

## Considerații teoretice

### Formatori:

Tutor: [Stângaciu Valentin](#)  

Tutor: [Belu Claudiu-Marcel](#)  

+3


### Data de începere a cursului:

 25.09.2023

 [Utilizatori înscrși](#)

 [Calendar](#)

 [Note](#)

 [Cursurile mele](#) [S1-L-AC-CTIRO1-PC](#) [Laborator 11: Tipuri de date definite de utilizator](#) [Considerații teoretice](#)

## Considerații teoretice

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 12.1:** 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:

venim de-a face cu tipuri mai complexe, pentru a nu trebui de fiecare dată să scriem tot tipul

orim 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 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* 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 12.1:** 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 12.2:** Să se implementeze o bază de date cu produse definite prin câmpurile *nume* și *pret*. 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 necunoscuta\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 12.2:** 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 scris pentru variabilele simple.

**Exemplul 12.3:** 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)
    getch();
    char nume[50];
    printf("nume: ");
    fgets(nume, 50, stdin);
    nume[strcspn(nume, "\n")] = '\0';
    for (i = 0; i < n; i++){
        if (!strcmp(nume, produse[i].nume)){
            // itereaza toate produsele
            // nume identice => sterge
            // muta la stanga produsele de dupa cel de sters
            for (j = i + 1; j < n; j++){
                produse[j - 1] = produse[j];
            }
            n--;
            i--;
        }
    }
    break;
```

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

**Aplicația 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 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 12.4:** 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;
}

```

## Uniuni

O uniune este o structură a cărei membri ocupă același loc în memorie. Din acest motiv, uniunile se pot folosi doar în cazurile în care membrii lor sunt *mutual exclusivi* (folosirea unuia exclude folosirea celorlalți). În această situație, uniunile sunt mai eficiente în folosirea memoriei, deoarece nu se mai rezervă memorie și pentru ceilalți membri, care sunt nefolosiți. Dimensiunea unei uniuni este dimensiunea celui mai mare membru al ei, fiindcă toți ceilalți membri, mai mici, se încadrează în această dimensiune.

O uniune se declară și se folosește la fel ca o structură, doar că în loc de *struct* se folosește **union**:

```

#include <stdio.h>

// o uniune care poate pastra fie un int fie un double
union int_sau_double {
    int n;
    double d;
};

int main(void)
{
    // O variabila de tipul uniunii definite mai sus.
    union int_sau_double u;

    // dimensiunea ocupata de uniune este maximul dintre dimensiunile campurilor sale.
    printf("dimensiune int: %d\n", sizeof(int));
    printf("dimensiune double: %d\n", sizeof(double));
    printf("dimensiune union: %d\n", sizeof(union int_sau_double));

    // daca atribuim valoare campului int, valoarea campului double devine nespecificata.
    u.n = -101;
    printf("%d %lg\n", u.n, u.d);

    // daca atribuim valoare campului double, valoarea campului int devine nespecificata.
    u.d = 77.12;
    printf("%d %lf\n", u.n, u.d);

    return 0;
}

```

Pentru citirea valorilor de tip *double* se folosește **%lg** (**long g** sau **e** sau **f**). La execuție, acest program va afișa:

```
dimensiune int: 4
dimensiune double: 8
dimensiune union: 8
-101 0.000000
343597384 77.120000
```

Se poate constata că dimensiunea uniunii este de 8 octeți pentru a putea memora cel mai mare membru al ei, "*double d*". Din cauză că cei doi membri, *n* și *d*, sunt suprapuși în memorie, în momentul în care modificăm pe unul dintre ei, se va modifica și valoarea celuilalt. Din acest motiv, vom putea folosi la un moment dat doar unul dintre membrii unei uniuni.

O uniune nu știe prin ea însăși ce membru al ei este folosit. De aceea, în general trebuie folosită o altă variabilă în care să ținem minte ce membru folosim dintr-o uniune. De obicei această variabilă, numită *selector* (*tag*), se grupează împreună cu uniunea, folosind o structură:

```
#include <stdio.h>

// cantitatea unui produs poate fi data in numar de bucati sau prin greutatea sa, exprimata in kg
union Cantitate{
    int nBucati;
    double kg;
};

struct Produs{
    char um;// unitate de masura: 'b'-la bucata, 'k'-kilograme
    union Cantitate c;
};

void afisare(struct Produs *p)
{
    switch(p->um){
        case 'b':printf("%d bucati\n",p->c.nBucati);break;
        case 'k':printf("%g kg\n",p->c.kg);break;
        default:printf("unitate de masura invalida\n");
    }
}

void introducere(struct Produs *p)
{
    printf("unitate de masura (b sau k): ");
    scanf("%c",&p->um);
    switch(p->um){
        case 'b':
            printf("numar de bucati: ");
            scanf("%d",&p->c.nBucati);
            break;
        case 'k':
            printf("greutate (kg): ");
            scanf("%lg",&p->c.kg);
            break;
        default:printf("unitate de masura invalida\n");
    }
}

int main()
{
    struct Produs p;
    introducere(&p);
    afisare(&p);
    return 0;
}
```

În acest exemplu, structura *Produs* folosește câmpul *um* pentru a se ști ce membru din uniunea *Cantitate* este folosit. Deoarece există doar două alternative pentru *um*, s-ar fi putut folosi și o variabilă booleană, de exemplu *false* indicând folosirea câmpului *nBucati*, iar *true* folosirea câmpului *kg*. În acest fel, s-ar fi putut simplifica selecția câmpului uniunii, folosind în loc de *switch* o instrucțiune de forma *if(p->um){/\*kg\*/}else{/\*nBucati\*/}*.

În general structurile se transmit funcțiilor prin adresă. În programul de mai sus avem o variabilă "*p*" de tip structură, definită în *main*, care este transferată prin adresă funcțiilor care o folosesc. Pentru accesarea unui membru al unei structuri prin intermediul unui pointer la o structură, s-a folosit operatorul "*ptr->membru*". Acesta are semnificația "selecția câmpului *membru* al structurii pointate de pointerul *ptr*". Acest operator este echivalent cu "*(\*ptr).membru*".

În caz că o uniune se folosește doar în interiorul unei structuri, putem să declarăm acea uniune și câmpul de tipul său direct în interiorul structurii respective. Mai mult decât atât, putem să ometem numele uniunii, folosind astfel o uniune *anonimă*. În exemplul de mai sus putem combina *Cantitate* cu *Produs*, folosind o singură structură cu o uniune anonimă în interior. Restul programului rămâne neschimbat:

```
struct Produs{
    char um;// unitate de masura: 'b'-la bucata, 'k'-kg
    union{
        int nBucati;
        double kg;
    }c;// cantitate
};
```

Ca orice structură, *Produs* mai poate conține și alte câmpuri, în afară de uniune și de selectorul său. De exemplu, putem să adăugăm câmpuri care sunt comune pentru orice fel de produs:

```
struct Produs{
    char nume[20];
    double pret;
    char um;// unitate de masura: 'b'-la bucata, 'k'-kg
    union{
        int nBucati;
        double kg;
    }c;// cantitate
};
```

## Enumerări (enum)

În exemplul de mai sus, am folosit constantele caracter 'b' și 'k' pentru a codifica unitățile de măsură. În caz că avem mai multe coduri sau vrem să folosim nume mai descriptive, limbajul C ne pune la dispoziție instrucțiunea **enum** (enumeration), cu ajutorul căreia putem declara mai multe constante. Sintaxa lui *enum* este:

```
enum NumeUnion{NumeCt0, NumeCt1, ..., NumeCtN};
```

*enum* are următoarele efecte:

*NumeUnion* este opțional și, dacă este dat, va denumi un nou tip de date, scris "*enum NumeUnion*", care are ca membri constantele declarate.

**enum** se poate folosi și fără un nume: *enum* {NumeCt0, NumeCt1, ..., NumeCtN};

Constantele *NumeCt0...NumeCtN* vor avea un tip întreg ales de compilator (nu neapărat *int*, poate fi și *char*), suficient de mare încât să poată memora toate valorile din *enum*.

Constantelor li se atribuie automat valori începând cu 0, în sens crescător. Astfel *NumeCt0* va fi echivalentă cu cifra 0, *NumeCt1* va fi echivalentă cu cifra 1, etc.

Dacă dorim să inițializăm cu alte valori constantele declarate în *enum*, putem specifica aceste valori în felul următor: *enum* {NumeCt0, NumeCt1=5, NumeCt2};

În această situație, *NumeCt0* va fi 0, *NumeCt1* va fi 5, iar *NumeCt2* va fi 6 (se continuă atribuirea cu următorul număr în sens crescător).

Deoarece un *enum* declară valori constante, acestora nu li se vor putea atribui alte valori. De exemplu, atribuirea ulterioară "*NumeCt0=7;*" este invalidă.

Folosind *enum*, programul de mai sus devine:

```
#include <stdio.h>

enum UnitateMasura{UMBucata,UMKg};

struct Produs{
    char nume[20];
    double pret;
    enum UnitateMasura um;// unitate de masura
    union{
        int nBucati;
        double kg;
    }c;
};

void afisare(struct Produs *p)
{
    switch(p->um){
        case UMBucata:printf("%d bucati\n",p->c.nBucati);break;
        case UMKg:printf("%g kg\n",p->c.kg);break;
        default:printf("unitate de masura invalida\n");
    }
}

void introducere(struct Produs *p)
{
    int um;// variabila auxiliara folosita pentru citire, deoarece nu se stie dimensiunea tipului enum
    printf("unitate de masura (0-nr. bucati, 1-kilograme): ");
    scanf("%d",&um);
    p->um=um;// copiere din variabila auxiliara in variabila de tip enum
    switch(p->um){
        case UMBucata:
            printf("numar de bucati: ");
            scanf("%d",&p->c.nBucati);
            break;
        case UMKg:
            printf("greutate (kg): ");
            scanf("%lg",&p->c.kg);
            break;
        default:printf("unitate de masura invalida\n");
    }
}

int main()
{
    struct Produs p;
    introducere(&p);
    afisare(&p);
    return 0;
}
```

Se poate constata că în această variantă folosirea unor nume mai descriptive atât pentru tipul de date al unității de măsură (*enum UnitateMasura*), cât și pentru unitățile de măsură în sine (*UMBucata*, *UMKg*), ajută la înțelegerea mai ușoară a programului.

Deoarece tipul întreg corespunzător lui *enum* nu este specificat (compilatorul are latitudinea să folosească orice tip convenabil), în funcția *introducere* a trebuit folosită o variabilă auxiliară *um* de tip cunoscut (*int*), pentru a se face corect citirea cu *scanf*.

La fel ca în cazul structurilor și a uniunilor, putem folosi *typedef* și pentru *enum*, pentru a atribui un nou nume tipului de date creat de *enum*, astfel încât să nu mai trebuiască să folosim "*enum Nume*" ci doar *Nume*:

```
typedef enum {UMBucata,UMKg} UnitateMasura;
UnitateMasura um;
```

Uniunile pot conține membri oricât de complecși, inclusiv alte structuri, uniuni sau vectori, pe oricâte niveluri de imbricare. Acești membri pot fi declarați fie separat și apoi grupați în interiorul uniunii, fie se pot declara direct în uniune, ca structuri sau uniuni anonime.

**Exemplu:** Să se definească o structură de date pentru memorarea unor informații despre vehicule. Structura va folosi unde este posibil uniuni, pentru a minimiza memoria folosită. Un vehicul este definit prin:

tipul motorului: benzină, motorină, electric

marca: producător și tip (ex: Dacia Logan)



tipul vehiculului: de persoane (se da numărul de locuri și de airbaguri), de marfă (se dă capacitatea și dacă este frigorific sau nu) și special (se dă numele întrebuintării specifice, ex: "tractor").

Vom implementa această structură de date în două feluri. Prima oară vom folosi structuri distincte:

```
typedef struct{
    int nrLocuri;
    int nrAirbaguri;
}VehiculPersoane;

typedef struct{
    double capacitate;
    char frigorific;// false (0), true (1)
}VehiculMarfa;

typedef enum{TMBenzina, TMMotorina, TMElectric}TipMotor;
typedef enum{TVPersoane, TVMarfa, TVSpecial}TipVehicul;

typedef struct{
    TipMotor tm;
    char marca[20];
    TipVehicul tv;
    union{
        VehiculPersoane p;
        VehiculMarfa m;
        char special[20];// "tractor", "balastiera", ...
    }specific;
}Vehicul;
```

Pentru variabile de tip boolean, ca de exemplu *frigorific*, este suficient să folosim tipul *char*, deoarece practic ele folosesc doar 1 bit pentru valoarea lor. Pentru a se citi un *char* ca valoare numerică de la tastatură (0 sau 1) se va folosi **%hhd** în *scanf* (dacă se folosește *%c* - citire cod ASCII, atunci când se va introduce 0, de fapt în variabilă se va stoca codul ASCII al lui 0, care este 48).

Pentru a nu mai fi nevoie să scriem *struct* sau *enum* înaintea tipurilor de date, am folosit pentru fiecare dintre acestea *typedef*. Dacă analizăm structurile de mai sus, putem constata că *VehiculPersoane* și *VehiculMarfa* de fapt nu sunt descrieri complete de vehicule, ci implementează doar unele caracteristici specifice. Aceste structuri se folosesc doar în interiorul structurii *Vehicul*. Din acest motiv, nu are rost să le definim în mod separat, ci le vom defini ca structuri anonime în interiorul uniunii *specific*:

```
typedef enum{TMBenzina, TMMotorina, TMElectric}TipMotor;
typedef enum{TVPersoane, TVMarfa, TVSpecial}TipVehicul;

typedef struct{
    TipMotor tm;
    char marca[20];
    TipVehicul tv;
    union{
        struct{
            int nrLocuri;
            int nrAirbaguri;
        }p;
        struct{
            double capacitate;
            char frigorific;// false (0), true (1)
        }m;
        char special[20];// "tractor", "balastiera", ...
    }specific;
}Vehicul;
```

În această structură, presupunând că avem o variabilă "*Vehicul \*v*" și dorim să accesăm câmpul *nrLocuri*, vom folosi selectorul "*v->specific.p.nrLocuri*", adică din structura pointată de pointerul *v* selectăm câmpul *specific*, apoi din interiorul câmpului *specific* selectăm câmpul *p*, și în final din *p* selectăm câmpul *nrLocuri*.

În limbajul C nu se poate afla în mod direct numele corespunzător unui membru al unei enumerări. De exemplu, dacă am dori să afișăm tipul motorului, am putea să folosim direct doar "*printf("tip motor: %d\n",v->tm);*". Se afișează 0, 1 sau 2, ceea ce nu este prea descriptiv.

Pentru a afișa un șir de caractere corespunzător unui membru de enumerare, putem folosi următoarea modalitate: declarăm un vector de nume (șiruri de caractere), care are pe fiecare poziție numele corespunzător membrului enumerării de la poziția respectivă. În acest fel, vom putea folosi variabila de tip enumerare ca index în vectorul de nume, pentru a afla numele ei corespondent:

```
typedef enum{TMBenzina, TMMotorina, TMElectric}TipMotor;
const char *numeTM[]={ "benzina", "motorina", "electric"};

void afisare(Vehicul *v)
{
    printf("tip motor: %s\n",numeTM[v->tm]);
}
```

Cea mai simplă modalitate de introducere de la tastatură a unui membru de enumerare este să-l considerăm ca fiind un cod numeric de tip *int* și să-l citim direct în enumerare:

```
void citire(Vehicul *v)
{
    int tm;
    printf("tip motor (0-benzina, 1-motorina, 2-electric): ");
    scanf("%d",&tm);
    v->tm=tm;
}
```

**Aplicația 2.1:** Pentru structura *Vehicul* de mai sus, să se implementeze funcțiile de introducere de la tastatură și de afișare.

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 definire 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 *tip*  
 30 de biți neutilizați  
 4 octeți (32 de biți) pentru câmpul *greutate*  
 3 biți pentru câmpul *nCereale*  
 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 *tip*  
 3 biți pentru câmpul *nCereale*  
 27 de biți neutilizați  
 4 octeți (32 de biți) pentru câmpul *greutate*

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

**Aplicația 12.5:** O structură conține ora (întreg) la care s-a măsurat o anumită temperatură și valoarea acestei temperaturi (real). Se cere  $n \leq 10$  și apoi  $n$  temperaturi. Se cere apoi o oră de început și una de sfârșit. Să se afișeze media temperaturilor care au fost măsurate în acel interval orar, inclusiv în capetele acestuia.

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

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

**Aplicația 12.8:** Se citește un  $n$  oricât de mare și apoi  $n$  puncte în plan, definite prin coordonatele lor  $(x,y)$ . Să se afișeze toate punctele, grupate în seturi de puncte care sunt pe aceeași linie orizontală (au același  $y$ ). Memoria folosită va fi minimă.

◀ Test grilă 3

Sari la...

Teme și aplicații ►

✉ Contactați serviciul de asistență

Sunteți conectat în calitate de

S1-L-AC-CTIRO1-PC

Meniul meu

Profil

Preferințe

Calendar

 ZOOM

Română (ro)

English (en)

Română (ro)

Rezumatul păstrării datelor

Politici utilizare site