între -9999 și 9999, deci poate fi reprezentat pe 15 biți (avem din start un bit de semn deoarece vrem să putem reține și numere negative, iar cea mai mică putere a lui 2 mai mare decât 9999 este $16384 = 2^{14}$; rezultă în total 15 biți). Vedem că cele trei câmpuri împreună ar necesita de fapt doar $5 + 4 + 15 = 24$ biți, adică 3 octeți. Cu alte cuvinte folosim un spațiu de memorie de patru ori mai mare decât ar fi necesar.

În limbajul C putem specifica pentru câmpurile de tip (*unsigned*) *int* sau *char* dimensiunea lor în biți. Dimensiunea în biți se specifică plasând imediat după numele câmpului caracterul `:` (două puncte), urmat de numărul de biți pe care dorim să îl ocupe câmpul. Putem redefini structura de mai sus astfel:

```
struct DataBiti {
    unsigned int zi:5;
    unsigned int luna:4;
    int an:15;
};
```

Dacă testăm (cu ajutorul operatorului `sizeof`) dimensiunea ocupată de structura redefinită astfel, vom vedea că este într-adevăr mai mică. Totuși, deși ne-am aștepta ca ea să fie de 3 octeți, în practică ea este de 4 octeți. Aceasta se produce din motive de aliniere a variabilelor în memorie, deoarece tipul de bază al câmpurilor este (*unsigned*) *int*, care înseamnă 4 octeți. Deoarece tipurile simple sunt în general aliniate în memorie la un multiplu al dimensiunii lor, structura este extinsă la multiplu de 4 octeți.

Într-o structură pot alterna câmpurile definite pe biți cu cele definite clasic. Spre exemplu, să considerăm o structură pentru descrierea unor produse de panificație. Vom reține tipul produsului (paine, covrig sau corn), greutatea produsului în kg (număr real) și numărul de cereale ce intră în compoziția produsului (minim 0, maxim 7). Tipul produsului poate lua trei valori pe care le codificăm cu 0, 1 și 2. Ca urmare, avem nevoie de doi biți pentru memorarea tipului. Greutatea este număr real și o vom păstra într-un câmp de tip *float*. Numărul de cereale poate lua valori de la 0 până la 7, deci încapă pe 3 biți. Avem următoarea definiție a structurii:

```
struct Panificatie {
    unsigned int tip:2;           // 0-paine, 1-covrig, 2-corn
    float greutate;              // greutate in kg
    unsigned int nCereale:3;      // numarul de cereale ce intra in compozitie, maxim 7
};
```

Deoarece câmpurile pe biți nu acceptă enumerări (în mod standard), nu am putut folosi pentru câmpul *tip* o enumerare cu toate tipurile de panificație. Putem totuși să folosim și enumerări, dacă folosim conversii explicite:

```
typedef enum{TPPaine,TPCovrig,TPCorn}TipProdus;  
struct Panificatie p;  
p.tip=(unsigned int)TPPaine;
```

Primul câmp ocupă 2 biți. Al doilea câmp ocupă 4 octeți, fiind de tip *float*. Al treilea câmp ocupă 3 biți. În total avem $2+32+3=37$ biți, care încap în 5 octeți. Totuși, dacă testăm dimensiunea structurii folosind operatorul *sizeof*, vom vedea că în realitate ocupă 12 octeți. Motivul este că atunci când un câmp pe biți este urmat de un câmp definit clasic, câmpul pe biți este completat automat cu biți neutilizați până la dimensiunea tipului de bază. Cum în exemplul nostru câmpul *tip* are ca tip de bază *unsigned int*, el va fi completat cu biți nefolosiți până la 4 octeți. La fel se întâmplă și dacă ultimul câmp din structură este definit pe biți: el va fi completat cu biți neutilizați până la dimensiunea tipului de bază (*unsigned int* în cazul nostru). Pentru structura *Panificatie* harta memoriei arată astfel:

2	biți	pentru	câmpul	<i>tip</i>
30	de	biți	neutilizați	
4	octeți	(32 de biți)	pentru câmpul	<i>greutate</i>
3	biți	pentru câmpul	<i>nCereale</i>	
29	de biți neutilizați			

Avem în total $30+29=59$ de biți neutilizați. Pentru a scădea la minim risipa de spațiu, trebuie să grupăm câmpurile pe biți unul lângă altul. Rescriem structura astfel:

```
struct PanificatieOptimizat {  
    unsigned int tip:2;           // 0-paine, 1-covrig, 2-corn  
    unsigned int nCereale:3;      // numarul de cereale ce intra in compozitie, maxim 7  
    float greutate;              // greutate in kg  
};
```

De data aceasta se adaugă biți neutilizați doar după câmpul *nCereale*. Harta memoriei arată astfel:

2	biți	pentru	câmpul	<i>tip</i>
3	biți	pentru	câmpul	<i>nCereale</i>
27	de	biți	neutilizați	
4	octeți (32 de biți)	pentru câmpul	<i>greutate</i>	

Numărul biților neutilizați a scăzut la 27, iar dimensiunea totală a structurii a scăzut la 8 octeți.

Atenție la modificatorul *unsigned*. Un câmp definit pe doi biți cu modificatorul *unsigned* va reține valori de la 0 până la 3. Dacă nu apare modificatorul *unsigned*, atunci câmpul este cu semn și va reține valori de la -2 până la 1. Trebuie avut în vedere acest lucru atunci când definim structuri folosind câmpuri pe biți.

Adresele câmpurilor pe biți

Un câmp pe biți nu are adresă. Din cauză că el poate fi plasat în memorie în interiorul unui octet, nu se poate lucra cu adresa lui, deoarece adresele de memorie de fapt sunt adrese de octeți (adresele nu au granularitate mai mică de un octet). Pentru a citi valori în câmpuri pe biți folosind funcția *scanf*, se poate face citirea într-o variabilă auxiliară, iar apoi se poate copia valoarea din variabila auxiliară în câmpul structurii. Spre exemplu, următorul program generează eroare de compilare:

```
#include <stdio.h>  
  
struct Ceva{  
    int a:5;  
};
```

```
int main()
{
    struct Ceva s;

    scanf("%d", &s.a);
    return 0;
}
```

Mesajul de eroare este de genul: *error: cannot take address of bit-field 'a'*

Varianta de program corectă folosește o variabilă auxiliară pentru citire:

```
#include <stdio.h>

struct Ceva{
    int a:5;
};

int main()
{
    struct Ceva s;
    int aux;

    scanf("%d", &aux);
    s.a = aux;
    return 0;
}
```

Aplicații propuse

Probleme ușoare

Aplicația 4.0: Să se creeze o structură care reține următoarele informații despre un student: nume (maxim 20 de caractere), nota. Să se citească un element din structură și să se afișeze pe ecran.

Aplicația 4.1: Să se codifice cu ajutorul structurilor cu câmpuri pe biți informațiile despre un fișier: nume (șir de maxim 15 caractere), dimensiunea acestuia (int - număr de octeți, maxim 1024 octeți), tipul acestuia (.txt - 0, .doc - 1, .xls - 2), tipul fișierului (normal - 0, read-only - 1). Să se citească informația despre un fișier, și să se afișeze pe ecran.

Aplicația 4.2: Cu ajutorul structurii de la problema mai sus să se salveze într-un vector de structuri declarat static informații despre mai multe fișiere (maxim 10). Să se citească de la tastatură informațiile despre fișiere și să se salveze în fișierul memorie.txt.

Aplicația 4.3: Modificați problema de mai sus astfel încât vectorul de structuri să fie alocat dinamic.

Probleme medii

Aplicația 4.4: Folosind câmpuri pe biți, definiți o structură pentru memorarea următoarelor informații despre animale:

- numărul de picioare: număr întreg, minim 0 (ex. șarpe), maxim 1000 (ex. miriapod)
- greutatea în kg: număr real
- periculos pentru om: da/nu
- abrevierea științifică a speciei: șir de maxim 8 caractere
- vârsta maximă în ani: număr întreg, minim 0, maxim 2000

Unde este posibil, codificați informațiile prin numere întregi de dimensiune cât mai mică, spre exemplu “da”=1, “nu”=0. Definiți structura în așa fel încât să ocupe spațiul minim de memorie posibil. Afișați spațiul de memorie ocupat, folosind operatorul `sizeof`. Folosind structura definită, citiți de la tastatură informații despre un animal, iar pe urmă afișați-le pe ecran.

Exemplu: valoare=147000, unitate:gram => 147 kilogram

Aplicația 4.5: Să se codifice următoarele informații despre un medicament, astfel încât ele să ocupe în total un singur octet. Să se scrie un program care să citească datele unui medicament și ulterior să le afișeze.

- gradul de periculozitate: scăzută, medie, mare
- dacă se eliberează doar pe bază de rețetă: da, nu
- vârsta minimă de administrare: 1...18 ani inclusiv

Aplicația 4.6: Se consideră că o măsurătoare are asociată unitatea de măsură (gram, metru, litru) și multiplicatorul acesteia (pico, nano, mili, centi, deci, deca, hecto, kilo, mega, giga).

- Să se codifice o măsurătoare astfel încât să conțină o valoare întreagă pozitivă care să acopere un interval cât mai mare de valori, unitatea de măsură și multiplicatorul acesteia. O măsurătoare va ocupa exact 2 octeți.
- Să se citească o valoare *unsigned* și o unitate de măsură. La citire nu există multiplicator. Să se determine multiplicatorul astfel încât valoarea stocată să fie cât mai mică, fără a se pierde precizie, iar apoi să se stocheze într-o măsurătoare. Dacă valoarea nu încapă, se va afișa o eroare. Să se afișeze toate câmpurile măsurătorii.

Exemple de probleme date la examen

Subiectul 1: Să se imlementeze un program care gestionează un parc de automobile. Informațiile relative la un automobil sunt: codul mașinii (număr între 1 și 9000), numărul de locuri (între 1 și 9), puterea (în cai putere între 1 și 500), marca, culoarea, anul fabricației mașinii (între 1800 și 2017). Parcul conține n automobile, datele sunt citite de la tastatură. Să se scrie următoarele funcții de gestiune a mașinilor:

- Introducerea unui automobil nou;
- Ștergerea unui anumit automobil, al cărui cod e citit de la tastatură;
- Afișarea întregii liste de automobile;
- Afișarea pe ecran doar a automobilelor cu un anumit număr de locuri, citit de la tastatură;
- Ordonarea listei în funcție de anul fabricației, folosind `qsort`.

Să se definească structura pentru o mașină cu ajutorul structurilor cu câmpuri pe biți astfel încât spațiul ocupat să fie minim.

Subiectul 2: Să se realizeze o aplicație pentru gestionarea unei biblioteci folosind struct și union, iar dacă e posibil și structuri cu câmpuri pe biți.

Se vor memora următoarele informații pentru fiecare înregistrare:

- Titlu
- Autor
- Număr de exemplare

- Preț (exprimat în lei, număr întreg, maxim 500 lei)
- Număr de pagini (maxim 1000 pagini)
- Greutate (exprimată în grame, maxim 5000 grame)
- Dacă e Carte sau Revistă
- Dacă e carte se va memora ISBN, sir de 13 caractere, exemplu: 9783161484100, iar dacă e revistă se va memora anul în care a apărut primul număr.

Folosind un meniu să se realizeze următoarele operații:

- a) Introduceți un produs.
- b) Afișați toate publicațiile aflate în stoc care sunt cărți si au prețul peste 100.
- c) Afișați toate înregistrările.
- d) Salvați toate publicatiile - Toate produsele se vor salva într-un fișier binar (inventar.bin) folosind structura creată anterior.

Datele de intrare se vor citi de la tastatură.