2. TIPURI DE DATE ŞI DATE ELEMENTARE

2.1 Date şi informaţii

În practică se face deosebire între o dată şi o informaţie. Exemplele oferite în cele mai multe cazuri sunt edificatoare. Există şi tendinţe de a oferi definiţii pentru date şi pentru informaţii. Dilemele când o informaţie este considerată dată şi când o dată este o informaţie, sunt rezolvate pentru mulţi specialişti, dar rămân dileme pentru o altă categorie de specialişti.

Din punct de vedere al programatorului, ceea ce face obiectul prelucrării sunt de fapt şiruri de biţi care reprezintă date sau informaţii, funcţie de contextul în care sunt generate şi de modul în care se interpretează rezultatele. Pentru a nu complica şi mai mult problematica, se consideră că în activitatea de programare se operează cu date. Toate intrările şi ieşirile programelor sunt date. Sistemele de prelucrare, însă sunt intitulate în continuare sisteme informaţionale sau sisteme informatice, în mod ornamental din punctul de vedere al programatorilor.

În realitate, atunci când acestea funcţionează corect, prelucrează într-adevăr informaţii. Atunci când, însă, fluxurile sunt greoaie şi determină un nivel de istorism costisitor, prelucrările sunt ale unor date certe.

Pentru ca în literatura de specialitate capitolul deţinut descrierii operanzilor – informaţii sau date – se numeşte STRUCTURI DE DATE, în continuare, nu se mai face deosebirea dintre informaţie şi dată. Utilizatorii sunt aceia care decid dacă oferă spre prelucrare informaţii sau date şi dacă rezultatele prelucrării sunt date sau sunt informaţii.

2.2 Clasificări ale datelor

Există numeroase puncte de vedere de abordare a grupării datelor, fiecare constituindu-se într-un criteriu. Ceea ce este însă adevărat, este legat de faptul că fiecărei date i se ataşează totalitatea atributelor ce rezultă din multitudinea de clasificări care se iau în considerare.

a) Criteriul variabilităţii – grupează datele în:
   - date constante, care nu se modifică într-un interval de timp sau pe durata execuţiei programului; în cazul în care pentru a face un program lizibil constantele sunt puse în corespondenţă cu anumiţi identificatori, în programe sunt vehiculaţi aceştia din urmă, formând constantele simbolice.
   - date variabile, ale căror niveluri se modifică fie într-un interval de timp, fie pe parcursul execuţiei unui program; întotdeauna se vorbeşte de o valoare iniţială, valori intermediare şi o valoare finală; numărul valorilor intermediare determină mecanismele necesare prelucrărilor, includerea în structuri repetitive sau stocarea lor în fişiere;

b) Criteriul compunerii – diferenţiază datele astfel:

- date simple sau elementare, fiecare având o anumită semnificaţie şi fiind independente de celelalte date care apar într-un context specificat; datele elementare se mai numesc atomi;
- date compuse sau structurate, formate din date elementare sau date la rândul lor structurate; fiecare componentă are o anumită poziţie în cadrul structurii şi împreună cu celelalte formează un întreg; între părţile care alcătuiesc o dată compusă există legături în primul rând de conţinut şi numai toate la un loc caracterizează un fenomen, un proces sau un individ dintr-o colectivitate: apartenenţa şi poziţia fiecărei componente se precizează explicit la descrierea datei structurate;

c) Criteriul semnificaţiei conţinutului conduce la:
- date care fac obiectul operaţiilor de prelucrare, adică participă ca operanzi în expresii, se iniţializează prin atribuiri sau operaţii de intrare, se stochează pe suporţi, se afişează sau se transmit ca parametri;
- date care permit adresarea operanzilor şi care au valori cuprinse între limite precizate, care prin calcule de adrese localizează corect fie operanzi, fie alte date de adresare, fie funcţii de prelucrare;
- date care efectuează prelucrarea, care apar ca succesiuni de instrucţiuni direct executabile dacă fişierul care le aparţine este încărcat în memoria unui calculator şi se comandă lansarea în execuţie a acestuia;

d) Criteriul naturii datelor – generează tipurile de date următoare:
- date de tip întreg, ale căror elemente aparţin mulţimii Z;
- date de tip real, ale căror elemente aparţin mulţimii R;
- date de tip complex, ale căror elemente aparţin mulţimii C, iar coeficienţii care desemnează partea reală şi partea imaginară aparţin mulţimii R;
- date de tip boolean, ale căror elemente aparţin mulţimii *{TRUE, FALSE}* sau *{0, 1};*
- date de tip caracter, ale căror elemente aparţin mulţimii caracterelor ce sunt definite prin combinaţie de biţi la nivelul unui bait; din cele 256 de combinaţii unele sunt grupate pentru litere, altele pentru cifre, altele pentru caractere speciale şi pentru caractere de control; corespunzător, sunt definite date de tip alfabetic, de tip numeric, date de tip caractere de control etc.; aceste date au câte un singur element din mulţimea care îi defineşte tipul;
- date de tip şir de caractere – reprezintă o compunere prin concatenare a datelor de tip caracter; datele acestea au un delimitator al sfârşitului de şir, fie o constantă de tip întreg la început, precizând numărul de caractere care intră în alcătuirea şirului;

e) Criteriul construirii tipurilor conduce la:
- date de tip fundamental – care aparţin unui tip implementat în fiecare limbaj de programare, precum tipurile întreg, real,

caracter, boolean, complex; programatorul are posibilitatea definirii constantelor simbolice şi variabilelor proprii specificând tipurile fundamentale şi alege prelucrările compatibile acestora;
- date de tip derivat – care se obţin prin includerea în cadrul unor structuri a componentelor având unul din tipurile fundamentale implementate în limbaj; rezultatul obţinut este un tip de dată derivat care se pune în corespondenţă cu un identificator şi care este folosit de programator pentru a defini variabilele în program având respectivul tip;

f) Criteriul dispunerii în memoria internă, grupează datele în:
- date dispuse în zone contigue – care permit localizarea uneia dintre ele cunoscând o adresă şi o deplasare; în cazul în care zonele de memorie ocupate au aceeaşi lungime, adresa fiecărei date se constituie ca termen al unei progresii aritmetice şi este calculată cunoscând adresa primei date şi poziţia în şirul datelor contigue a elementului căutat;
- date dispersate în memoria internă - se obţin în cazul alocării dinamice a memoriei necesare, ceea ce impune stocarea şi conservarea adresei zonei de memorie asociată fiecărei date; dacă datele dispuse în zone contigue, au realizată proiectarea alocării în faza de compilare, datelor dispersate li se alocă memorie efectiv în faza de execuţie şi nu există posibilitatea ca în mod direct să se construiască modele de calcul a adreselor fizice pe care datele le ocupă, mai ales dacă alocarea memoriei este un proces ce depinde de testarea unor condiţii din program;

g) Criteriul câmpului de acţiune, împarte datele în:
- date cu caracter global – care se definesc o singură dată, dar care sunt utilizate din orice punct al programului sau a funcţiilor şi procedurilor care intră în componenţa lui; aceste date se definesc şi li se alocă memorie o singură dată şi au câmpul de acţiune cel mai cuprinzător;
- date cu caracter local – sunt în fiecare procedură şi li se alocă memorie dinamic, automat, la apelarea fiecărei proceduri sau funcţii; odată cu revenirea în secvenţa apelată – deci la ieşirea din funcţie sau din procedură - are loc eliberarea memoriei alocate (dealocarea memoriei); variabilele locale nu sunt folosite decât în procedura sau funcţia unde au fost definite;
- date de tip registru – au rolul de a pune la dispoziţie programatorului în limbaje evoluate, accesul la registrele calculatorului; în cazul unei folosiri judicioase există posibilitatea creşterii vitezei de prelucrare, iar în cazul folosirii abuzive a registrelor se obţine fenomenul invers;

h) Criteriul definirii domeniului presupune:
- date al căror domeniu este specificat prin limita inferioară, limita superioară şi forma de prezentare generică a elementelor;
- date al căror domeniu este definit odată cu enumerare elementelor care îi formează.

i) Criteriul alocării memoriei, grupează datele în:

- date statice – calcule de alocare a memoriei se efectuează în faza de compilare, iar înainte de execuţie, alocarea este efectivă;
- date dinamice a căror memorie este alocată şi dealocată în timpul execuţiei programului, prin funcţii de bibliotecă apelate.

Într-un program, o anumită dată este astfel definită încât se încadrează într-una din subgrupele fiecărui criteriu. Astfel, definirea:

```
// PROGRAM definire:
#include<…….>
#include<…….>
.....................................
int  k;
main( )
{
      .....................................
}
```

dintr-un program C/C++ se interpretează astfel:

- *k* este o dată variabilă (criteriul variabilităţii);
- *k* este o dată elementară (criteriul compunerii);
- *k* este o dată de tip operand (criteriul semnificaţiei);
- *k* este o dată de tip întreg (criteriul naturii datelor);
- *k* este o dată de tip fundamental (criteriul construirii tipurilor);
- *k* este o dată dispusă într-o zonă contiguă (criteriul dispunerii în memoria internă);
- *k* este o dată globală (criteriul câmpului de acţiune).

Deci, *k* este un operand, variabila elementară, globală, de tipul fundamental întreg, dispus într-o zonă contiguă.

2.3 Modele de prezentare a datelor

Între forma de reprezentare naturală sau externă a datelor şi forma de reprezentare internă a acestora, există mari diferenţe.

Reprezentarea internă a datelor, se realizează utilizând algoritmi de codificare, care pun în corespondenţă datele cu şiruri de biţi. Pentru fiecare tip de dată se defineşte lungimea zonei de memorie şi algoritmul de codificare, precum şi codurile operaţiilor care utilizează operanzii în concordanţă cu caracteristicile de tip ale acestora.

a) Modelul de verificare a concordanţei tip-conţinut-adresă

Modele de reprezentare a datelor au în componenţa lor:

- $LSUP_i$,$LIMF_i$ – valori care precizează limita inferioară şi limita superioară a intervalului căreia îi aparţine data de tip i specificat;
- A_i – indicatorul algoritmului de realizare a reprezentării interne pentru datele de tip *i*;
- *f( )* – funcţia de apartenenţă a datei la un anumit tip;

- *M* – mulţimea funcţiilor de conversie;
- *n* – numărul de tipuri de date;
- *k* – cerinţe de aliniere a adresei de început a zonei de memorie, k $\in$ {1, 2, 4, 8};
- T_i – natura *i* a datei;
- *adr( )* – funcţia de determinare a adresei unei zone de memorie pusă în corespondenţă cu un identificator specificat:

$$adr : J \rightarrow N \qquad (2.1)$$

unde:

- *J* – mulţimea identificatorilor;
- *N* – submulţime a numerelor naturale cu care se localizează fiecare bait al zonei de memorie la dispoziţia programatorului;

- $\widetilde{N} = [a,b] \cap N$ – delimitează posibilităţi hardware de dispunere în memorie a programului, unde a, b $\in$ N, a < b;
- *cont( )* – funcţia conţinut a zonei de memorie:

$$cont : J \times T_i \rightarrow \overset{n}{\underset{i=1}{U}} D_i \qquad (2.2)$$

- *tip( )* – funcţia de identificare a tipului variabilei:

$$tip : J \rightarrow T \qquad (2.3)$$

Mulţimea T_j a naturii datelor fundamentale implementate în limbajul de programare L_j, se defineşte prin:

$$T_j = \{ T_1, T_2, ..., T_n \} \qquad (2.4)$$

Dacă *j* este *C*, atunci:

$$T_C = \{int, bool, float, char, string\} \qquad (2.5)$$

deci n = 5, fără a fi luate în considerare variantele pentru datele întregi, reale şi posibilitatea de a specifica seturi de valori.

Pentru datele de natură boolean, $LSUP_2$ este 1 sau TRUE şi $LINF_2$ este 0 sau FALSE.

Pentru datele de natură real A_3, corespunde modului de construire a mantisei şi caracteristicii precum şi dispunerea acestora pe cei 6 baiţi.

Pentru datele întregi *H*, mulţimea funcţiilor de conversie are elementele:

$$H = \{f_{11}, f_{12}, f_{13}, f_{14}, f_{15}\} \qquad (2.6)$$

Dintre acestea numai f_{11} şi f_{14} sunt inversabile, deci tabloul funcţiilor:

$$f_{ij} ( ), \; i = 1, 2, 3, 4, 5 \text{ şi } j = 1, 2, 3, 4, 5 \qquad (2.7)$$

demonstrează că se efectuează conversii în toate direcţiile cu o anumită pierdere a unor simboluri din descrierea iniţială.

Cerinţele de aliniere sunt specifice particularităţii hardware a sistemelor de calcul. În cazul în care la compilare nu este realizată optimizarea alocării memoriei, apar zone neutilizabile cu efecte ce sunt interpretate mai dificil.

Declararea:

```
...
char a;
float b;
char c;
char d;
...
```

în absenţa optimizării alocării de memorie, conduce la rezervarea:

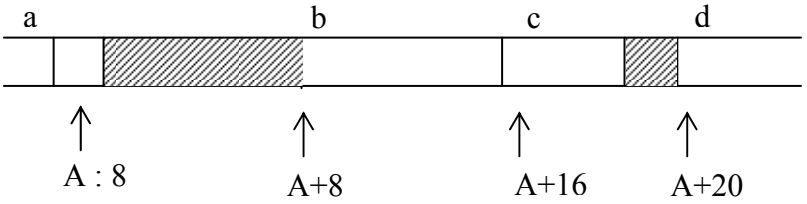


Figura 2.1 *Alocarea în memorie a variabilelor a, b, c şi d*

În cazul optimizării, secvenţei de program îi corespunde:

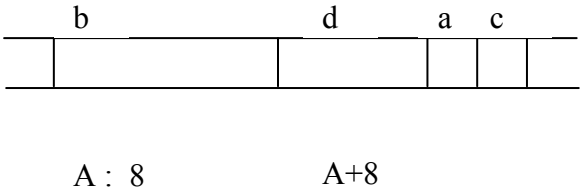


Figura 2.2 *Alocarea optimizată în memorie a variabilelor a, b, c şi d*

Este posibilă optimizarea datorită comunicativităţii dispunerii operanzilor într-o secvenţă, atunci când aceştia sunt elementari şi nu apare problema redefinirii.

Funcţia de apartenenţa a datelor la un anumit tip se defineşte prin:

$$f : U \times D_i \rightarrow \{ FALSE, TRUE\} \quad (2.8)$$

unde:

- U - mulţimea şirurilor ce se generează cu simbolurile alfabetului nstruit pentru un limbaj;
- D_i – intervalul sau mulţimea elementelor specifice tipului de date i.

$$f(x,i)=\begin{cases} FALSE & daca\ x \notin Di \\ TRUE & daca\ x \in Di \end{cases} \qquad (2.9)$$

De exemplu:

$$f(13, int) = TRUE \quad (2.10)$$

pentru că 13 $\in$ [-32768, 32767] $\cap$ Z, D_{int} fiind domeniul întregilor, în timp ce

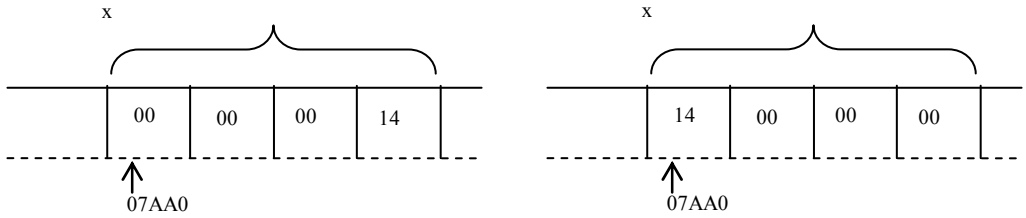
$$f(-4, bool) = FALSE \quad (2.11)$$

pentru că –4 nu aparţine mulţimii {FALSE, TRUE}.

În secvenţa:

```
...
int x;
...
x = 20;
...
```

presupunând că în compilare şi editare de legături, variabilei x i se asociază zona de memorie:



a b

Figura 2.3 *Zona de memorie asociată variabilei x*
(a – microprocesoare MOTOROLA, b – microprocesoare INTEL)

adr(x) = 07AA0
cont(x) = $(00000014)_{16}$
f(cont(x), int) = TRUE
tip (x) = int

Deci:

$$f(cont(x), tip(x)) = TRUE \quad (2.12)$$

Funcţia de aliniere:

$$K: adresa\ x\ tip_i \rightarrow TRUE \quad (2.13)$$

$$K(\text{adresa}, T_i) = \begin{cases} \text{TRUE} & \text{daca adresa } \mathrm{M}\, k_i \\ \text{FALSE} & \text{daca adresa} \div k_i \end{cases} \qquad (2.14)$$

unde k_i este factorul de aliniere cerut prin construcţie pentru tipul de date T_i .

$$K(07AA0, int) = TRUE \qquad (2.15)$$

pentru că 07AA0M4

$$K(adr(x), tip\ (x)) = TRUE \qquad (2.16)$$

Se spune că variabila x este:
- corect alocată;
- corect iniţializată;

dacă şi numai dacă:

$$f\ (cont(x), tip(x))\ AND\ K(adr(x), tip(x)) = TRUE \qquad (2.17)$$

b) Modelul de generare a constantelor pentru un tip specificat de date

Folosind convenţii şi simboluri, se definesc reguli, mecanisme de generare a constantelor. Astfel, se notează:

{ c }	- cifra
{ a }	- a sau b
{ b }	
[ ]	- construcție opțională

Figura 2.4 *Notaţiile utilizate în generarea constantelor*

Constantelor întregi li se asociază modelul de generare:

$$\left(\left\{ \begin{matrix} + \\ - \end{matrix} \right\} \right) \quad \text{ccccc. . . c}$$

Figura 2.5 *Modelul de generare a constantelor*

Putem verifica dacă şirurile:

0
- 125
. 3

+- 60
44 –

sunt sau nu constante întregi. Cu uşurinţă ne dăm seama că şirurile . 3 + - 60 şi 44 - nu îndeplinesc cerinţele impuse de şablonul model.

Pentru constantele de tip real se prezintă modelul de generare:

$$\left[\begin{Bmatrix}+\\-\end{Bmatrix}\right]\ [cc..c]\ \ [.]\ \ [cc..c]\ \ \left[\begin{Bmatrix}E\\e\end{Bmatrix}\right]\ \ \left[\begin{Bmatrix}+\\-\end{Bmatrix}\right]\ \ [cc]$$

Figura 2.6 *Modelul de generare a constantelor reale*

Şirurile:

+ 1 . 2e-4
- . 3 E2
2 . e-1
1e + 4

sunt constante reale întrucât respectă regulile de generare incluse în şablon.

c) Modele de descriere a datelor folosind grafuri.

Întrucât grafurile permit punerea în evidenţă a interdependenţelor dintre elemente omogene sau neomogene, se consideră utilizarea lor ca fiind sugestivă în cazul structurilor de date.

Prin convenţie, se stabilesc că nodurile care prezintă numai arce incidente spre interior corespund datelor elementare, iar nodurile care au arce incidente spre interior şi spre exterior corespund datelor de grup.

Astfel, graful:

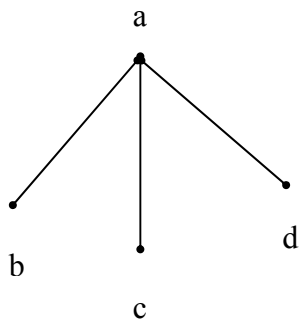


Figura 2.7 *Graful de reprezentare a datei compuse a*

este interpretat ca: data compusă *a* are în alcătuirea ei datele elementare *b*, *c* şi *d*.

Graful:

0 - ->0 - ->0- ->0
a b c d (2.18)

corespunde datelor interdependente, în care *b* urmează lui *a*, *c* urmează lui *b* şi *d* urmează lui *c*. Nodurile *a*, *b*, *c*, d sunt fie date elementare, fie date compuse, iar pentru stabilirea relaţiei de precedenţă este necesară memorarea unor adrese.

Pentru realizarea în cadrul programelor a definirii structurilor complexe de date este necesară reprezentarea acestora folosind grafuri şi după aceea scrierea în program a unei forme liniarizate neambigue.

De exemplu, pentru structura de tip arborescent, se utilizează scrierea parantetica, ce presupune ca elementele de pe acelaşi nivel să fie separate prin virgulă, iar pentru trecerea la nivel inferior, utilizarea unei paranteze rotunde deschise.

Revenirea la nivelul precedent se marchează cu o paranteză închisă. Astfel grafului:

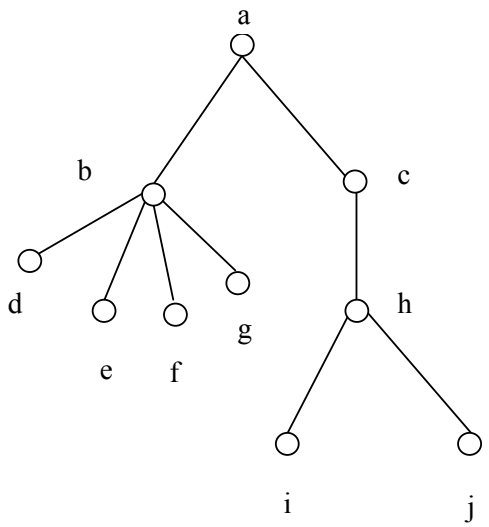


Figura 2.8 *Graful de reprezentare a datei complexe a*

îi corespunde liniarizarea:

$$a(b(d,e,f,g), c(h(i,j))) \qquad (2.19)$$

d) Modele care permit implementarea recursivităţii în descrierea datelor

Se face deosebire între modelele recursive de descriere a datelor precum:

<semn> : : = + | -
<cifra> : : = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<întreg fără semn> : : = <cifra> | <întreg fără semn><cifra> |
<cifra><întreg fără semn>
<întreg> : : = <întreg fără semn> | <semn><întreg fără semn>

şi modelele care implementează, recursivitatea în descrierea datelor. Astfel, construcţia:

$$tip\_de\_dată\ \alpha = \beta\ (\gamma\_întreg, \delta\_\alpha) \qquad (2.20)$$

pune în evidenţă ca dată β conţine două date elementare şi anume γ care are tipul întreg şi δ care are tipul α.

Aceste modele permit descrierea structurilor de date autoreferite: liste, stive şi arbori.

e) Modele grafice

Sunt utilizate reprezentări grafice pentru locaţiile de memorie asociate variabilelor şi constantelor unui program. Prin arce se stabilesc legăturile dintre locaţii. Acest model de descriere a datelor este sugestiv şi fără ambiguitate.

Construcţia:

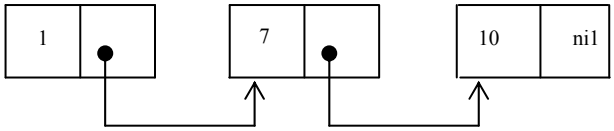


Figura 2.9 *Modelul locaţiilor de memorie asociate variabilelor şi constantelor*

reprezintă o listă, fiecare componentă având două elemente: primul reprezintă informaţia utilă având valorile 1, 7 şi 10, iar al doilea conţine adrese. Arcele orientate indică locaţia a cărei adresa este memorată în componenta precedentă.

f) Modelul vectorial

Se consideră un vector având un număr dat de componente. Fiecare componentă are o semnificaţie precizată, iar componentele luate în ansamblu lor descriu complet şi corect structurile de date.

Se observă că pentru datele elementare, multe dintre componentele vectorului sunt nule. Zerourile, arată lipsa dependenţelor în aval şi în amonte sau mărimea distanţei dintre două componente.

Funcţia distantă se defineşte astfel:

$$d(x,y) = adr\ (y) - adr\ (x) \qquad (2.21)$$

Definim $lg(x,T_i)$, funcţia lungime a zonei de memorie asociată operandului x. De exemplu, pentru secvenţa de program:

```
...
x : extended;
y : comp;
z : shortint;
w : word;
...
```

lg(x, extended) = 10 baiţi
lg (y, comp) = 8 baiţi
lg (z, shortint) = 1 baiţi
lg(w, word) = 4 baiţi
unde:

$$lg : I \times T \to \{1,2,4,6,8,10\} \quad (2.22)$$

Programul care pune în evidenţă lungimile tipurilor de date ale limbajului C/C++ este:

```
//program dimensiune_tip
#include <iostream.h>
#include <malloc.h>
main()
{
cout<<"\n - - - - - - - - - - - - - - - - - - - - - - - - -  - ";
cout<<"\n Reprezentarea datelor de tip întreg";
cout<<"\n - - - - - - - - - - - - - - - - - - - - - - - - -  - ";
cout<<"\n char: "<< sizeof(char)<<" octet";
cout<<"\n unsigned char: "<< sizeof(unsigned char)<<" octet";
cout<<"\n int: "<< sizeof(int)<<" octet";
cout<<"\n unsigned int: "<< sizeof(unsigned int)<<" octet";
cout<<"\n signed int: "<< sizeof(signed int)<<" octeti";
cout<<"\n short int: "<<sizeof(short int)<<" octeti";
cout<<"\n long int: "<< sizeof(long int)<<" octeti";
cout<<"\n - - - - - - - - - - - - - - - - - - - - - - - - -  - ";
cout<<"\n Reprezentarea datelor de tip real";
cout<<"\n - - - - - - - - - - - - - - - - - - - - - - - - -  - ";
cout<<"\n float: "<< sizeof(float)<<" octeti";
cout<<"\n double: "<<sizeof(double)<<" octeti";
cout<<"\n long double: "<<sizeof(long double)<<" octeti";
cout<<"\n - - - - - - - - - - - - - - - - - - - - - - - - -  - ";
cout<<"\n Reprezentarea datelor de tip caracter";
cout<<"\n - - - - - - - - - - - - - - - - - - - - - - - - -  - ";
cout<<"\n char: "<<sizeof(char)<<" octet";
cout<<"\n - - - - - - - - - - - - - - - - - - - - - - - - -  - ";
cout<<"\n Reprezentarea datelor de tip logic";
cout<<"\n - - - - - - - - - - - - - - - - - - - - - - - - -  - ";
cout<<"\n boolean: "<< sizeof(bool)<<" octet";
}
```

Dacă variabilele *x* şi *y* sunt contigue:

$$d(x,y) = lg\ (x,T_i) \quad (2.23)$$

În cazul vectorilor şi matricelor, apare posibilitatea punerii în evidenţă a contiguităţii. Pentru un vector *x* cu *n* componente:

$$d(x[j],\ x\ [j+1]) = 1g\ (x[1],T_i\ ) \qquad (2.24)$$

oricare *j* aparţine mulţimii {1, 2, . . ., n}.

În cazul în care variabilele ocupă zone de memorie necontigue:

$$d(x,y) > lg\ (x,T_i) \qquad (2.25)$$

inegalitatea depinzând de modalitatea în care s-a făcut alocarea memoriei.

Redefinirile sau reacoperirile, corespund unor distanţe fie nule, fie mai mici decât lungimea câmpului considerat reper.

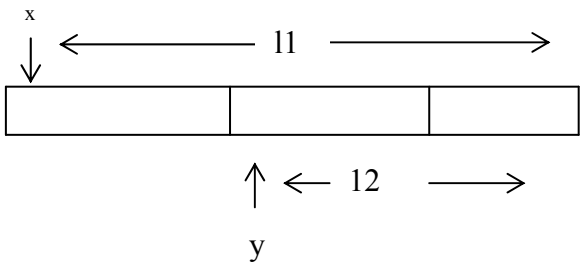


Figura 2.10 *Modelul de reacoperire a locaţiilor de memorie*

$$d(x,y) < lg\ (x) = l1 \qquad (2.26)$$

$$lg\ (y) = 12 \qquad (2.27)$$

$$adr(x) < adr(y) < adr(x) + 11 \qquad (2.28)$$

Uniunile de date apar drept cazuri particulare în acest model de descriere a datelor.

$$union\ a: T_1\ ,\ b:T_2\ ,c\ :T_3; \qquad (2.29)$$

$$dist(a,b) = dist(a,c) = dist(b,c) = 0 \qquad (2.30)$$

Lungimea zonei de memorie ocupată de variabilele "union" este:

$$1= max\ \{lg(a,T_1),lg(b,T_2),lg\ (c,T_3)) \qquad (2.31)$$

g) Modelul obiectelor generice

Datele aparţinând unui anumit tip apar în expresii precedate sau urmate de anumiţi operatori. De asemenea, ele sunt parametrii pentru anumite funcţii.

Modelul include:

- forma generică de construire a datei de un tip specificat
- operatorii şi funcţiile care utilizează datele definite pentru tipul respectiv, precum şi excepţiile de utilizare.

Acest model permite descrierea corectă a secvenţelor de program, cu construirea de obiecte acolo unde limbajele de programare implementează cerinţele programării orientate pe obiect.

Există multe alte modalităţi de descriere riguroasă a datelor, toate însă se subordonează unor obiecte dintre care cel mai important este crearea premiselor analizei semantice, pentru punerea în evidenţă mai întâi a corectitudinii descrierilor de date şi mai apoi a corectitudinii programelor.

2.4 Cerinţe de definire a datelor

În toate programele scrise apar variabile simple. Acestea definesc fie variabile de control, fie variabile în care se regăsesc totaluri sau rezultate independente de alte evaluări ale programului. Fiecare dată pe care un programator o specifică are caracteristici proprii, care determină tipul şi locul în care este definită, modul de alocare a memoriei, modalitatea de iniţializare şi felul în care este folosit în final conţinutul său.

Vom considera spre exemplificare un program P în care se utilizează variabilele *i* de tip întreg şi *s* de tip real.

Variabila *i* este o variabilă de control folosită în regăsirea elementelor vectorului definit prin:

```
float x[100];
```

Variabila *s* este definită astfel:

float s;

Pentru variabilele independente, definirea în secvenţă este comutativă fără a influenţa rezultatul prelucrării.

Secvenţele:

$$S1.\begin{cases}\text{variabile}\\ \quad \text{int i;}\\ \quad \text{float s;}\end{cases} \qquad S2.\begin{cases}\text{variabile}\\ \quad \text{float s;}\\ \quad \text{int i;}\end{cases} \qquad (2.32)$$

sunt echivalente, rezultatele prelucrării unui program care conţine una din cele două secvenţe sunt identice.

Dacă programului P i se ataşează o secvenţă S la stânga, se obţine programul P_1:

$$P_1 = S \,||\, P \qquad (2.33)$$

unde || este operatorul de concatenare.

Dacă programului P i se ataşează o secvenţă S la dreapta, se obţine programul P_2:

$$P_2 = P \,||\, S \qquad (2.34)$$

Se spune că secvenţa S de instrucţiuni este comutativă în raport cu operatorul || dacă:

$$rez\,(S \,||\, P;\, d) = rez\,(P \,||\, S;\, d) \qquad (2.35)$$

unde *rez( )* este funcţia rezultat definită:

$$rez: P \; X \; C \rightarrow C \qquad (2.36)$$

unde:
P – mulţimea programelor;
C – mulţimea constantelor elementare, vectoriale, matricele şi de alte structuri.

În cazul secvenţelor S_1, S_2 dacă:

$$rez(S_1 \;||\; P;d) = rez(S_2 \;||\; P;\; d) \qquad (2.37)$$

se spune că cele două secvenţe sunt echivalente, adică:

$$(S_1 \;\; U \;\; S_2) \;\; || \;\; P \qquad (2.38)$$

Se observă că în programul P, variabila de control *i* trebuie iniţializată cu valoarea 1 şi atinge cel mult valoarea 100, domeniul acesteia fiind:

$$Dom\ (i) = [1,100] \cap N \qquad (2.39)$$

Definirea cu tipul *integer* determină:

$$Dom(int) = [-37768,\ 32767] \cap N \qquad (2.40)$$

deci $Dom\ (i) \subset Dm\ (int)$ unde *Dom* este funcţia domeniu:

$$Dom : J \rightarrow D \qquad (2.41)$$

Această funcţie permite tratarea tipurilor fundamentale *int, float, bool* ca identificatori cu caracteristici prefixate prin construcţia limbajului C/C++.

Dacă presupunem că cele 100 de componente ale vectorului *x* au valori cuprinse între 1 şi 200 suma lor nu depăşeşte 200 * 100 = 20000. Deci:

$$Dom\ (s) = [100,20000] \cap N \qquad (2.42)$$

$$Dom\ (float) = [2.9E - 39,1.7E38] \qquad (2.43)$$

$$Dom\ (s) \subset Dom\ (float) \qquad (2.44)$$

Faptul că domeniile variabilelor *i* şi *s*, sunt incluse în domeniile definite tipurilor, determină excluderea situaţiei obţinerii de rezultate trunchiate şi deci de pierdere a controlului conţinutului lor.

Oportunitatea alegerii tipului întreg sau real, este pusă în evidenţă de ponderea funcţiilor de conversie care se activează la execuţie.

Pentru un programator care a lucrat într-un limbaj de asamblare, secvenţele:

```
int i;                          int i;
float s;                        float s;
. . .                           . . .
i = 1.;                         i  = 1;
s = 0;                          s = 0.;
```

sunt diferite pentru că:

- în prima secvenţă se generează constanta 1, ca având tip *float*; deci ocupă o zonă de memorie de 4 bytes (constante de maxim 7 - 8 cifre). Zona este structurată pentru caracteristică şi mantisă. Constanta 0 se generează într-o zonă de memorie de 2 sau 4 bytes corespunzătoare tipului *int*;
- în modul obiect generat la compilare, *i = 1* şi *s = 0*, se concretizează prin copieri (mutări) ale conţinutului zonelor de memorie care corespund operanzilor din dreapta semnului egal, în alte zone de memorie care corespund operanzilor din stânga semnului egal;
- instrucţiunile de copiere (mutare) presupun operanzi omogeni; dacă operandul receptor este de tip întreg, atunci operandul emiţător trebuie să fie tot de tip întreg; dacă operandul receptor este de tip real, atunci şi operandul emiţător trebuie să fie de tip real; în caz de neomogenitate, compilatorul generează secvenţe de apelare a funcţiilor de conversie;
- secvenţa S_1 necesită 2 apeluri de funcţii de conversie şi anume: conversie de la real la întreg, f13 şi conversie de la real la întreg f31;

rez1 = f13(1.)
copiere rez1 -> i
rez2 = f31 (0)
copiere rez2 -> s

- întrucât în secvenţa S_2, există concordanţa între tipurile constantelor generate ca operanzi emiţători şi operanzi receptori, nu mai sunt necesare apelări ale funcţiilor de conversie.

Ca o cerinţă în alegerea tipului, este realizarea unui nivel cât mai redus al apelurilor funcţiilor de conversie.

Posibilitatea definirii la utilizare a variabilelor elementare, conduce uneori la realizarea unei ocupări a memoriei cu operanzi cu grad redus de folosire.

Există limbaje, ca de exemplu Fortran şi Basic, care nu necesită definirea explicită a variabilelor, ci acestea se definesc la prima utilizare, tipul fiind precizat odată cu respectarea unei reguli de construire a identificatorilor.

Gradul de utilizare este marcat prin numărul de instrucţiuni în care variabilele apar, sau prin frecvenţa de modificare a conţinutului lor. Un program devine cu atât mai bun cu cât împrăştierea variabilelor elementare este mai redusă.

Astfel, dacă în secvenţa S_1 apare variabila *i*, iar în secvenţa S_2 apare variabila *j* şi programul:

$$P = S_1 \,||\, S_2 \qquad (2.45)$$

se observă că domeniile celor două variabile care au acelaşi tip sunt disjuncte, deci este definită o singură variabilă care este utilizată de ambele secvenţe.

Dacă:

$$rez\,(S_2;\, rez\,(S_1;\, i),\, j) = rez\,(S_2;\, rez(S_1;i),\, i) \qquad (2.46)$$

secvenţa:

```
int i;
int j;
...
S1 (i);
...
S2 (j);
```

va fi modificată obţinându-se secvenţa:

```
int i;
...
S1 (i);
...
S2 (i);
```

Caracterul local sau global, dinamic sau static, este dat de contextul în care se utilizează fiecare variabilă. Important este ca programul să realizeze pentru un exemplu de test din specificaţiile de programare, acelaşi conţinut pentru toate punctele de control.

Dacă pentru valorile 1, 2, 3, 4, 5 ale vectorului *x* de 5 componente, la iteraţia a treia, în specificaţiile de programare se indică pentru variabila s valoarea 6 şi dacă prin:

$$rez\,(S_1\, S_1\, S_1;\, s,\, i) \qquad (2.47)$$

cont(s) este 6, înseamnă că definirea *s* este corectă. Dacă însă în locul asociat structurii repetitive, este definit s şi este iniţializat:

$$rez\,(S_1\;\; S_1\;\; S_1;\, s,\, i) \qquad (2.48)$$

conduce la *cont (s)* cu valoarea 3, pentru că celelalte valori se pierd la fiecare activare a blocului, se conchide că definirea locală determină erori asupra rezultatului.

2.5 Cerinţe de iniţializare şi utilizare

Variabilele elementare se iniţializează sub control de către programator. Ele au semnificaţii precum:

- definesc dimensiunile problemei de rezolvat;
- definesc precizia rezultatelor;
- definesc opţiuni ale utilizatorului care determină funcţii care se activează;
- conţin rezultate cu grad de cuprindere diferenţiat;
- controlează execuţia repetitivă a secvenţelor;
- specifică limite de valori pe care le iau unele variabile;
- conţin niveluri puse în corespondenţă cu tipuri de erori, tipuri de rezultate sau evenimente în prelucrare.

Compilatoarele moderne pun în evidenţă situaţiile în care se definesc şi se utilizează variabile elementare, fără ca în prealabil să fie iniţializate. Iniţializarea unei variabile elementare se efectuează:

- la definire; există limbaje care permit definirea şi iniţializarea variabilei (de exemplu, în limbajul C, construcţiile *int s = 0, i = 0;* sunt frecvente);
- printr-o funcţie de citire;
- prin atribuire, variabila elementară aflându-se în membrul stâng.

O variabilă elementară se defineşte pentru a i se utiliza conţinutul cel puţin o singură dată într-o expresie, ca parametru într-o funcţie sau într-o expresie indicială.

Afişarea rezultatului conţinut de o variabilă elementară apare ca o utilizare a acesteia sub formă de parametru în funcţii precum *writeln( ), println().*

Se spune că variabila elementară *i* este corect definită şi corect utilizată dacă:

$$cont\_spcf\ (i,n) = = cont\_prg\ (i,m) \qquad (2.49)$$

unde:

cont_spcf ( ) – funcţia de conţinut a variabilei *i* după efectuarea pasului *n* al algoritmului precizat în specificaţiile de programare;

cont_prg ( ) – funcţia de conţinut a variabilei *i* după executarea instrucţiunii *m* din program, instrucţiune care delimitează sfârşitul pasului *n* al algoritmului descris în specificaţii.

Dacă:

$$cont\_prg\ (i, m) = = cont\_prg(i, m+1) \qquad (2.50)$$

oricare ar fi $m \varepsilon [1, M] \cap N$, unde M este numărul de instrucţiuni executabile care formează programul, se spune că *i* nu îşi modifică conţinutul, este deci o constantă şi ori este defectuos utilizată, ori trebuia definită nu ca variabilă ci ca o constantă simbolică.

Urma programului se obţine prin:

```
cout<<m<<" "<<cont_prg(i, m);
```

unde m = 1, 2, . . . , M sau numai pentru valori modificate:

```
if (cont_prg (i,m) != cont_prg(i,m+1))
      cout<<M+1<<" "<<cont_prg(i,m+1);
```

În cazul în care datele elementare se definesc numai pentru seturi de valori dintr-o mulţime, se utilizează funcţia de apartenenţă, care pune în evidenţă corectitudinea reiniţializării unei variabile sau a rezultatelor din calcule.

Dacă:

$$f\ (cont\_prg\ (x, m), T_i) = = FALSE \qquad (2.51)$$

înseamnă că la instrucţiunea a *m* – a a programului, valoarea variabilei elementare *x* nu corespunde setului de valori T ataşat acesteia.

De exemplu, dacă pentru marcarea erorilor de execuţie se atribuie codurile:

- 0, dacă execuţia s-a desfăşurat normal;
- 1, dacă există tentativa împărţirii prin zero;
- 2, dacă matricea este singulară;
- 3, dacă valorile unei expresii indiciale depăşesc limitele pentru care este definit operandul de tip masiv:

$$T_i = \{0, 1, 2, 3\} \qquad (2.52)$$

Dacă într-o funcţie de inversare a matricei, variabila *ierr* este definită pe mulţimea T_i şi dacă într-un punct *k* al funcţiei i se atribuie valoarea 7, atunci:

$$f\ (cont\_prg\ (ierr, k), T_i) = = FALSE \qquad (2.53)$$

înseamnă că s-a înregistrat o eroare, o îndepărtare de specificaţiile de programare. Este posibil ca uneori specificaţiile de programare să suporte modificări, care reflectă cerinţe de fineţe realizate în program. Mulţimea tipurilor de erori este diversificată şi atunci se obţine:

$$T'_i = T_i\ U\ \{4, 5, 6, 7\} \qquad (2.54)$$

unde codurile 4, 5, 6, 7 corespund unor noi situaţii care conduc la întreruperea execuţiei. În acest caz:

$$f(cont\_prg(ierr, k), T'_i) = = TRUE \qquad (2.55)$$

Cerinţele de iniţializare pentru date de acelaşi tip, conduc la trecerea de la variabile elementare, la variabile de tip masiv.

Construcţiei:

```
int a, b, c, d, e, f;
...
a = 0;
b = 0;
c = 0;
d = 0;
e = 0;
f = 0;
...
```

îi corespunde secvenţa compactă:

```
int x[6];
int i;
. . .
for (i=0; i<6; i++)
       x[i] = 0;
```

sau secvenţa:

```
int x[6] = {0, 0, 0, 0, 0, 0};
```

Alegerea dintre date elementare şi date compuse, este legată în primul rând de modul de calcul a adreselor şi de obiectivul urmărit prin prelucrare iar în al doilea rând, de compactitatea programului.

Datele elementare, se constituie ca o mulţime de noduri ale unui graf în care mulţimea arcelor este vidă. Adresele variabilelor elementare au caracter aleator. În general, nu se stabileşte o relaţie de calcul a adreselor unor elemente din mulţimea de date elementare, având ca reper un element aparţinând de asemenea acestei mulţimi.

2.6 Cerinţe de lizibilitate a programului

Datele elementare sunt puse în corespondenţă cu identificatori sugestivi. Astfel, pentru calculul volumului unei prisme paralelipipedice se definesc:

```
. . .
int lungime, latime, inaltime, volum;
cin>>lungime>>inaltime>>latime;
volum=lungime*latime*inaltime;
cout<<"volum = "<< volum;
. . .
```

În cazul definirii unei variabile compuse omogene, secvenţa echivalentă este:

```
. . .
int x[4];
cin>>x[0]>>x[1]>>x[2];
x[3]=x[0]*x[1]*x[2];
cout<<" volum = "<<x[3];
. . .
```

Lizibilitatea programului în acest caz, este crescută numai ca posibilitate de urmărire a sintaxei acestuia. În prima formă se înţelege exact semnificaţia prelucrării. În plus, dacă se acceptă utilizarea variabilelor de stare globale, care sunt în totalitate variabile elementare, după apelarea funcţiilor, se fac teste şi se continuă prelucrarea numai dacă acestea au nivelul pus în corespondenţă cu execuţia cerută.

Aceasta este de fapt cauza necesităţii standardizării răspunsului pe care îl oferă funcţiile în domeniul valorilor de stare pe care le returnează.

Secvenţa:

```
. . .
int stare;
. . .
stare = f1(p1  . . .  pn );
if (stare != 0)
     return(stare);
stare=f2 (p1. . . . .pn );
if (stare != 0)
     return(stare);
. . .
```

ilustrează controlul permanent al programatorului asupra rezultatelor prelucrării folosind variabila elementară *stare*.