9. REUNIUNILE DE DATE CONTIGUE

9.1 Necesitatea restructurării datelor

În multe aplicaţii codul materialului este privit ca întreg, iar în cazul verificării cifrelor de control, fiecare element sau grupuri de elemente sunt privite ca formate din câmpuri de 1 octet.

În primul şi al doilea caz, zona de memorie asociată codului de material este reprezentată prin două modele grafice şi anume:

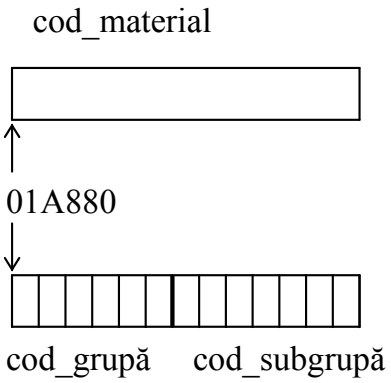

Figura 9.1 *Model grafic structurii de tip reuniune material*

Se observă că ambele structuri ocupă aceeaşi zonă de memorie, al cărui început este marcat de octetul cu adresa exprimată în hexazecimal cu valoarea 01A880.

O altă situaţie corespunde prelucrării articolelor dintr-un fişier. Pentru o aplicaţie sunt necesare primele cinci câmpuri, pentru o altă aplicaţie sunt necesare şapte câmpuri din interiorul articolului, iar pentru a treia aplicaţie sunt necesare ultimele patru câmpuri. Pentru toate aplicaţiile, primul câmp este necesar întrucât serveşte drept câmp de regăsire a informaţiilor.

Modelul grafic al zonei de memorie astfel structurat este:

Figura 9.2 *Structura zonei de memorie*

Cele trei structuri care se suprapun peste aceeaşi zonă de memorie, presupun lungimi identice şi o aceeaşi adresă de început.

În acest caz, rezultă că zona de memorie este mai întâi definită pentru precizarea uneia dintre structuri, iar celelalte structuri care se suprapun apar ca redefiniri ale zonei de memorie.

Atunci când se defineşte un masiv tridimensional *a*, cu 10 x 10 x 10 elemente şi un alt masiv *b*, unidimensional cu 1000 elemente, tipul lor fiind unic, dacă se procedează la punerea în corespondenţă a elementelor *a[1][1][1]* şi *b[1]*, în sensul:

$$adr(a[1][1][1]) = adr(b[1]) \qquad (9.1)$$

secvenţa:

```
for(i=0; i<10; i++)
    for(j=0; j<10; j++)
        for(k=0; k<10; k++)
            a[i][j][k] = 0;
```

este înlocuită cu secvenţa:

```
for(i=0; i<1000; i++)
      b[i]=0;
```

care din punct de vedere a volumului operaţiilor de prelucrare este mai eficientă.

Există probleme în care algoritmii de rezolvare cer definirea de masive care diferă ca nume şi ca dimensiuni de la o etapă la alta, dar care sunt disjuncte din punct de vedere al utilizării conţinutului.

De exemplu, pentru un algoritm alcătuit din trei etape, este necesară utilizarea structurilor menţionate în tabelul de mai jos:

Tabelul nr. 9.1 *Etapele unui algoritm*

Structura / Etapa	A	B	U	V	C	D	X
Etapa 1	*		*		*		
Etapa 2	*			*		*	
Etapa 3		*			*		*

unde structurile sunt definite astfel:

A [10][10]
B [20][20]
U [10]
V [20]
struct C {} , lg (C) = 50
struct D {} , lg (D) = 40
X [30]

Rezultă că matricele *A* şi *B* sunt suprapuse, vectorii *U, V* şi *X* de asemenea, iar structurile de tip articol *C* şi *D*, urmează aceeaşi cale.

Cele trei suprapuneri sunt reprezentate cu modelele grafice următoare:

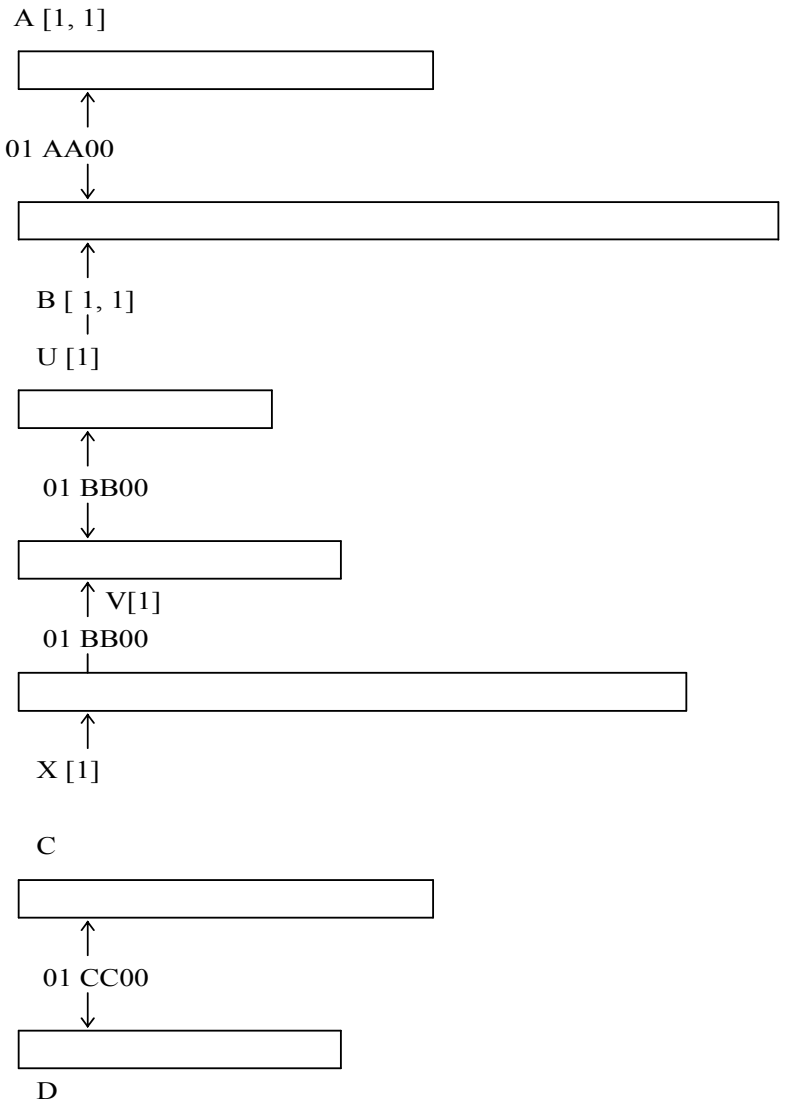

Figura 9.3 *Modelul grafic al suprapunerilor pe o zonă de memorie*

Problema suprapunerilor structurilor de date pe zone de memorie se rezolvă folosind funcţii de reunire.

9.2 Funcţia de reunire

Se consideră datele $d_1, d_2, d_3, ..., d_n$, având tipurile $T_1, T_2, ..., T_n$ şi $m_1, m_2, ..., m_n$ membrii de referinţă ai acestora.

În continuare un membru al unei structuri de date este numit de referinţă, dacă în raport cu el sunt puse în evidenţă. Se defineşte funcţia:

$$union: T_1 * T_2 ... * T_n \rightarrow (TRUE, FALSE) \quad (9.2)$$

şi:

$$union\ (m_1, m_2,...,m_n) = TRUE \quad (9.3)$$

dacă şi numai dacă:

$$adr(m_1) = adr(m_2) = adr(m_n) = \delta, \quad \delta \in [A_i, A_f] \cap N \qquad (9.4)$$

Se notează:

$$\alpha = \min_{1<i<n} \{adr(d_i) \mid adr(m_i) = \delta\} \qquad (9.5)$$

şi:

$$\beta = \max_{1<i<n} \{adr(d_i) + lg(T_i) \mid adr(m_i) = \delta\} \qquad (9.6)$$

Cele *n* structuri de date redistribuite ca dispunere în raport cu adresa *δ*, a celor *n* câmpuri de referinţă, ocupă o zonă de memorie de lungime α - β+1 octeţi.

Diferitele limbaje de programare impun restricţii precum:

$$\alpha = \max\{A_i, \min_{1<i<n}\{adr(d_i) \mid adr(m_i) = \delta\}\} \qquad (9.7)$$

ceea ce determină existenţa unui minim control asupra adresei operanzilor, în sensul că adresele acestora să nu fie în afara domeniului prefixat în limitele A_i şi A_f.

Dacă se consideră o dată de bază, de exemplu d_1:

$$union\ (m_1, m_2, ..., m_n) = TRUE \qquad (9.8)$$

dacă:

$$\begin{cases} adr(mi) = \delta & i = 1, 2, ..., n \\ adr(di) > adr(d1) & i = 2, 3, ..., n \end{cases} \qquad (9.9)$$

De exemplu, acest grup de restricţii se regăseşte în limbajul FORTRAN.

Se consideră masivele definite prin:

```
int a[10];
int b[5];
```

şi structura:

```
struct k
{
      int x;
      char g[10];
```

```
};
k t;
```

şi datele elementare:

```
int x, y;
```

Funcţia:

union (a[3], b[5], g, x, y) *(9.10)*

conduce la modelul grafic de suprapunere următor:

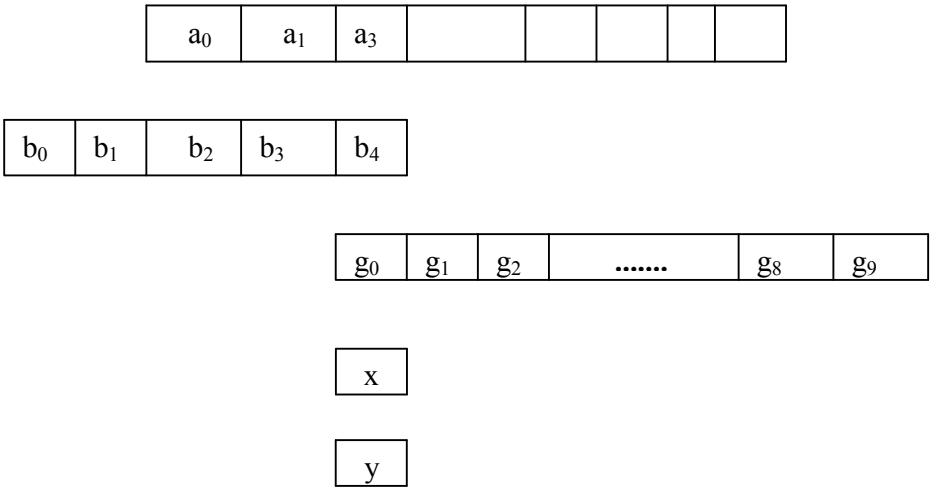

Figura 9.4 *Modelul grafic al suprapunerilor pe o zonă de memorie*

*adr (a [3]) = adr (b [5]) = adr (t*g) = adr (x) = adr (y) = δ* *(9.11)*

Adresa de început pentru fiecare din cele cinci câmpuri se obţine:

*adr (a [3]) = adr (a [0]) + 3*1g (int)*
*adr (a [0]) = δ - 3*1g (int)*
*adr (b [0]) = δ - 5*1g (int)*
*adr (g) = δ - 1*1g (int)*
adr (x) = δ
adr (y) = δ

În programele C/C++, există posibilitatea realizării reuniunii de date folosind variabile de tip pointer care referă aceeaşi zonă de memorie. De exemplu, secvenţa:

```
int x[3][3],*y[3],**z;
for(int i=0;i<3;i++)
      y[i]=x[i];
z=y;

for(i=0;i<3;i++)
      for(int j=0;j<3;j++)
```

```
                x[i][j]=i*3+j;

      cout<<"Adresa de memorie alocata pentru x este:"<<&x<<endl;
      cout<<"Adresa de memorie alocata pentru y este:"<<&y<<endl;
      cout<<"Adresa de memorie alocata pentru z este:"<<&z<<endl;

      cout<<"Adresele de memorie la care sunt stocate liniile pornind
de la variabila x:"<<endl;
      for(i=0;i<3;i++)
            cout<<"Linia "<<i+1<<"este incepe la adresa "<<x[i]<<endl;

      cout<<"Adresele de memorie la care sunt stocate liniile pornind
de la variabila y:"<<endl;
      for(i=0;i<3;i++)
            cout<<"Linia "<<i+1<<"este incepe la adresa "<<y[i]<<endl;

      cout<<"Adresele de memorie la care sunt stocate liniile pornind
de la variabila z:"<<endl;
      for(i=0;i<3;i++)
            cout<<"Linia     "<<i+1<<"este     incepe     la     adresa
"<<*(z+i)<<endl;

      cout<<"Elementele matricei afisate prin variabila x:"<<endl;
      for(i=0;i<3;i++){
            cout<<endl;
            for(int j=0;j<3;j++)
                  cout<<"X["<<i+1<<"]["<<j+1<<"]="<<x[i][j]<<" ";
      }

      cout<<"Elementele matricei afisate prin variabila y:"<<endl;
      for(i=0;i<3;i++){
            cout<<endl;
            for(int j=0;j<3;j++)
                  cout<<"Y["<<i+1<<"]["<<j+1<<"]="<<*(y[i]+j)<<" ";
      }

            cout<<"Elementele   matricei   afisate   prin   variabila
z:"<<endl;
      for(i=0;i<3;i++){
            cout<<endl;
            for(int j=0;j<3;j++)
                  cout<<"Z["<<i+1<<"]["<<j+1<<"]="<<*(*(z+i)+j)<<" ";
      }

      cout<<"Adresele  elementelor  matricei  afisate  prin  variabila
x:"<<endl;
      for(i=0;i<3;i++){
            cout<<endl;
            for(int j=0;j<3;j++)
                  cout<<"ADR(X["<<i+1<<"]["<<j+1<<"])="<<&x[i][j]<<" ";
      }

      cout<<"Adresele  elementelor  matricei  afisate  prin  variabila
y:"<<endl;
      for(i=0;i<3;i++){
            cout<<endl;
            for(int j=0;j<3;j++)
                  cout<<"ADR(Y["<<i+1<<"]["<<j+1<<"])="<<y[i]+j<<" ";
      }

      cout<<"Adresele  elementelor  matricei  afisate  prin  variabila
```

```
z:"<<endl;
      for(i=0;i<3;i++){
            cout<<endl;
            for(int j=0;j<3;j++)
                  cout<<"ADR(Z["<<i+1<<"]["<<j+1<<"])="<<*(z+i)+j<<" ";
      }
```

efectuează punerea în corespondenţă:

$$adr(x[0][0]) = y[0]+0$$
$$adr(x[0][1]) = y[0]+1$$
$$adr(x[0][2]) = y[0]+2$$
$$adr(x[1][0]) = y[1]+0$$
$$adr(x[1][1]) = y[1]+1$$
$$adr(x[1][2]) = y[1]+2$$
$$adr(x[2][0]) = y[2]+0$$
$$adr(x[2][1]) = y[2]+1$$
$$adr(x[2][2]) = y[2]+2$$

între variabilele *x* şi *y*, precum şi punerea în corespondenţă:

$$adr(x[0][0]) = *(z+0)+0$$
$$adr(x[0][1]) = *(z+0)+1$$
$$adr(x[0][2]) = *(z+0)+2$$
$$adr(x[1][0]) = *(z+1)+0$$
$$adr(x[1][1]) = *(z+1)+1$$
$$adr(x[1][2]) = *(z+1)+2$$
$$adr(x[2][0]) = *(z+2)+0$$
$$adr(x[2][1]) = *(z+2)+1$$
$$adr(x[2][2]) = *(z+2)+2$$

între variabilele x şi z.

Astfel, aceeaşi zonă de memorie este accesată prin intermediul variabilelor x, y şi z.

În condiţiile construirii de tipuri de date derivate, nu se efectuează alocare de zone de memorie. Prin definirea de variabile pointer $p_1, p_2, ..., p_n$ spre tipurile de date derivate, o dată cu construirea modelului grafic al reuniunii de date, se evaluează adresa α ca fiind adresa de început a unei variabile d_k, din lista $d_1, d_2, ..., d_n$.

$$\alpha = \mathrm{adr}(d_k) = \min_{1<i<n}\{\mathrm{adr}(d_i)\} \qquad (9.12)$$

cu condiţia ca:

$$adr(m_1) = adr(m_2) = ... = adr(m_n) \qquad (9.13)$$

Se calculează deplasările:

$$D_i = depl(d_i, d_k) \qquad (9.14)$$

cu $D_i > 0$ pentru i = 1, 2, ..., n.

Prin evaluarea expresiei:

$$P_i = convp(\alpha, T_i) + D_i \qquad (9.15)$$

se obţine adresa de început a fiecărei date din tipul T_i, astfel încât:

$$adr(m_1) = adr(m_2) = ... = adr(m_n) = adr(d_1) + depl(m_1, d_1) \qquad (9.16)$$

Mecanismele de implementare prin variabile pointer a reuniunii de structuri de date sunt utile mai ales în cazul structurilor de date contigue a căror alocare se efectuează dinamic.

Chiar dacă, iniţial, funcţiile de reuniune sunt definite cu unele restricţii asupra alinierii, la stânga sau la dreapta a elementelor, printr-o aritmetică adecvată de evaluare a expresiilor în care apar variabile pointer aceste restricţii sunt eliminate.

Reuniunile de date, în multe cazuri, sunt rezultatul unor prelucrări accidentale şi în depanarea programelor e necesară identificarea modulului în care s-au făcut suprapunerile unor date peste altele.

9.3 Zonele de memorie tampon – gazde ale reuniunilor de date contigue

Prelucrarea datelor în condiţiile scăderii intensităţii pe care resursa memorie o impune ca restricţie revine la a defini zone de memorie în care se citesc date din fişiere, uneori chiar fişierele integral, pentru a fi prelucrate ca date existente numai în memoria internă a calculatorului.

În acest context, memoria tampon este pusă în corespondenţă cu structuri de tip articol şi părţi ale sale sunt interpretate sau intră ca operanzi în expresii, sub forma membrilor de structură.

Cazul cel mai frecvent corespunde situaţiei în care, pentru orice structură de date din şirul ordonat $d_1, d_2, ... d_n$ avem:

$$union(m_i, m_j) = FALSE \qquad (9.17)$$

$$adr(d_{i+1}) = adr(d_i) + lg(d_i) \qquad (9.18)$$

$(\forall)\, i = 1, 2, ..., n-1$, adică datele sunt disjuncte.

În cazul lucrului cu buffere, se caută ca prin aritmetica de pointeri să se obţină acea suprapunere care coincide, fie cu modul în care a fost creat fişierul, fie cu obiectivul urmărit.

Dacă programul de creare a fişierului conţine ca definire structura d_k, iar programul de exploatare a fişierului conţin aceeaşi structură d'_k, cu deosebirea că:

$$lg(d_k) \neq lg(d'_k) \qquad (9.19)$$

conduce la concluzia că cele două structuri sunt asemănătoare numai prin numărul identic de câmpuri şi coincidenţa tipurilor, dar diferă prin lungimile a două câmpuri corespondente.

Se presupune că:

$$lg(d_k) - lg(d'_k)=1 \quad (9.20)$$

După *n* citiri de înregistrări d'_k din fişierul creat cu înregistrări având structura d_k, se obţine o diferenţă de *n* baiţi până la a *n + 1* înregistrare corect plasată în fişier.

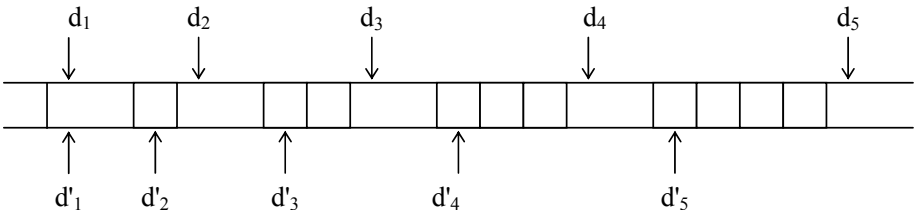

Figura 9.5 *Decalajele între articolele celor două fişiere*

Decalajul de 1 octet generează o suprapunere dinamică, distanţa între începutul înregistrării d_k şi înregistrarea d'_k se măreşte pe măsură ce indicele *k* creşte, fiind de *k-1* octeţi.

Problema depistării cauzelor de obţinere a rezultatelor eronate creşte în complexitate dacă diferenţa:

$$lg(d_k) - lg(d'_k)>1 \quad (9.21)$$

În aceste cazuri, este necesar să se definească funcţii de verificare a concordanţei dintre parametrii ce caracterizează o structură.

De asemenea, din punct de vedere al tipurilor pe care le au datele elementare, care intră în componenţa articolelor sau a masivelor, reunirile apar ca suprapuneri de tipuri, omogene sau nu, cu toate consecinţele ce decurg.

Astfel, o dată elementară sau atom este definită din punct de vedere al tipului ca:

$$E = (T_i) \quad (9.22)$$

unde T_i, este unul din tipurile fundamentale $T_1, T_2,..., T_n$, *n* fiind numărul de tipuri fundamentale implementate în limbajul de programare considerat.

Masivul unidimensional, se defineşte ca tip vector T_v astfel:

$$T_v = (T_i, T_i, ..., T_i) \quad (9.23)$$

Numărul de componente coincide cu dimensiunea vectorului şi tipul este acelaşi pentru toate elementele.

O matrice este de tipul T_m, definit:

$$T_m = (T_v, T_v ..., T_v) \quad (9.24)$$

O structură de tip articol, are tipul:

$$T_a = (T_{i1}, T_{i2}, ..., T_{im}) \quad (9.25)$$

unde, $T_{ik} \in \{T_1, T_2, ..., T_n, T_v, T_m, T_a\}$.

O structură de vectori, este definită prin:

$$T_{sv} = (T_v, T_v ..., T_v) \qquad (9.26)$$

Un vector de structură, se defineşte prin:

$$T_{vs} = (T_a, T_a ..., T_a) \qquad (9.27)$$

Deplasând problematica reuniunii de date la nivelul reuniunii de date ca tipuri, obţinem o nouă interpretare şi anume:

$$\text{union}(T_{x_1}, T_{x_2}, ..., T_{x_n}) = \text{TRUE} \qquad (9.28)$$

dacă există cel puţin un $T_{ki} \in T_{xi}$, astfel încât oricare ar fi două elemente de tip T_{xi} şi T_{xj} să existe:

$$\text{tadr}(T_{ki}) \cap \text{tadr}(T_{kj}) \neq \phi \qquad (9.29)$$

unde, T_{ki} reprezintă tipul membrului cu poziţia k din tipul derivat T_{xi}.

În acest context, $tadr(T_{ki})$ reprezintă o mulţime a adreselor operanzilor de tip T_{ki} ai structurii de tip derivat T_{xi}.

Se consideră spre exemplificare structurile:

```
struct a
{
      char e[20];
      int a;
      float c;
};
```

şi:

```
struct x
{
      int y;
      char z[4];
      float u;
      int w;
};
```

Se consideră structura:

$$T_a = (\ char[], int, float, bool) \qquad (9.30)$$

şi structura:

$$T_x = (\ int, char[], float, int) \qquad (9.31)$$

Structura *union* (T_a, T_x) se evaluează definind mulţimile:

$$\text{tadr}(T_{a\,\text{int}}) = \{\text{adr}(a)\} \qquad (9.32)$$

$$\text{tadr}(T_{b\,\text{int}}) = \{\text{adr}(y), \text{adr}(w)\} \qquad (9.33)$$

unde *tadr()* este funcţia de extragere a adreselor elementelor de un tip specificat dintr-o structură.

$$\text{tadr}(T_{a\,\text{float}}) = \{\text{adr}(c)\} \qquad (9.34)$$

$$\text{tadr}(T_{x\,\text{float}}) = \{\text{adr}(u)\} \qquad (9.35)$$

$$\text{tadr}(T_{a\,\text{char}[\,]}) = \{\text{adr}(e)\} \qquad (9.36)$$

$$\text{tadr}(T_{x\,\text{char}[\,]}) = \{\text{adr}(z)\} \qquad (9.37)$$

$$\text{tadr}(T_{a\,\text{bool}}) = \{\text{adr}(d)\} \qquad (9.38)$$

$$\text{tadr}(T_{x\,\text{bool}}) = \phi \qquad (9.39)$$

Dacă se presupune că programatorul are la dispoziţie posibilitatea de a modifica conţinutul contorului de locaţii, care să realizeze o definire a structurilor *a* şi *b* încât:

$$\text{tadr}(T_{a\,\text{boolean}}) \cap \text{tadr}(T_{x\,\text{real}}) \neq \phi \qquad (9.40)$$

deci:

$$\text{union}(T_a, T_x) = \text{TRUE} \qquad (9.41)$$

Implementările curente generează cazuri particulare în care:

$$\text{adr}(x_i') = \text{adr}(x_j') \qquad (9.42)$$

unde x_i', x_j' sunt membrii cu poziţia 1 din structurile *i* şi *j*.

Membrii care alcătuiesc o structură dintr-o reuniune de structuri, au adresa calculată faţă de primul membru al structurii. Toţi primii membrii ai structurilor, au aceeaşi adresă.

Privind structurile de date ca structuri de tipuri:

$$\text{tadr}(T_{1i}) \cap \text{tadr}(T_{2i}) = \text{adr}(x_j') = \text{adr}(x_i') \qquad (9.43)$$

pentru oricare *i* şi *j*, ce corespund tipurilor de date derivate ce definesc structurile considerate.

O astfel de abordare măreşte generalitatea modelului asociat reuniunii de structuri, întrucât nu mai sunt luate în calcul direct lungimile efective ale operanzilor de un anumit tip, lungimi ce diferă de la un mod de implementare al unui limbaj la alt limbaj.

Mai mult, aplicarea funcţiei *union* la suprapunerile accidentale a structurilor, prin considerarea tipurilor ca entităţi de bază, permite descifrarea rezultatelor care numai aparent au caracter nedeterminat.

Evidenţierea lucrului cu astfel de structuri se realizează prin intermediul următoarelor două exemple.

Primul exemplu iniţializează o zonă sub forma unui masiv unidimensional pe care o utilizează ca masiv bidimensional la afişare.

```
union reuniune
{
      int a[3][3];
      int b[9];
}z;
…
int i, j;
for(i=0; i<9; i++)
     z.b[i] = i * i;
for(i=0; i<3; i++)
{
     for(j=0; j<3; j++)
        cout<<"\n "<<z.a[i][j];
}
…
```

Al doilea exemplu defineşte în cadrul unor structuri de tip *struct*, câmpuri pe care le reuneşte la aceeaşi adresă de memorie, efectuând exploatarea diferenţială a acesteia.

```
struct union1
{
      char x[10];
};
struct union2
{
      char y[5];
};
struct union3
{
      char z[10];
};

union uniune2
{
      union1 un1;
      union2 un2;
      union3 un3;
}u2;
…
strcpy(u2.un1.x,"1234567890");
cout<<u2.un1.x;
cout<<"\n";

for(int i=0; i<5; i++)
      u2.un2.y[i]='a'+i;

for(i=0; i<5; i++)
      cout<<u2.un2.y[i];

cout<<"\n"<<u2.un1.x;

cout<<"\n";
```

```
for(i=0; i<15; i++)
      cout<<u2.un3.z[i];
…
```

Exemplele anterioare evidenţiază modalităţi de realizare şi operarare a reunirilor de date.