21. CONVERSII ALE STRUCTURILOR DE DATE

21.1 Procese de conversie

Dezvoltarea domeniului IT a dus la intensificarea concurenţei pe piaţa produselor software. Apariţia la momentul potrivit a pachetului de aplicaţii care să satisfacă cerinţele clienţilor este ţelul fiecărui concurent. Din păcate unii reuşesc fructificarea momentului oportun la maximum, chiar dacă eficienţa produsului respectiv nu este dintre cele mai bune.

Eficienţa produsului software este mărită prin mai multe metode, capitolul de faţă oprindu-se la metoda conversiei structurilor de date. În dezvoltarea unui program, la un moment, este necesară realizarea conversiei, de exemplu, de la tipul real la tipul întreg. Acest lucru este folositor desfăşurării normale a algoritmului gândit de programator. Sub mediul de programare Microsoft Visual C++, conversia de la un tip la altul are o importanţă deosebită, fapt demonstrat de existenţa operatorului de *cast*. Acest operator se supraîncarcă în funcţie de necesităţile ulterioare ale programatorului.

Aparent, necesitatea operatorului *cast* este nesemnificativă. În schimb programarea sub Windows utilizează foarte des conversiile, cele mai des uzitate fiind cele ale pointerilor către anumite clase. Deci, existenţa conversiei fluidizează procesul de programare, măreşte eficienţa şi simplitatea programului.

Aşa cum se afirmă mai sus, acest capitol se opreşte asupra conversiei structurilor de date. Asemănarea dintre economie şi conversie este dată de nivelul la care se face analiza. Conversia tipurilor de date este asociată nivelului microeconomic, iar conversia structurilor de date este asociat nivelului macroeconomic. Deci, conversia structurilor de date este tratată ca un factor important în creşterea eficienţei produsului final.

Eficienţa sporită este dată şi de creşterea gradului de reutilizare a anumitor structuri de date. Să presupunem că numărul matricol al unor elevi este memorat într-un vector. Acest vector este folosit într-o aplicaţie de gestionare a elevilor şcolii respective. La un moment dat se doreşte schimbarea aplicaţiei. Programatorul care realizează noua aplicaţie doreşte păstrarea numărului matricol într-o structură de date de tip listă. Cum se poate micşora efortul de programare? Simplu, prin folosirea procedurii care realizează conversia de la un vector către o listă.

În acest caz programatorul nu mai este nevoit să reintroducă toate numerele matricole asociate elevilor, ceea ce îi reduce foarte mult efortul de programare. Existenţa unei astfel de biblioteci de funcţii şi proceduri este foarte folositoare. În continuarea capitolului sunt prezentate proceduri şi funcţii de conversie a structurilor de date realizate în limbajul de programare C.

Reprezentarea în limbajul de programare C a structurilor folosite este:

```
//lista simplu înlănțuită
struct lista
{
//informația elementului
int info;
```

```
//lista dublu înlănțuită
struct listad
{
//informația elementului
int info;
```

```
//pointer elementul următor
lista *next;
};
```

```
//pointer  elementul următor
lista* next;
//pointer  elementul anterior
lista* prev;  };
```

```
//elementul din lista arcelor
struct arc
{
//referință către nod destinație
struct nodgraf * destinatie;
//elementul următor din listă
struct arc * next_arc;
//greutatea arcului
int weight;
};
```

```
//nodul unui graf
struct nodgraf
{
//informația nodului
int info;
//nodul următor
struct nodgraf * next;
//capăt lista arcelor
struct arc * capat;
};
```

```
// nodul arborelui binar
struct arbbin
{
// informația nodului
int info;
// referință către nodul stâng şi cel drept
arbbin *ss,*sd;
};
```

Într-o aplicaţie informatică se utilizează mai multe structuri de date. Pentru a prelucra informaţii referitoare la elementele unei colectivităţi, stocarea este realizată sub forma de fişiere.

Pentru toate structurile de date sunt definite articole ce includ variabile pointer specifice legăturilor dintre elementele acestora şi informaţia utilă care descrie elementele unei colectivităţi formată din persone care lucrează într-o organizaţie, astfel:

- cod numeric personal, dată de tip vector cu 13 elemente de tip caracter;
- nume şi prenume, dată de tip vector de caractere;
- adresa, structură de tip articol formată 3 câmpuri: oraş, stradă, număr;
- vârsta, dată de tip întreg, definit pe 4 baiţi;
- salariu, dată de tip întreg, definit pe 4 baiţi.

Aplicaţia informatică pentru rezolvarea sistemului de ecuaţii AX=B de mari dimensiuni presupune:

- crearea unui fişier ce conţine elementele matricei A;
- crearea unui fişier ce conţine elementele termenului liber B;
- încarcarea în memorie sub forma unei matrici rare a elementelor matricii A (în blocuri/totalitate);
- încarcarea în memorie a elementelor vectorului de termeni liberi B (încărcarea se efectuează parţial)
- se derulează calcule cu aceste informaţii;
- se obţine soluţia sistemului de ecuaţii şi se memorează într-un fişier.

Problema se reduce la conversie de date organizate sub forma de fişier în matrice/vector, respectiv conversie din matrice/vector în fişier.

În acest capitol nu sunt descrise toate combinaţiile posibile de conversii din două motive: sunt lăsate în grija cititorului sau acestea se pot obţine combinând tipurile prezentate. De exemplu conversia unui graf în vector se realizează combinând conversia graf – matrice şi matrice – vector.

21.2 Conversie masiv unidimensional – listă simplă sau listă dublă

Conversia unei structuri de tip masiv unidimensional în listă simplă sau listă dublu înlănţuită este operaţia prin care se formează unul din cele două tipuri de liste cu elementele vectorului. Operaţia constă în citirea vectorului element după element şi construirea dinamică a listei în mod dinamic inserând la sfârşit noile elemente.

Avantajul operaţiei constă în prelucrarea ulterioară a datelor care se află în listă, prelucrare care se face mai rapid şi utilizând mai puţine resurse. În unele cazuri, elementele listelor nu trebuie să conţină numai valorile vectorului, acest lucru fiind un avantaj, deoarece se adaugă informaţii care să uşureze prelucrarea ulterioară, de exemplu un index.

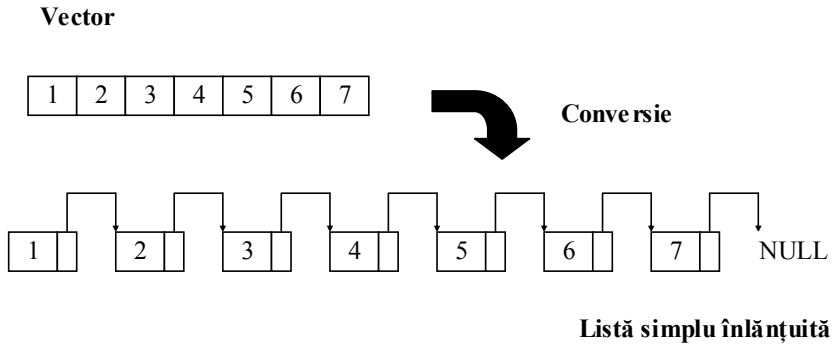

Figura 21.1 *Conversia unui vector în listă simplu înlănţuită*

Funcţia care transformă un vector într-o listă simplă primeşte ca parametrii vectorul respectiv şi dimensiunea sa. Se apelează funcţia ajutătoare *lista *inssf(lista *cap, int info)*, pentru a crea dinamic lista simplă.

```
void vector_to_lista(int v[10], int n)
{
   for (int i=0; i<n; i++) l=inssf(l,v[i]);
}

lista *inssf(lista *cap, int info)
{
   lista *temp;
   lista  *nou=(lista*)  malloc(sizeof(lista));  /*se  alocă  dinamic
spațiu pentru element*/
   nou->info=info;
   nou->next=NULL;
   if  (cap==NULL)  return  nou;  /*dacă  lista  era  vidă,  noul  element
devine capătul ei*/
```

```
   temp=cap;
   while (temp->next) temp=temp->next;
   temp->next=nou;
   return cap;
}
```

În celălalt caz, transformarea vectorului într-o listă dublu înlănţuită, se foloseşte funcţia *lista* vector_to_lista_dubla(int v[10], int n)* care returnează capătul listei şi primeşte ca parametrii vectorul de transformat şi dimensiunea sa. Lista este construită dinamic direct în corpul acestei funcţii.

```
lista* vector_to_lista_dubla(int v[10], int n)
{
   listad *temp;
   for (int i=0; i<n; i++)
   {
      listad *nou=(lista*) malloc(sizeof(lista)); /*se alocă dinamic
spațiu pentru element*/
      nou->info=v[i];
      nou->next=NULL;
      if(cap==NULL) {nou->prev=NULL; cap= nou; temp = cap;}
       nou->prev=temp;
      temp->next=nou;
      temp=nou;
   }
   return cap;
}
```

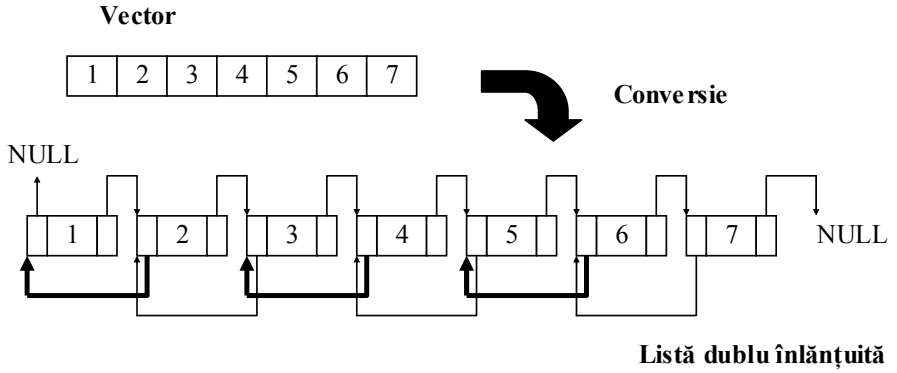

Figura 21.2 *Conversia unui vector în listă dublu înlănţuită*

Conversia în celălalt sens constă doar în parcurgerea listei şi iniţializarea elementelor vectorului. Problema care se pune în acest caz este alegerea modului de lucru. Ori se declară vectorul static cu un număr mai mare de elemente decât al listei, din motive de risipă de spaţiu şi chiar de incapacitate de realizarea a conversiei, dacă lista are mai multe elemente, sau se numără elementele listei şi se alocă dinamic spaţiu pentru vector.

21.3 Conversie masiv bidimensional – masiv unidimensional

O mare aplicabilitate are şi vectorizarea unei matrice sau conversia acesteia la vector. Aceasta s-a concretizat prin funcţia *void matrice_to_vector(int a[30][30], int n, int m, int x[], int *k)*.

Această funcţiei preia o matrice şi o transformă într-un vector de *n*m+2* componente. Traversarea matricei se face pe linii, deci în vector vor fi regăsite elementele matricei ca şi când am cap la cap fiecare linie. Dimensiunea vectorului este cu două componente mai mare decât numărul de elemente ale matricei deoarece pe ultimele două poziţii se păstrează numărul de linii, respectiv numărul de coloane ale matricei.

Este importantă păstrarea dimensiunilor matricei pentru o eventuală operaţie inversă, de la vector la matrice, sau pentru operaţiile de adunare, scădere, transpunere şi înmulţire cu matrice vectorizate.

Funcţia primeşte următorii parametrii:

- *int a[30][30]* – parametru de intrare şi reprezintă matricea care va fi vectorizată;
- *int n, m* – parametrii de intrare; reprezintă dimensiunile matricei *a*;
- *int x[]* – parametru de ieşire şi reprezintă vectorul în care va fi transformată matricea *a*;
- *int *k* – parametru de ieşire şi reprezintă dimensiunea vectorului *x*; transferul se face prin valoare.

Funcţia are următoarea structură:

```
void matrice_to_vector (int a[30][30], int n, int m, int x[], int *k)
{
int i,j;
*k=0;
for(i=0; i<n; i++)
     for(j=0; j<m; j++)
    {
          x[*k]=a[i][j];
         (*k)++;
    }
x[*k]=n;
(*k)++;
x[*k]=m;
*k=(n*m)+2;
}
```

Pentru a înţelege mai bine algoritmul, se consideră următorul masiv de 3 linii şi 3 coloane:

$$M=\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{bmatrix} \qquad (21.1)$$

În urma apelării funcţiei *matrice_to_vector* se obţine următorul vector:

Vector rezultat

1	2	3	4	5	6	7	8	9	3	3

Figura 21.3 *Vectorul rezultat prin conversie*

După cum se observă, apar două componente diferite de elementele matricei. Ele reprezintă dimensiunile matricei şi vor întotdeauna memorate pe ultimele două poziţii din vector, de unde şi dimensiunea *3*3+2* componente. Dacă operaţia nu se vrea a fi reversibilă, atunci se renunţă la ultimele două elemente ale vectorului.

21.4 Conversie listă simplă – fişier

Conversia unei liste simplu înlănţuită într-un fişier este echivalentă cu scrierea elementelor listei într-un fişier, el reprezentând la rândul său o structură de date. Această operaţie este diferită de memorarea unei liste într-un fişier, pentru că în acest caz se scriu în fişier şi informaţiile de legătură dintre elementele listei.

Funcţia care transformă o listă într-un fişier este *void lista_to_fisier(lista *cap)* şi are ca parametru de intrare adresa de început a listei. Aceasta se parcurge şi pentru fiecare element se scrie în fişier informaţia sa.

```
void lista_to_fisier(lista *cap)
{
FILE *f;
if ((f=fopen("elem.dat","wb"))==NULL)
{
     printf("\n Fisierul nu se poate deschide ");
     exit(1);
}
lista *temp=cap;
while(temp) fwrite (&temp->info, sizeof(temp->info) ,1, f);
temp=temp->next;
fclose(f);
}
```

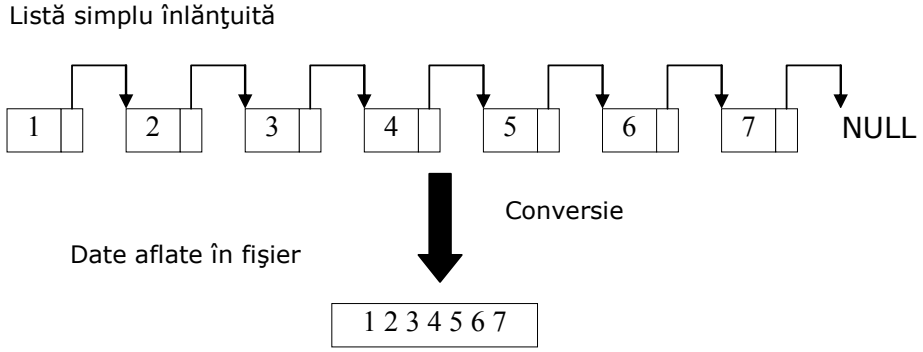

Figura 21.4 *Conversia unei liste simplu înlănţuite în fişier*

Luând lista cu elementele 1, 2, 3, 4, 5, 6, 7 şi convertind-o în fişierul *elem.dat* atunci acesta conţine doar numerele întregi 1, 2, 3, 4, 5, 6, 7 şi are dimensiunea de 7 baiţi.

21.5 Conversie fişier – listă simplă

Conversia unui fişier într-o listă simplu înlănţuită este echivalentă cu crearea unei liste de elemente care să conţină datele din fişier. Această operaţie este inversa conversiei listă – fişier.

Lista este creată dinamic, pe măsură ce sunt citite elementele fişierului. Funcţia întoarce adresa de început a listei.

```
lista * fisier_to_lista()
{
FILE *f;
int i, temp;
lista *cap=NULL;
if ((f=fopen("elem.dat","rb"))==NULL)
{
       printf("\n Fisierul nu se poate deschide ");
       exit(1);
}
for (i=0; i<n; i++) fread(&temp,sizeof(temp),1,f);
cap=inssf(cap,temp);
fclose(f);
}
```

Lucrul cu structuri dinamice presupune opera□ia de creare a structurii de date cu alocare dinamică şi temporară de memorie, efectuarea de prelucrări în condi□ii de performan□ă crescută, folosind elementele structurii dinamice şi, în final, încheierea lucrului cu structura dinamică, înso□ită de dealocarea de memorie, cu pierderea informa□iei utile rezultate din prelucrare.

Pentru a asigura conditiile continuării ciclului de prelucrări este necesar să se salveze informa□ia utilă din structura de date alocată dinamic într-un fisier.

Dacă aplica□ia utilizează liste simple sunt necesare apeluri pentru:

- copierea informa□iei utile dintr-un fişier într-o listă simplă, în vederea prelucrărilor eficiente folosind procedurile corespunzătoare opera□iilor pe liste simple;
- salvarea informa□iei utile din lista simplă alocată dinamic în fişier la sfârşitul prelucrărilor corespunzătoare aplica□iei, asigurând în acest fel condi□iile de reluare a prelucrărilor în alte momente de timp.

21.6 Conversie arbore binar – fişier

Conversia unui arbore binar într-o altă structură de date de tip fişier, constă în crearea unui fişier care să conţină informaţia nodului arborelui. Ca şi în cazul conversiei listă – fişier, conversia nu însemnă copierea sau scrierea arborelui într-un fişier, fapt care implică scrierea nodului (informaţia şi legăturile către celelalte noduri).

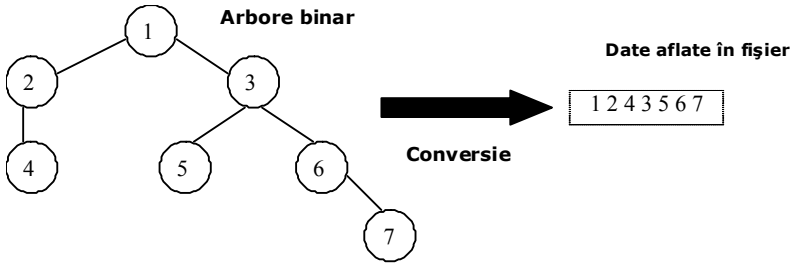

Figura 21.5 *Conversia unui arbore binar în fişier*

Rezultatul conversiei depinde în primul rând de modul în care este traversat arborele. Cum există trei moduri de parcurgere: preordine, inordine şi postordine şi datele din fişier vor reflecta acest lucru.

Funcţia care realizează acest tip de conversie este *void arbore_binar_to_fisier(arbbin *rad)*. Ea primeşte ca parametru de intrare adresa nodului părinte a arborelui.

```
void arbore_binar_to_fisier(arbbin *rad)
{
FILE *f;
if ((f=fopen("elem.dat","wb"))= =NULL) //se creează fişierul
{
      printf("\n Fisierul nu se poate deschide ");
      exit(1);    //dacă nu se deschide fişierul
}
if(rad)
{     //arborele se parcurge utilizând unul din cele 3 moduri
      fwrite(&rad->info,1,sizeof(rad->info),fis);
      arbore_binar_to_fisier(rad->ss);
      arbore_binar_to_fisier(rad->sd);
      //în acest caz, se parcurge în preordine
}
}
```

În exemplul din figura 21.5, arborele este traversat în preordine.

Problemele complexe presupun procese de căutare şi regăsire a informa□iei după o cheie în care structura dinamică arbore binar se dovedeşte a fi deosebit de eficientă.

Probleme economice reale, care reflectă colectivită□i formate din sute şi mii de persoane, conduc la construirea unor arbori binari cu sute sau mii de noduri, organiza□i pe zeci de niveluri.

Informa□ia utilă din arborii binari, care este supusă prelucrării provine dintr-un fişier, iar la sfârşitul prelucrărilor, este necesară salvarea acestei informa□ii, de asemenea, într-un fisier.

21.7 Conversie graf – matrice

Conversia unui graf într-o matrice se defineşte ca fiind operaţia de creare a matricei de adiacenţă asociată grafului respectiv.

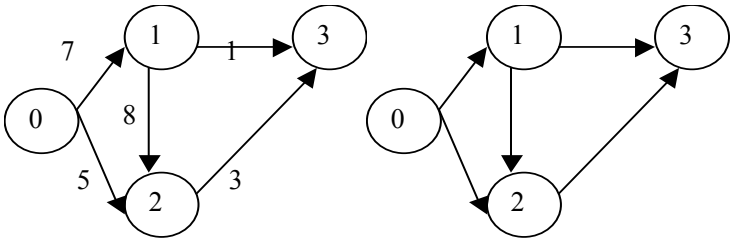

Figura 21.6 *Graf orientat cu greutate şi fără greutate*

Acest tip de conversie este utilizat în cazul lucrului cu grafuri reprezentate prin intermediul listelor de liste şi nu au asociate matrice de adiacenţă. Un motiv al necesităţii conversiei graf – matrice este dat de următorul fapt: în cazul grafului cu un număr de mic de noduri, matricea de adiacenţă ocupă un spaţiu relativ mic şi este mai uşor de lucrat cu aceasta.

Funcţia primeşte ca date de intrare pointer la capătul grafului şi construieşte matricea de adiacenţă asociată grafului. Codul sursă se bazează pe logica faptului că nodurile grafului sunt numerotate începând cu 0. Dacă de exemplu graful este compus din trei noduri notate cu 99, 100, 106, în final s-ar obţine o matrice de adiacenţă cu 107 linii şi 107 coloane.

```
void graf_to_matrice(nodgraf *cap)
{
nodgraf* p,*q;
int max,i,j;
int **matr;
if(cap==NULL) printf("\n Graful nu exista !");
else
{
      max=cap->info;
      for(p=cap; p!=NULL; p=p->next) //se caută nodul  notat
      {                              //cu cel mai mare număr de ordine
            if (max<p->info) max=p->info;
      }
      matr=alocmatr(max+1,max+1);  /*se  alocă  dinamic  memorie  ptr.
Matrice*/
      for(i=0; i<=max; i++)
            for(j=0; j<=max; j++)
                  matr[i][j]=0; //se inițializează elementele sale cu 0

      //Matricea de adiacenta asociata grafului este:
      for(p=cap; p!=NULL; p=p->next)
            for(q=cap; q!=NULL; q=q->next)
              if(q->info!=p->info)
              {
                   //dacă există arc între 2 noduri se găseşte
                   //valoarea arcului
                  matr[p->info][q->info]=verif_arc(p,q);
              }
}
}
```

Funcţia *int verif_arc(nodgraf *s, nodgraf *d)* este apelată în interiorul funcţiei precedente pentru a verifica dacă există arc între două noduri indicate prin pointer şi pentru a întoarce greutatea lui în caz afirmativ. În caz contrar funcţia returnează valoarea 0. Astfel dacă între nodurile *i* şi *j*

există arcul cu greutatea *g*, atunci elementul matricei *matr[i][j]* ia valoarea *g*.

```
int verif_arc(nodgraf *s,nodgraf *d)
{
arc * p,*aux;
int gasit=0;

for(p=s->capat; p!=NULL; p=p->next_arc)
   if(p->destinatie==d) {
      gasit=1;
      aux=p;
      return aux->weight;     //întoarce greutatea arcului
    }

if (gasit==0) {
      return 0;
}
else return aux->weight;
}
```

Funcţia *int ** alocmatr(int n,int m)*, după cum s-a observat, este folosită pentru a aloca dinamic spaţiu matricei de adiacenţă, ea întorcând un pointer la matrice. Este un mod economic de a lucra cu matrice mai ales în cazul în care nu se ştie de la început dimensiunile ei.

```
int ** alocmatr(int n,int m)
{
int **x;
int i,j;
x=(int **)malloc(n * sizeof(int));
for(i=0;i<n;i++) x[i]=(int *)malloc(m * sizeof(int));
return x;
}
```

Matricele de adiacenţă corespunzătoare grafurilor sunt :

$$M_1=\begin{bmatrix}0&7&5&0\\0&0&8&1\\0&0&0&3\\0&0&0&0\end{bmatrix} \qquad (21.2) \quad şi \quad M_2=\begin{bmatrix}0&1&1&0\\0&0&1&1\\0&0&0&1\\0&0&0&0\end{bmatrix} \qquad (21.3)$$

Funcţia prezentată este folosită în cazul grafului cu greutate, dar făcând mici modificări ea este aplicată şi grafului fără greutate. Aceste modificări constau în schimbarea funcţiei *verif_arc* astfel încât să întoarcă valoarea 1 dacă există arc între cele două noduri cercetate.

21.8 Omogenitatea prelucrărilor

Alegerea tipului de dată este determinată de domeniul căruia aparţin nivelele consemnate pentru fiecare caracteristică la descrierea elementelor unei mulţimi.

Dacă de exemplu, pentru specificarea mediului din care provin persoanele unui eşantion se înregistrează răspunsurile *mediu rural* sau *mediu urban*, domeniul asociat celor două nivele este format din elementele 0 şi 1. $D = \{0, 1\}$, unde:

- *0* – pentru mediul rural;
- *1* – pentru mediul urban.

Caracteristica aceasta se reprezintă pe un bit.

Pentru specificarea vârstei persoanelor, domeniul este cuprins între 0 şi 130, $D_V=[0,130]$. Reprezentarea vârstelor se efectuează pe zone de memorie de un bait, domeniul $D = [0, 255]$, cu $D_V \subset D$. De asemenea, numărul de copii aflaţi în îngrijirea unei persoane se reprezintă pe 5 biţi, domeniul caracteristicii *număr_copii* fiind $D_C=[0,31]$. Aceleaşi aprecieri se efectuează şi în legătură cu alte caracteristici ale altor colectivităţi.

Se obţine o diversitate de domenii, care în final determină constituirea de structuri de tip articol, cu câmpuri neomogene.

Necesitatea omogenităţii apare în procesul prelucrării.

Pentru calculul totalului unei facturi se utilizează membri definiţi în structura:

```
struct factura {
      int numar;
      struct data_emiterii dat;
      struct rand y[10];
      float total_tva;
      double total;
} f;

struct data_emiterii {
       int ziua;
       int luna;
       int anul;
};

struct rand {
        int nr_crt;
        char denum[30];
        char um[5];
        int cantitate;
        float pret_unitar;
        double pret;
        int tva;
        float tva_calculat;

        float pret_vanzare;
};
```

Se construiesc expresiile:

```
f.y[i].pret = f.y[i].cantitate * f.y[i].pret_unitar;

f.y[i].tva_calculat = f.y[i].pret * f.y[i].tva / 100;

f.y[i].pret_vanzare = f.y[i].pret + f.y[i].tva_calculat;
```

Câmpurile care intervin sunt neomogene. Astfel, *cantitate* are tipul întreg, reprezentat pe 2 baiţi. Câmpul *pret_unitar* este de tip float, reprezentat pe 4 baiţi. Rezultatul înmulţirii, *pret*, este de tip double.

Pentru evaluarea acestei expresii se efectuează conversiile din figura 21.7.

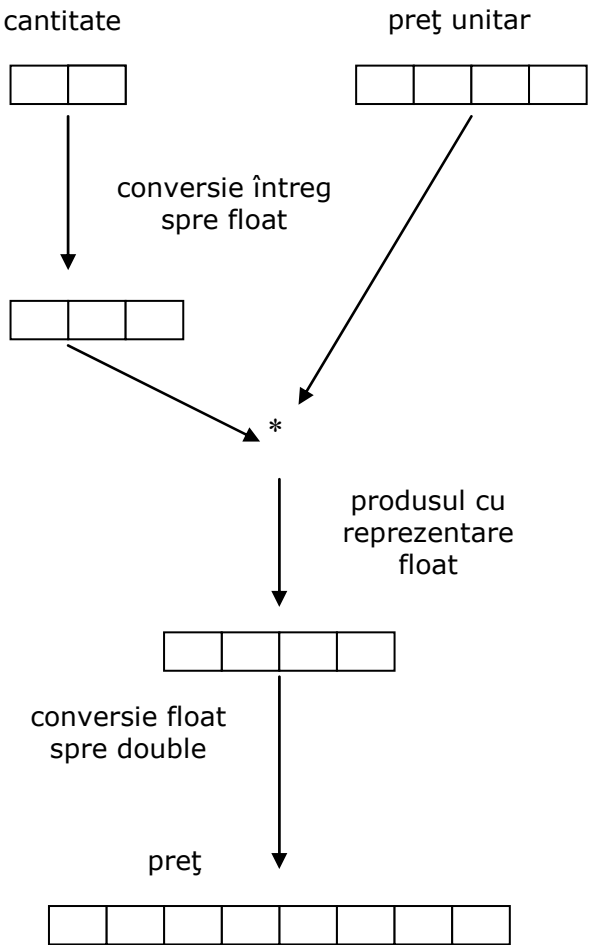

Figura 21.7 *Omogenizarea expresiei de calcul pret.*

Pentru obţinerea *tva_calculat* se face conversia din figura 21.8.

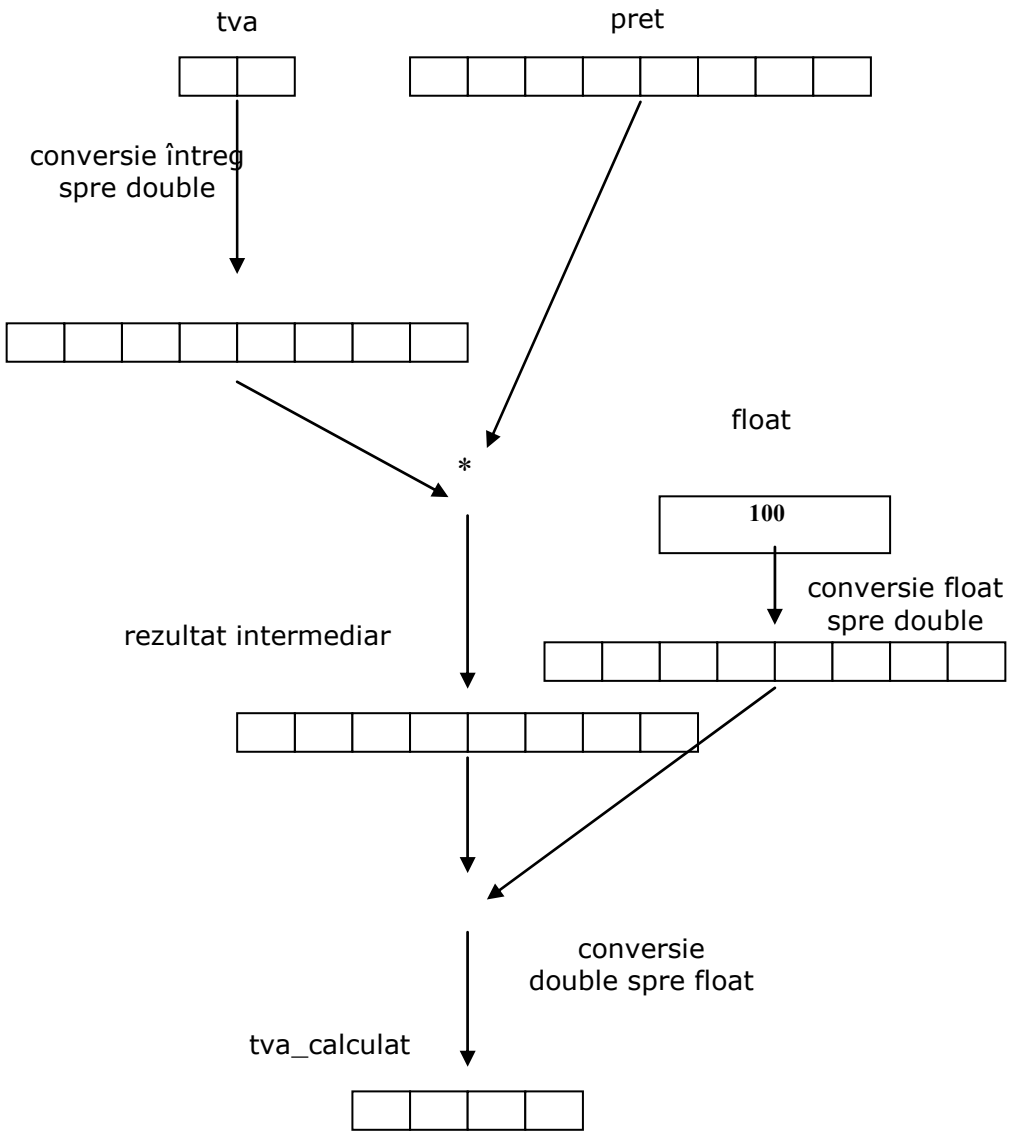

Figura 21.8 *Conversii necesare obţinerii tva_calculat*

Pentru calculul *pret_vanzare* se efectuează conversiile din figura 21.9.

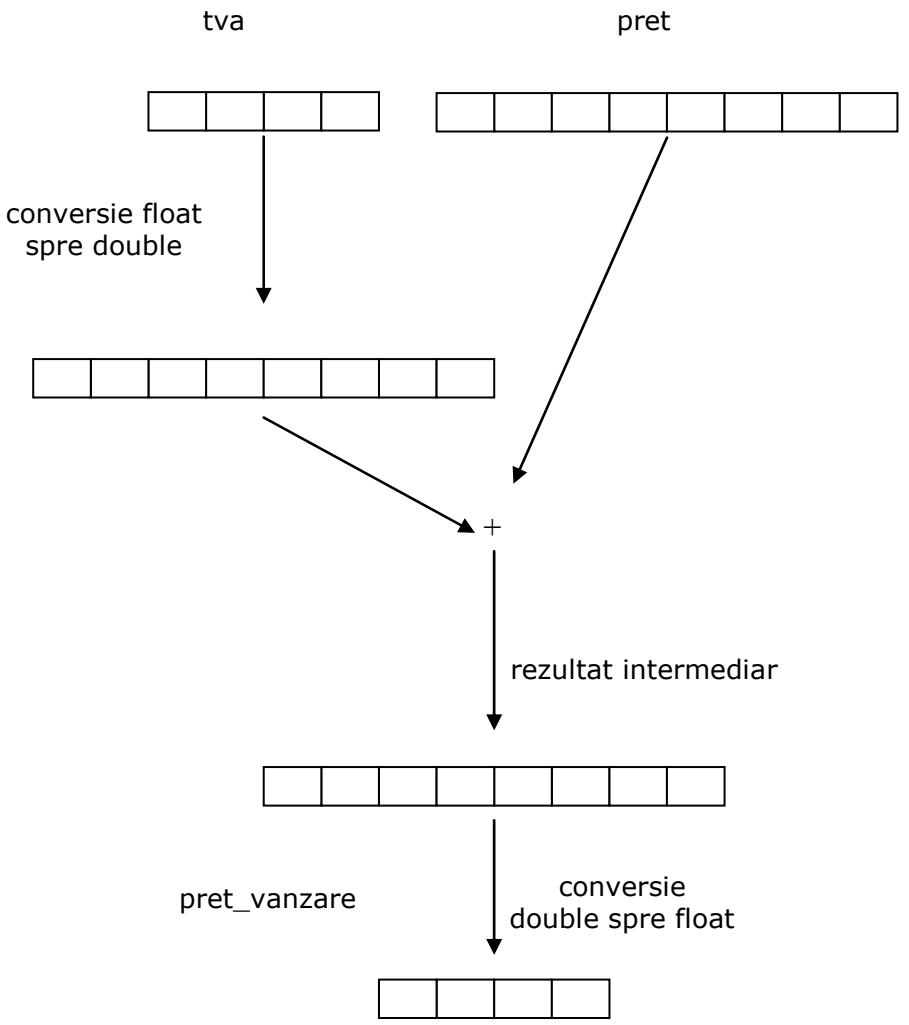

Figura 21.9 *Conversii necesare calculului pret_vanzare*

Dacă toate datele se definesc cu acelaşi tip, având un nivel de omogenitate maxim, de la încărcarea fişierelor se utilizează structura:

```
struct rand_omogen {
        int nr_crt;
        char denum[30];
        char um[5];
        double cantitate;
              double pret_unitar;
              double pret;
              double tva;
        double tva_calculat;
       double pret_vanzare;
};
```

Fişierul *FACTURI*, având iniţial articole de lungimea *sizeof(struct facturi)*, o dată cu definirea masivului *y[ ]* prin *struct rand_omogen y[10];* va avea articole de lungimea *sizeof(struct factura_omogena)*, unde:

```
struct factura_omogena {
      int numar;
```

```
        struct data_emiterii dat;
        struct rand_omogen  y[10];
        double total_tva;
        double total;
} g;
```

devenind fişierul *FACTURI_OMO*.

Dacă se consideră un număr *n* de facturi, prelucrările din fişierul *FACTURI_OMO* nu necesită conversii datorită omogenităţii datelor, în timp ce prin utilizarea fişierului *FACTURI* sunt necesare conversiile:

1. întreg *(cantitate)* spre float;
2. float *(rezultat)* spre double *(pret)*;
3. întreg *(tva)* spre double;
4. constantă float *(100)* spre double;
5. double *(rezultat)* spre float *(tva_calculat)*;
6. float *(tva_calculat)* spre double;
7. double *(rezultat)* spre float *(pret_vanzare)*;

pentru fiecare articol prelucrat, deci un total de *7*n* conversii.

Apare problema luării unei decizii în legătură, pe de o parte pentru a economisi spaţiul de memorie cu:

$$G = \left(1 - \frac{sizeof(factura)}{sizeof(factura\_omogena)}\right) * 100 \qquad (21.4)$$

dar în schimb cu un volum de *7*n* conversii, iar pe de altă parte, fără economie de spaţiu pe suport, fişierul *FACTURI_OMO* având *n*sizeof(factura_omogena)* baiţi şi prelucrările necesitând zero conversii.

21.9 Resursele conversiilor

Se consideră două alfabete *A* şi *B*, $A = \{a_1, a_2, \ldots, a_n\}$, $B = \{b_1, b_2, \ldots, b_m\}$, unde a_i reprezintă simbolul *i* al alfabetului *A*, iar b_j simbolul *j* al alfabetului *B*. Alfabetul *A* are *n* simboluri şi alfabetul *B* are *m* simboluri.

Cu alfabetul *A* se construiesc cuvinte, alcătuindu-se vocabularul V_A, cu simbolurile alfabetului *B* se alcătuiesc cuvinte care formează vocabularul V_B. Fiecare cuvânt are o lungime dată de numărul simbolurilor din care este alcătuit.

Se numeşte algoritm de conversie succesiunea de prelucrări care pornind de la un cuvânt $x \in V_A$ conduce la obţinerea cuvântului $y \in V_B$ şi succesiunea de prelucrări care pornind de la cuvântul $y \in V_B$ conduce la obţinerea cuvântului $x \in V_A$.

Când cuvintele *x* şi *y* se reprezintă sub forma unor configuraţii de biţi, lungimile cuvintelor ca număr de biţi sunt *lg(x)* şi, respectiv, *lg(y)*.

În acest fel se realizează legarea cuvintelor de zone de memorie, iar algoritmii de conversie de restricţiile acestei resurse, respectiv zona de memorie.

Definiţiile care antrenează ca resurse alfabete, zone de memorie şi seturi de prelucrări dau o maximă generalitate conversiilor.

Procesul de compresie de date este un caz particular de conversie, în care singurul obiectiv urmărit este ca pornind de la un alfabet *A* să se construiască un alfabet *W* bazat pe particularităţi de construire a cuvintelor din vocabularul V_A astfel încât, pentru orice cuvânt $x \in V_A$ să se obţină un cuvânt $y \in V_W$ aşa fel încât $lg(x) >> lg(y)$.

Algoritmul de compresie este operaţional dacă este posibilă construirea şi a algoritmului de decompresie care permite ca pornind de la cuvântul $y \in V_W$ să se obţină cuvântul $x \in V_A$.

În cazul în care cuvântul *x* este chiar fişierul însuşi, *y* devine fişierul compresat.

Manipularea zonelor de memorie impune gestionarea parametrului de definire a acesteia, lungimea, ca număr de baiţi.

Conversie de lungime

Fie un cuvânt $x \in V_A$ şi $lg(x) = k$ biţi.

Dacă se defineşte o zonă de memorie *Z* având *m* biţi, $m > k$, se pune problema memorării cuvântului *x* în *Z*.

Instrucţiunile de atribuire sau de transfer din limbajele evoluate realizează conversiile în diferite moduri.

Dacă în limbajul COBOL era definit:

```
77 x1 PIC 9(5) VALUE 12345.
77 x2 PIC 9(7).
```

instrucţiunea:

```
MOVE  x1 TO x2
```

realizează conversia de lungime cu generare de zerouri în partea stângă a configuraţiei iniţiale de biţi, figura 21.10:

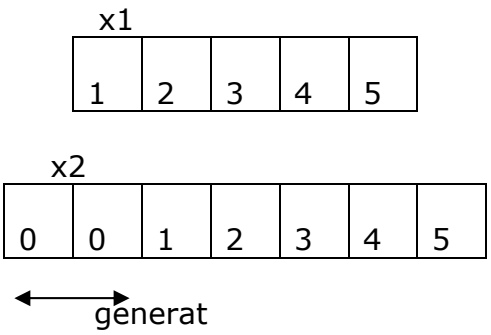

Figura 21.10 *Conversie pe lungime date numerice zecimal despachetat*

În cazul în care tipul de dată este alfanumeric la conversia de lungime se generează configuraţia de biţi asociată caracterului *spaţiu* 32h.

Pentru definirea în limbajul COBOL:

```
77 y1 PIC X(4) VALUE 'ABCD'.
77 y2 PIC X(7).
```

instrucţiunea:

```
MOVE y1 TO y2
```

conduce la obţinerea configuraţiei de biţi din figura 21.11:

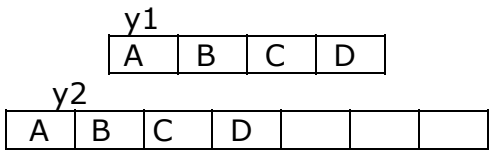

Figura 21.11 *Conversie pe lungime şir de alfanumerice*

Conversia numerică

Cele mai frecvente conversii sunt cele numerice.
Se consideră următoarele tipuri de date:

întreg
virgulă mobilă simplă precizie
virgulă mobilă dublă precizie
zecimal împachetat
zecimal despachetat
caractere numerice
hexazecimal

Pentru toate numerele există o reprezentare sub formă de configuraţie de biţi. Reprezentarea unui număr în zone de memorie este dată în tabelul 21.1.

Tabelul nr. 21.1 *Reprezentarea valorilor numerice*

Valori / Tip	**25**	**-25**	**1**	**-1**	**0**
întreg	0019	FFE7	0001	FFFF	0000
virgulă mobilă simplă precizie					00000000
virgulă mobilă dublă precizie					0000000000000000
zecimal împachetat	0205	S0205	01	S01	00
zecimal despachetat	0025	S0025	01	S01	00
caractere numerice	3235	2D3235	31	2D31	30

De la tastatură de introduc în zonele de memorie şiruri de caractere. Funcţiile de I/E realizează conversii. În acest scop se folosesc descriptori. Astfel, descriptorul *%d* realizează conversia de la şir numeric-caracter spre întreg binar.

Descriptorul *%f* activează realizarea conversiei de la şir de caractere numerice spre tipul *float*.

Pentru realizarea conversiei unui şir de simboluri cifrice într-un întreg binar este utilizată funcţia:

```
int atoi( char sir[] )
 {
  int i,numar,semn;
  for(i=0;sir[i]<'0' || sir[i] >'9' ||
sir[i]!='-' || sir[i]!='+';i++);
  if(sir[i]=='-') { semn=-1; i++;}
  if(sir[i]=='+') { semn=1; i++;}
  for(numar=0; sir[i]>='0' && sir[i]<='9'; i++)
      numar=numar*10+(sir[i]-'0');
 numar*=semn;
 return numar;
}
```

Pentru conversia de la întreg binar spre sir de simboluri numerice este folosită funcţia:

```
itoa(int numar,char sir[])
{
  int i, j, semn, lungime, carc;
  if(n<0) {
        semn=-1;
        numar=-numar;
  }
  i=0;
  do
  {
    sir[i++]=numar%d+'0';
  }
  while((n/10)>0 );
  if(semn <0) sir[i++]='-';
  else sir[i++]='+';
  sir[i]='\0';
  lungime=strlen(sir)-1;
  for(i=0, j=lungime; i<j; i++, j--)
  {
     carc=sir[i];
     sir[i]=sir[j];
     sir[j]=carc;
  }
}
```

Există o diversitate de funcţii pentru conversie.

Ipotezele de lucru pentru realizarea conversiei şi ceea ce returnează reprezintă un aspect esenţial. Astfel, pentru conversia unui şir de simboluri numerice se consideră şirul declarat prin:

```
char sir[];
```

a cărui lungime trebuie impusă să nu depăşească *k* poziţii.

Dacă este vorba de o conversie de la şir de simboluri numerice spre întregul binar *numar*, definit pe cuvânt şi *-32768 ≤ cont(numar) ≤ 32767*,

şirul de tip *char* trebuie să conţină secvenţe de simboluri numerice care să nu depăşească lungimea 5.

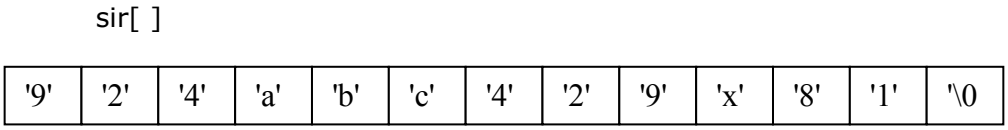

Figura 21.12 *Secvenţe de simboluri numerice*

Traversarea de la stânga la dreapta a masivului unidimensional permite extragerea simbolurilor *9, 2, 4* şi constituirea numărului întreg *924* în reprezentarea binară din zona de memorie asociată variabilei *x*.

Delimitatorul de sfârşit al secvenţei este un caracter diferit de simbol cifric.

De asemenea, poate fi interpretat delimitator de sfârşit de secvenţă însuşi delimitatorul de sfârşit de şir *'\0'*.

Funcţia iniţializează cu zero variabila *y* în cazul în care şirul are structura dată în figura 21.13.

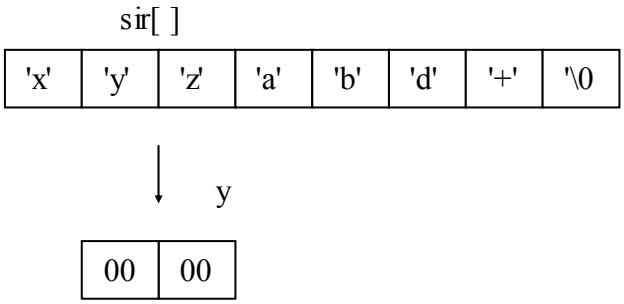

Figura 21.13 *Iniţializarea cu zero în cazul secvenţei vide de simboluri cifrice*

Se introduce o ambiguitate, în ceea ce priveşte imposibilitatea de a face o conversie şi cazul real în care secvenţa conţine o succesiune de simboluri de *0*.

Lipsa testelor privind simbolurile cifrice şi lungimea secvenţei face imposibilă introducerea în limitele specifice tipului int.

Este utilă construirea funcţiei de evaluare a lungimii secvenţei consecutive de simboluri numerice *nstrlen( )*.

```
int nstrlen (char sir[])
{
  int i,k;
  i=k=0;
  while(sir[i+1]!='\0')
  {
    if(sir[i]>='0' && sir[i]<='9'  &&
(sir[i+1]>='0' && sir[i+1]<='9'))
        k++;
    i++;
 }
 return ((k) ? k++ : k);
}
```

Funcţia returnează lungimea secvenţei de simboluri cifrice şi se utilizează pentru testarea din interiorul funcţiei *atoi( )*:

```
      …
if(nstrlen(sir)>4) return 0;
if(nstrlen(sir)==4 && sir[i]-'0'>3) return 0;
if(nstrlen(sir)==4 && sir[i]-'0'>2) return 0;
if(nstrlen(sir)==4 && sir[i]-'0'>7) return 0;
if(nstrlen(sir)==4 && sir[i]-'0'>6) return 0;
if(nstrlen(sir)==4 && sir[i]-'0'>7) return 0;
      …
```

Conversia începe numai după ce s-a efectuat încadrarea între limitele domeniului asociat tipului *int*, respectov *[-32768, 32767]*.

În acest caz, funcţia nu returnează şi o informaţie privind modul în care s-a efectuat conversia.

Este de dorit să se construiască funcţii de conversie care să ofere mai multe informaţii, mai ales că funcţiile nu sunt apelate oricum.

Astfel, este posibilă identificarea următoarelor situaţii:

- secvenţă vidă de simboluri cifrice;
- secvenţă de simboluri cifrice mai mare ca 4;
- secvenţa de 4 simboluri cifrice care prin conversie conduce la o valoare din afara intervalului [-32768, 32767].

Procedura care efectuează conversia de la şir de simboluri cifrice, *sir[]* spre tipul zecimal despachetat în şirul *upk[]*, este:

```
int atoupk(char sir[],char upk[])
{
 int i;
 i=0;
 while(sir[i]!='\0')
 {
   upk[i] = sir[i] - '0';
   i++;
 }
 upk[i]='*';
 return i;
}
```

Funcţia returnează lungimea şirului convertit.

Funcţia care efectuează conversia numărului zecimal despachetat spre şir de simboluri cifrice este:

```
int upktoa(char upk[],char sir[])
{
   int i;
   i=0;
   while(upk[i]!='*')
  {
     sir[i]=upk[i]+'0';
     i++;
```

```
  }
  sir[i]='\0';
  return i;
}
```

S-a ales delimitatorul de sfârşit al numărului zecimal despachetat caracterul *'*'* pentru a nu se denatura conversia din cauza coincidenţei caracterului *'\0'* cu zero binar.

Funcţia care efectuează conversia unui şir de simboluri cifrice într-un număr zecimal împachetat este:

```
int atopk(char sir[],char pk[])
{
   int i,k;
   k=i=0;
   while(sir[i]!='\0')
   {
        pk[k]=sir[i]-'0';
        if(sir[i+1]!='\0')
     {
       pk[k]<<4;
       pk[k]|=sir[i+1]-'0';
       k++;
       i++;
     }
  }
 return k;
}
```

Masivul unidimensional *sir[]* trebuie să conţină în mod obligatoriu simboluri cifrice pentru ca rezultatul conversiei să fie corect. Pentru şirul *'A2 B4F8\0'* se va obţine prin conversie şirul:

pk[0]=0xA2
pk[0]=0x20 prin pk[0]<<4
pk[0]=0xA4 prin 0xB4-0x30=0x84
0x84+0x20=0xA4.

O ameliorare constă în operarea cu masca *0x0F*, funcţia devenind:

```
int atopk(char sir[],char pk[])
{
 int i,k;
 k=i=0;
 while(sir[i]!='\0')
   {
     pk[k]=sir[i]&0x0F;
     if(sir[i+1]!='\0')
    {
       pk[k]<<4;
       pk[k]|=sir[i+1]&0x0F;
```

```
        k++;
        i++;
       }
    }
 return k;
}
```

Puterea unor limbaje de programare este dată în primul rând de diversitatea de tipuri de date şi de diversitatea conversiilor acceptate.

Când se prezintă limbajele este necesară definirea tabelelor de conversie, de forma tabelului 21.2.

Tabelul nr. 21.2 *Conversii posibile*

TIP	**Tip_1**	**Tip_2**	**...**	**Tip_j**	**...**	**Tip_k**	**...**	**Tip_n**
Tip_1								
Tip_2								
...								
Tip_i				D		N		
...								
Tip_n								

Simbolul *D* arată că este permisă conversia de la tipul Tip_i la tipul de dată Tip_j. Simbolul *N* arată că nu este permisă conversia de la tipul de dată Tip_i la tipul de dată Tip_k.

Conversia matricelor

Matricele în care un număr important de elemente, peste 70%, sunt nule, necesită un alt fel de stocare.

O posibilitate de conversie a matricelor revine la a considera o matrice *A* rară încărcată în memorie şi de a trece de la aceasta la o listă simplă ale cărei elemente conţin numai informaţii care prezintă valori nenule.

Dacă se consideră o matrice de mari dimensiuni cu m linii şi n coloane, mai întâi aceasta este stocată în fişierul F, după care programul de conversie realizează citirea de date din fişier şi transpunerea acestora într-una din forme de prezentare în memorie a matricei de mari dimensiuni, figura 21.14.

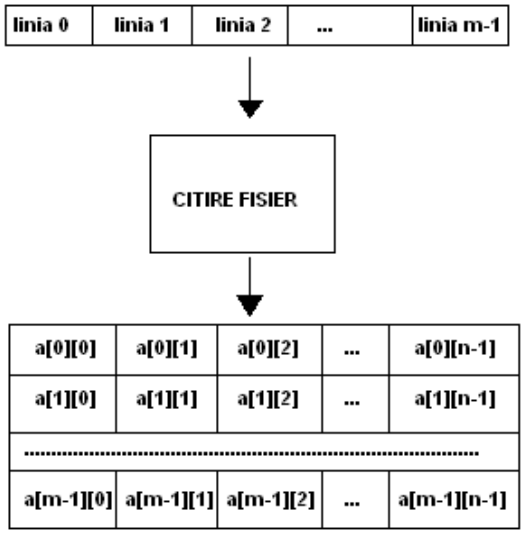

Figura 21.14 *Conversia fişier-matrice*

Pentru conversia matricei sub forma de fişier, figura 21.15, trebuie specificat modul în care sunt preluate elementele matricei, fie linie de linie, fie coloană de coloană.

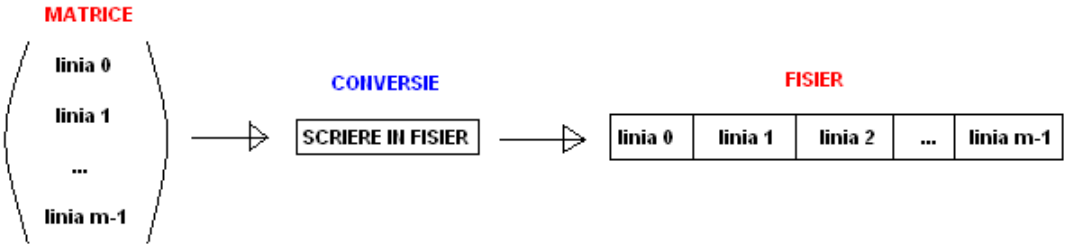

Figura 21.15 *Conversia matrice-fişier*

Problemele de optimizare, probleme de calcule inginereşti complexe, presupun lucrul cu matrice de dimensiuni mari, matrice pentru care trebuie asigurată completitudinea şi corectitudinea datelor.

Există foarte mari riscuri legate de procesul de asigurare a calită□ii con□inutului unei matrice, fapt pentru care încărcarea elementelor matricei şi verificarea acestora trebuie să respecte reguli stricte de management a con□inutului.

Matricea încărcată în memoria internă este supusă prelucrărilor, ob□inându-se o nouă matrice sau valori agregate.

Este necesar ca matricele complete şi corecte să fie salvate în fişier.

Fişierele care con□in matrice sunt folosite pentru a încărca matricea în memorie în vederea utilizării acestei proceduri de prelucrare.

Conversia listelor simple şi duble

În cazul în care se doreşte conversia de la structură de tip listă simplu înlănţuită, la cea de tip listă dublu înlănţuită, se parcurg nodurile listei şi se copiază informaţia utilă în nodurile listei duble, figura 21.16.

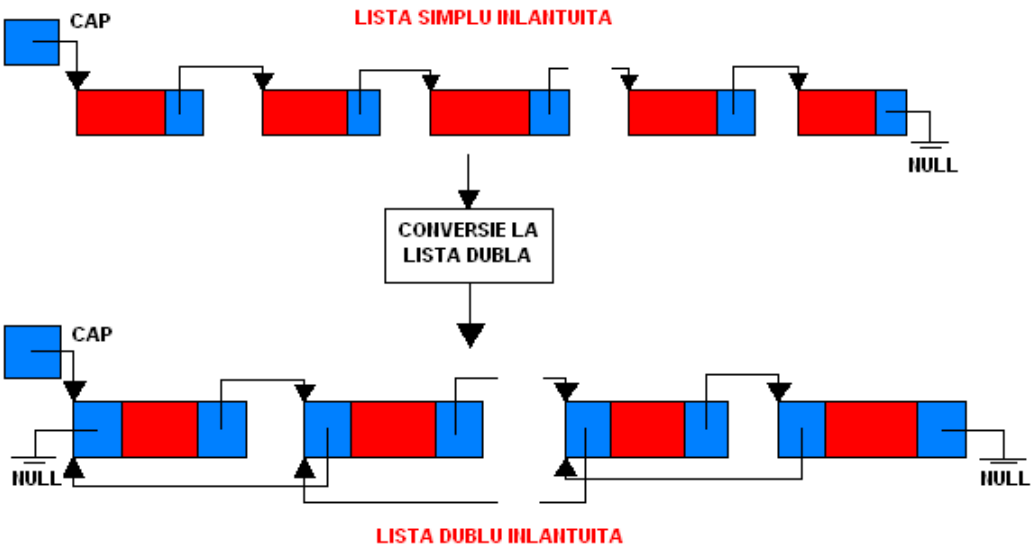

Figura 21.16 *Conversia listă simplu înlănţuită- listă dublu înlănţuită*

Pentru lista simplu înlănţuită definită mai jos:

```
struct lista_simpla
{
            float info;
            lista_simpla* next;
};
```

se va implementa o procedură de conversie în listă dublu înlănţuită, cu următoarea structură:

```
struct lista_dubla
{
            float info;
            lista_dubla *prev,*next;
};
```

Dacă se consideră structurile de date S_1, S_2, …, S_N problema conversiei revine la a scrie proceduri de reprezentare a informa□iei utile din structura S_i într-o structură S_j.

Procedura are rolul de a crea şi ini□ializa variabile pointer care să ajute la traversarea corectă în structura S_j folosind reguli deja definite pentru structura S_i.

Sunt necesare n*n-n proceduri, întrucât conversia de la S_i la S_i nu se justifica.

Dacă se consideră o structură de bază S_b şi se doreşte conversia de la structura S_i la structura S_j prin intermediul structurii de bază, se vor scrie n-1 proceduri de conversie de la structura S_i la structura S_b, respectiv, tot n-1 proceduri de conversie de la structura S_b la structura S_j. În total, vor fi necesare 2(n-1) proceduri de conversie, şi nu n*n-n.

Conversia listei simple spre arbore binar se implementează folosind o procedură care copiază informa□ia utilă a elementului din lista simplă în partea destinată informa□iei utile din nodul arborelui binar. Variabilele pointer ale arborelui binar se ini□ializează la construirea acestuia, astfel încât să se asigure traversarea corectă a nodurilor.

Variabila pointer din lista simplă nu este preluată în arborele binar datorită volatilită□ii procesului de alocare dinamică a memoriei, nefiind utilizabil la o conversie de la arborele binar spre lista simplă.

Pentru conversia listei duble în listă simplă, se utilizează procedura care:

- traversează lista dublă;
- creează un element al listei simple;
- asigură legatura cu elementul precedent al listei simple;
- copiază informa□ia utilă din elementul listei duble în elementul creat al listei simple;
- asigură ini□ializarea variabilei pointer de legatură;
- reia procesul.

Pentru conversia listei duble în fişier este necesar ca procedura să realizeze:

- traversarea listei duble;
- preluarea informa□iei utile din lista dublă în articolul fişierului;
- scrierea articolului în fişier;
- reluarea procesului până la încheierea procesului de traversare.

Conversia listei duble în arbore binar constă în:

- traversarea listei duble;
- crearea unui nod din arborele binar prin alocare dinamică de memorie;
- asigurarea legăturii cu nodul părinte;
- copierea informa□iei utile din lista dublă în zona de memorie aferentă nodului creat;
- reluarea procesului de traversare a listei duble până la epuizarea elementelor acesteia.

Conversia de fişiere

În cazul în care există două sisteme de operare S_1 şi S_2, avem algoritmi de conversie pentru tipurile de date diferite, moduri diferite de gestionare a pozi□iei semnelor, a mantisei, a caracteristicii, lungimi diferite ale cuvintelor şi modalită□i specifice de construire a etichetelor, a delimitatorilor de articole.

Fiecare suport tehnic de stocare a fişierelor are caracteristicile sale.

Conversia de suport revine la a realiza un fişier F_2 pe suportul X pornind de la fişierul F_1 aflat pe suportul Y. Astfel de probleme apar la trecerile spre noi generații de purtători tehnici de informa□ie.

Recensămintele anilor ’70 au fost realizate cu genera□ia a III-a de calculatoare, iar fişierele care con□in fondul de date se află pe benzi magnetice. Filosofia noilor calculatoare impune o altă abordare. Pentru a efectua compara□ii, ori se reintroduc datele din documentele primare, dacă mai există, ori se face conversie fişiere, de suport.

Pornind de la structura fişierelor pe bandă magnetică, de la reprezentarea informa□iei pentru genera□iile de calculatoare din anii ’70, se vor scrie programe care să convertească aceste fişiere.

Un alt mod de a privi conversia de fişiere constă în elaborarea unui program care are ca intrare un fişier F_1 realizat pentru a efectua anumite prelucrări şi a prelua din fişier anumite articole, anumite câmpuri, a le modifica aşa încât să se ob□ină la ieşire un fişier F_2. Fişierul F_2 este fişier de intrare pentru un program care efectuează prelucrări.

Situa□ia este des întâlnită. Fişiere realizate în FoxPro trebuie prelucrate în programe C++.

Pentru conversia de fişiere, sistemul FoxPro dispune de comanda *IMPORT FROM*.

Se consideră un program care efectuează calcule de salarii. Fişierul de intrare con□ine:

- numărul de intervale pentru diferențierea procentului de impozitare: int n; 0<n<20;
- limitele superioare ale intervalelor: float x[20];
- articolele fişierului ce conţin date despre salariaţi, având structura:

```
struct salariat {
      int marca;
      char nume[30];
      int vârsta;
      int timp;
      float salariu_orar;
      float reținеri;
      float sporuri;
};
```

Fişierul iniţial conţine numai articole cu date ale salariaţilor construite după structura:

```
struct sal {
      int marca;
      char nume[20];
      int timp_lucrat;
      int salariu_orar;
      int reținеri;
};
```

Lipsesc câmpurile *vârsta* şi *sporuri*.

La conversia de fişiere nu se realizează completarea cu valori câmpurilor, ci se alocă numai spaţiu, eventual iniţializat cu zero. Procedurile de conversie efectuează trecerea de la structura articolului *sal* la structura articolului *salariat*, figura 21.17:

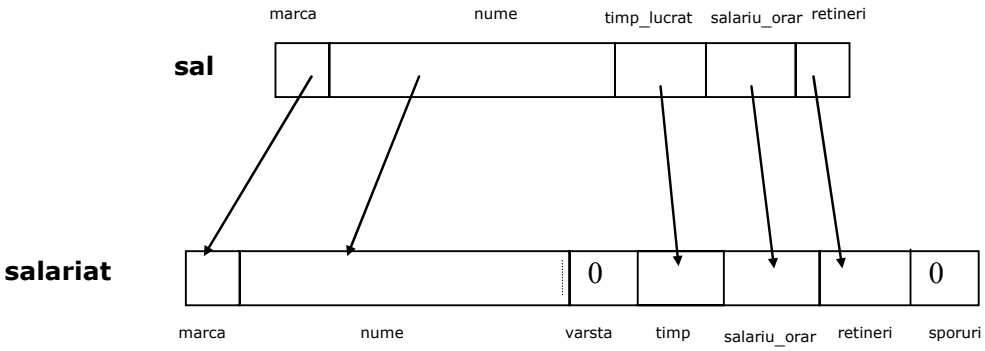

Figura 21.17 *Conversia de structură a articolelor unui fişier*

Programul C++ care efectuează conversia acestor fişiere este :

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

 struct salariat {
      int marca;
      char nume[30];
      int varsta;
      int timp;
      float salariu_orar;
      float retineri;
      float sporuri;
};

struct sal {
      int marca;
      char nume[20];
      int timp_lucrat;
      int salariu_orar;
      int retineri;
};

void main()
{
  sal S1;
  salariat S2;
  FILE *pf1,*pf2;
  pf1=fopen("sal.dat","rb");
  pf2=fopen("salariat.dat","wb");
  S2.varsta=0;
  for(int i=20; i<30; i++)
      S2.nume[i]=' ';
  while(fread(&S1,sizeof(struct sal),1,pf1))
{
        S2.marca=S1.marca;
        strncpy(S2.nume,S1.nume,20);
        S2.timp=S1.timp_lucrat;
        S2.salariu_orar=S1.salariu_orar;
        S2.retineri=S1.retineri;
        fwrite(&S2,sizeof(struct salariat),1,pf2);
}
 fclose(pf1);
 fclose(pf2);
}
```

Pentru a reduce volumul conversiilor în timpul execuţiei programelor se efectuează conversia de câmpuri în fişiere.

Se consideră seriile de timp T_1, T_2, T_3, T_4 având n termeni. Un articol din fişierul F_1 care le descrie are forma:

```
struct articol {
      int anul;
      int t1;
      float t2;
      int t3;
      float t4;
};
```

Pentru calculul valorilor medii ale celor patru serii se impun conversii. Cel mai simplu este să se efectueze conversiile câmpurilor din articole, ajungându-se la structura:

```
struct art {
      int anul;
      float t[4];
};
```

În loc de efectua prelucrările cu două funcţii din programul:

```
#include <stdio.h>
#define N 10

struct articol {
      int anul;
      int t1;
      float t2;
      int t3;
      float t4;
};

float media(int x[], int n)
{
   int s=0;
   for(int i=0; i<n; i++)
      s+=x[i];
    return (float)s/n;
}

float media(float x[], int n)
{
   float s=0.0;
   for(int i=0; i<n; i++)
      s+=x[i];
  return s/n;
}

void main()
{
 float medii[4];
 int T1[N],T3[N];
 float T2[N],T4[N];
 articol A;
 FILE *pf;
 pf=fopen("serii.dat","rb");
 for(int i=0;i<N;i++)
 {
      fread(&A,sizeof(struct articol),1,pf);
      T1[i]=A.t1;
      T2[i]=A.t2;
      T3[i]=A.t3;
      T4[i]=A.t4;
 }
 fclose(pf);
 medii[0]=media(T1,N);
 medii[1]=media(T2,N);
 medii[2]=media(T3,N);
```

```
 medii[3]=media(T4,N);
 for(i=0;i<4;i++)
 printf("\n media seriei %d este %f",i+1,medii[i]);
}
```

se va efectua aceeaşi prelucrare cu programul:

```
#include <stdio.h>
#define N 10

struct art {
            int anul;
      float t[4];
};

float media(float x[], int n)
{
 float s=0.0;
 for(int i=0; i<n; i++)
       s+=x[i];
 return s/n;
}

void main()
{
 float medii[4];
 float T[4][N];
 art A1;
 FILE *pf;
 pf=fopen("serii1.dat","rb");
 for(int i=0; i<N; i++)
 {
     fread(&A1,sizeof(struct art),1,pf);
    T[0][i]=A1.t[i];
    T[1][i]=A1.t[i];
    T[2][i]=A1.t[i];
    T[3][i]=A1.t[i];
 }
 fclose(pf);
 for(i=0; i<4; i++)
 {
     medii[i]=media(T[i],N);
     printf("\n media seriei %d este %f",i+1,medii[i]);
  }
}
```

În limbajul C++ delimitatorul de sfârşit de şir de caractere este *'\0'*. În limbajul Pascal un şir este structurat printr-un bait ce conţine numărul de poziții urmat de şirul propriu-zis de lungimea indicată.

Conversia fişierului F_1 realizat în C++ la fişierul F_2 realizat în Pascal presupune:

- alocare un bait;
- copierea şirului din F_1 în F_2 cu contorizarea în variabila x a caracterelor;
- la apariția *'\0'* se pune în baitul alocat valoarea contorului x.

Conversia fişierului F_2 Pascal în fişierul F_1 C++ presupune :

- preluarea în variabila x a lungimii şirului;

- copierea unui număr de caractere cât arată contorul *x* din fişierul F_2 în fişierul F_1;
- inserarea în F_1 în continuare a simbolului *'\0'* pentru a marca sfârşitul fişierului.

Programul este:

```
#include <stdio.h>

void main()
{
  char c,x;
  int i=0;
  FILE *pf1,*pf2;
  pf1=fopen("F2.txt","r");
  pf2=fopen("F1.txt","w");
  x=fgetc(pf1);
  while(i<x)
  {
        c=getc(pf1);
        putc(c,pf2);
        i++;
  }
putc('\0',pf2);
fclose(pf1);
fclose(pf2);
}
```