# CAPITOLUL 1. ALGORITMI

## 1.1. Noţiuni generale

Noţiunea de algoritm este primară, nu se poate defini. În schimb algoritmul poate fi descris, în esenţă putând să îl privim ca pe o succesiune de etape care se pot aplica mecanic, în vederea obţinerii unui anumit rezultat.

De exemplu algoritmul prin care se adună două fracţii presupune etapele: fracţiile se aduc la acelaşi numitor, se fac înmulţirile, adunările, apoi fracţia este simplificată.

Se pot face următoarele ***observaţii, valabile pentru orice algoritm***:

1. Se porneşte de la ceva (*datele de intrare*) şi se doreşte obţinerea unui anumit rezultat (*datele de ieşire*).

În exemplul de mai sus, se porneşte de la cele două fracţii (datele de intrare) şi se obţine suma lor (aceasta reprezentând singura dată de ieşire).

2. Se operează cu anumite *"obiecte"* asupra cărora sunt permise anumite *operaţii*.

În exemplul de mai sus, obiectele cu care se operează pot fi valori numerice sau simbolice (cu ajutorul cărora se reprezintă numărătorul şi numitorul fiecărei fracţii), iar operaţiile sunt descrise de reguli matematice.

3. În cele mai multe cazuri cel care elaborează algoritmul este diferit de executant.

În cele ce urmează se va trata numai cazul elaborării algoritmilor pentru programarea calculatoarelor, executantul fiind calculatorul iar cel care elaborează algoritmul fiind programatorul.

Frecvent intervine încă o persoană, diferită de programator, utilizatorul, căreia îi este necesară doar o pregătire minimă de specialitate în informatică, trebuind doar să utilizeze programul obţinut şi să beneficieze de avantajele lui.

## 1.2. Enunţul unei probleme, date de intrare şi de ieşire, etapele rezolvării unei probleme

Considerăm pentru exemplificare următoarea funcţie: f:R->R, $f(x)=\begin{cases} -x & pentru\ x<0 \\ x^2 & pentru\ x\geq 0 \end{cases}$.

Se cere calculul valorilor pe care le ia funcţia f pentru fiecare x dintr-un un set de 10 valori, furnizate ca şi date de intrare.

Pentru fiecare dintre cele 10 valori trebuie executat următorul grup de operaţiuni (grup pe care îl vom nota G1).

- Compararea lui *x* cu 0;
- În funcţie de rezultatul comparaţiei, alegerea ramurii care furnizează expresia de calcul pentru *f* şi calculul lui *f*;
- Afişarea perechii corespunzătoare (*x*,*f*).

Executarea manuală a acestor operaţiuni este plicticoasă, supusă erorilor şi mare consumatoare de timp. Problema se poate rezolva mai repede şi fără erori cu ajutorul unui program. Acesta se obţine prin codificarea algoritmului corespunzător într-un limbaj de programare.

În linii mari, **etapele obţinerii unui algoritm** sunt:

1. *Identificarea datelor de intrare şi a celor de ieşire*. În exemplul nostru, datele de intrare sunt $x_1$, $x_2$,..., $x_{10}$ iar cele de ieşire sunt cele 10 perechi de forma (*x*,*f*).
2. *Elaborarea algoritmului de rezolvare a problemei*.

Algoritmul specifică operaţiile pe care calculatorul trebuie să le efectueze pentru ca, pornind de la datele de intrare să obţină datele de ieşire.

Pentru exemplul nostru, aceste operaţii au fost prezentate deja (grupul de operaţii G1).

Algoritmul a fost descris în limbaj natural, în practică utilizându-se limbaje de tip pseudocod sau, mai rar, scheme logice.

Algoritmii se scriu în pseudocod pentru a se elimina diferenţele generate de regulile proprii fiecărui limbaj.

Un *limbaj de tip pseudocod* are caracteristicile:

a) Nu are multe reguli, dar seamănă cu orice limbaj de programare.

b) Un algoritm redactat în pseudocod nu poate fi rulat direct pe calculator. Trebuie convertit în limbajul de programare dorit.

c) Permite o conversie uşoară în orice limbaj de programare.

3. *Transpunerea algoritmului într-un limbaj de programare* (Pascal, C, etc.)
4. *Testarea programului şi corectarea sa* până când acesta funcţionează corect.

Observaţie Algoritmul poate fi codificat direct în limbajul de programare dorit, dacă programatorul are suficientă experienţă.

## 1.4. Obiectele cu care lucrează algoritmii şi operaţii permise

### 1.4.1. Date

După tipul lor, datele se pot clasifica în 3 categorii mari:

a) *Numerice*, cu tipurile *întreg*, respectiv *real*. În informatică printr-o *dată reală* se înţelege o valoare cu un număr finit de zecimale, iar virgula care separă partea întreagă de cea zecimală se reprezintă prin simbolul "." (punct zecimal).

*b) Logice*. O astfel de dată poate lua doar două valori: *TRUE* şi *FALSE*.

c) *Şir de caractere*. O astfel de dată este reprezentată de un şir de caractere cuprins între apostroafe. Un exemplu de dată de acest tip este "aaa-rfger8 fff".

*Observaţie* O categorie aparte de date este formată din *constante*, care pot avea oricare dintre tipurile prezentate. Constantele nu pot fi citite şi nici obţinute din calcule.

## 1.4.2. Variabile

În exemplul de la paragraful § 1.2 operațiile din grupul *G1* fac referire la $x_i$ pentru $\forall$ i=1,10 folosind notația unică *x*. Asemănător nu s-a folosit *f(xi)* , ci simplu *f*. Acest lucru este posibil deoarece atunci când se calculează $f(x_1)$ în x este memorat $x_1$ iar în *f* este memorat $f(x_1)$, când se calculează $f(x_2)$ în *x* este memorat $x_2$ iar în *f* este memorat $f(x_2)$, ș.a.m.d. Pentru algoritmul respectiv *x* și *f* sunt *variabile*.

Variabilele pot fi imaginate ca niște cutiuțe care rețin date. Principalele caracteristici ale unei variabile sunt:

1. Are un nume.
2. Are un tip, care determină natura datelor care pot fi reținute de variabila respectivă. Se folosesc notațiile *integer* pentru tipul întreg, *real* pentru tipul real, *boolean* pentru variabilele logice şi respectiv *string* pentru variabilele de tip şir de caractere.
3. Trebuie declarată (anunțată) înainte de a o utiliza.

Exemplu: *integer a;*
*boolean c,d;*

În acest exemplu s-au declarat variabilele de tip întreg *a* şi respectiv de tip logic *c,d*.

4. I se rezervă spațiu de memorie în memoria internă a calculatorului.

## 1.4.3. Expresii

În scopul efectuării calculelor, algoritmii folosesc expresii. O expresie este alcătuită din unul sau mai mulți *operanzi*, legați între ei prin *operatori*.

Operanzii pot fi constante sau variabile.

Operatorii desemnează operațiile care se execută spre a obține rezultatul şi pot fi de 3 feluri: aritmetici, relaționali şi respectiv logici.

Modul în care se face legătura între operanzi şi operatori respectă anumite *reguli sintactice*.

În timpul execuției, *expresiile sunt evaluate* (se efectuează calculele impuse de operanzi).

**Exemplul 1**. Fie două variabile de tip întreg: *a* reține valoarea 1 şi *b* reține valoarea *3*. Considerăm expresia:

(a)+(3)*((b)+(2))

În această expresie s-au marcat prin cerculețe operanzii, iar + şi * sunt operatori. În urma evaluării expresiei se obține valoarea *întreagă* 16.

**Exemplul 2**. Fie *a* o variabilă întreagă ce reține valoarea 3 şi *b* o variabilă reală ce reține valoarea 1.9. În urma evaluării expresiei

(a)>(b)

se obține valoarea logică TRUE.

### 1.4.3.1. Operatori aritmetici

Operatorii aritmetici sunt de două feluri:

a) *Operatori unari*. Există doi operatori unari: + şi respectiv -. Aceştia acționează asupra unui singur operand, care poate fi o variabilă sau o constantă de tip numeric. Operandul se află totdeauna la dreapta operatorului unar.

Exemplu: Fie *a* o variabilă întreagă ce reține valoarea 2. Atunci evaluarea expresiei - *a* are ca rezultat valoarea - 2, pe când evaluarea expresiei - - *a* are ca rezultat valoarea 2.

b) *Operatori binari* . Următorii operatori sunt de tip binar: +, - , * , / , *div*, *mod*. Aceşti operatori apar în construcții sintactice de forma:

*O1 OB O2*

unde prin O1 şi respectiv O2 s-au notat operanzii, iar prin OB s-a notat operatorul binar.

b1) Operatorul + semnifică adunarea. Într-o construcție de forma O1+O2, O1 şi O2 pot fi de tip întreg sau real. Dacă cel puțin unul dintre cei doi operanzi este real, rezultatul evaluării expresiei O1+O2 este de tip real, altfel este de tip întreg.

Exemplu Dacă *c* este o variabilă de tip real ce conține valoarea 2.3, atunci expresia *c+1* produce valoarea 3.3, rezultatul fiind deci de tip real.

b2) Operatorul - semnifică scăderea.

b3) Operatorul * semnifică înmulțirea.

*Observație* La operatorii - şi respectiv * se aplică aceleaşi reguli sintactice ca şi la operatorul +.

b4) Operatorul / semnifică împărțirea reală. Operanzii pot fi de tipul întreg sau real dar *totdeauna rezultatul este de tip real*.

**Exemplu** 3/2 are ca rezultat valoarea reală 1.5, iar 4/2 are ca rezultat valoarea **reală** 2.0.

b5) Operatorul DIV semnifică împărțirea întreagă. Apare în construcții sintactice de forma *O1 DIV O2* .O1 şi O2 sunt obligatoriu de tip întreg, rezultatul evaluării acestei expresii fiind partea întreagă a câtului obținut după împărțirea operanzilor.

*Observație* DIV furnizează rezultatul corect numai dacă ambele valori sunt numere naturale.

Exemplu 14 DIV 5 furnizează rezultatul 2.

b6) Operatorul MOD are semnificația de rest al împărțirii pentru numere întregi. Apare în construcții sintactice de forma *O1 MOD O2*. O1 şi O2 sunt obligatoriu de tip întreg.
Exemplu 14 DIV 5 furnizează rezultatul 4.

### 1.4.3.2. Operatori relaționali

Există 5 operatori relaționali: >, >=, <, <= şi =.

Aceşti operatori sunt totdeauna binari şi apar în construcții sintactice de forma:

*O1 OR O2,*

unde *O1* şi *O2* sunt operanzii iar *OR* este unul dintre cele 5 tipuri de operatori relaționali. Operanzii pot fi variabile sau constante de oricare dintre tipurile prezentate anterior. În acest capitol se va studia numai modul de acționare al operatorilor relaționali asupra operanzilor de tip numeric (întreg sau real).

Rezultatul evaluării unei expresii *E* de forma *E=O1 OR O2* este totdeauna o valoare logică.
De exemplu după evaluarea expresiei *2<3* se obține valoarea TRUE.

### 1.4.3.3. Operatori logici

Se definesc două tipuri de operatori logici: unari şi binari.

Există un singur operator logic **unar** , şi anume NOT (negare), care acționează astfel:
*NOT(TRUE)=FALSE* şi respectiv *NOT(FALSE)=TRUE*.

Toți ceilalți 3 operatori logici sunt **binari**, definindu-se operatorii binari: *AND, OR* şi respectiv *XOR*. Aceştia apar în construcții sintactice de forma:

*O1 OL O2,*

unde *O1* şi *O2* sunt operanzii iar *OL* este unul dintre cele 3 tipuri de operatori logici.
*O1* şi *O2* pot fi de tip numeric sau logic, dar în cele ce urmează ne vom referi numai la operanzi de tip logic.

În tabelul următor se centralizează rezultatele evaluării expresiilor de tip *E=O1 OL O2* folosind convenția: 0 - FALSE şi respectiv 1- TRUE.

| *O1* | *O2* | *O1 AND O2* | *O1 OR O2* | *O1 XOR O2* |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

Se observă că:
- la aplicarea operatorului *AND* se obține *TRUE* numai dacă ambii operanzi sunt TRUE;
- la aplicarea operatorului *OR* se obține *FALSE* numai dacă ambii operanzi sunt FALSE;
- la aplicarea operatorului *XOR* se obține *TRUE* numai dacă operanzii diferă.

### 1.4.3.4. Prioritatea operatorilor

Operatorii sunt împărțiți în 4 grupe după prioritate. Cei din grupa 1 au prioritate maximă, apoi prioritatea scade atunci când numărul grupei creşte, cei din grupa 4 având prioritatea minimă.

Împărțirea operatorilor după grupe de prioritate se face astfel:

- grupa 1 conține operatorii UNARI;
- grupa 2 conține operatorii MULTIPLICATIVI (*AND, *, /, DIV, MOD*);
- grupa 3 conține operatorii ADITIVI (*OR, XOR, +,* -);
- grupa 4 conține cei 5 operatori RELATIONALI.

### 1.4.3.5. Evaluarea expresiilor

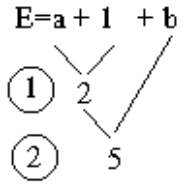


Fig. 1.1

Pentru a înțelege modul în care se evaluează o expresie *E* vom prezenta următoarele 4 cazuri distincte:

**Cazul 1** Expresia *E* nu conține paranteze şi este formată din mai mulți operanzi, legați prin operatori de acelaşi tip. Expresiile de acest tip se evaluează de la stânga la dreapta.
*Exemplu:* Fie variabilele *a* şi *b* de tip întreg, *a=1* şi *b=3*. Expresia *E=a+1+b* se evaluează în ordinea descrisă de numerele încercuite din figura 1.1

**Cazul 2** Expresia *E* nu conține paranteze. În expresie intervin operatori care nu sunt de acelaşi tip dar au aceeaşi prioritate.
Ca şi în cazul 1, expresia se evaluează de la stânga la dreapta.
*Exemplu:* expresia *2*3 div 4* se evaluează ca în figura 1.2.

**Cazul 3** Expresia *E* nu conține paranteze. În interiorul său apar mai mulți operatori care au priorități diferite. În acest caz se efectuează mai întâi, de la stânga la dreapta, operațiile descrise de operatorii cu prioritate maximă. Procedeul se reia până când sunt efectuate toate operațiile cerute de operatorii cu cea mai mică prioritate existenți în expresie.
*Exemplul 1* Expresia *2+3*5 –6/2* se evaluează ca în figura 1.3, cu observația că, datorită operatorului de împărțire reală se va obține un număr real.
*Exemplul 2* Expresia *FALSE OR TRUE AND TRUE* se evaluează ca în fig. 1.4.

**Cazul 4** Acesta este cazul cel mai general. Comparativ cu cazul 3, aici apar şi paranteze.

Pentru evaluarea unei expresii care se încadrează în acest caz se procedează astfel: se parcurge de la stânga la dreapta expresia pentru a identifica prima paranteză deschisă. Apoi se continuă parcurgerea de la stânga la dreapta,

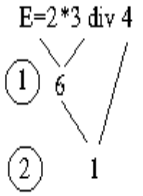


Fig. 1.2

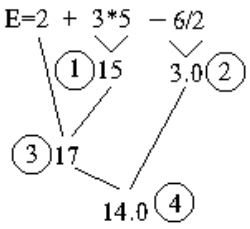


Fig. 1.3

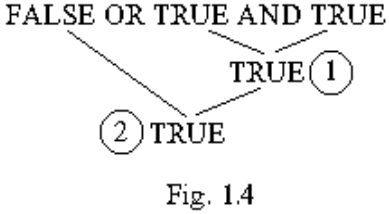


Fig. 1.4

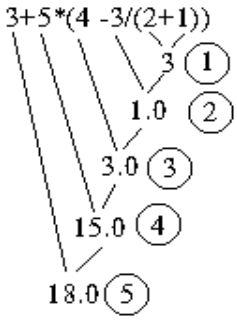


Fig. 1.5

putând exista două cazuri:
a) Se detectează întâi paranteza de închidere. Se evaluează expresia dintre paranteze ca şi în cazul 3.
b) Se detectează încă una (sau mai multe) paranteze de deschidere. Dacă s-au detectat *n* paranteze de deschidere, se evaluează întâi expresia delimitată la stânga de cea de a *n*-a paranteză, apoi cea delimitată la stânga de cea de a (*n-1)*-a paranteză, etc., aplicându-se regulile de la cazul 3.
*Exemplu*: Expresia *3+5*(4 –3/(2+1))* se evaluează ca în figura 1.5.

### 1.4.3.8. Tipul expresiilor

La evaluarea expresiilor se obţine un rezultat. *Tipul rezultatului* stabileşte *tipul expresiei*. Acesta este determinat de tipurile operanzilor şi de operatorii aplicaţi în cadrul expresiei:
Exemple: tipul expresiei *4>2* este *boolean*, tipul expresiei *1+2* este *integer*, iar tipul expresiei *1.3+2* este *real*.

## 1.5. Operaţiile pe care le efectuează un algoritm

### 1.5.1. Operaţii de intrare/ieşire

Prin *operaţia de intrare (citire)* se înţelege preluarea unei date de la un dispozitiv de intrare către memoria internă a calculatorului, în zona de memorie rezervată pentru aceasta, adică în variabila care memorează data respectivă.

Dispozitivele de intrare cele mai cunoscute sunt: tastatura (care este şi cel mai des utilizată), o unitate de disc (flexibil, CD-ROM, hard disk), etc.

Prin *operaţia de ieşire (scriere)* se înţelege preluarea unei date din memoria internă a calculatorului (adică din variabila care o memorează) către un dispozitiv de ieşire. Cele mai cunoscute dispozitive de ieşire sunt: monitorul, o unitate de disc, imprimanta, etc.

În pseudocod vom folosi cuvântul cheie *read* pentru a desemna o operaţie de citire şi respectiv vom folosi cuvântul cheie *write* pentru a desemna o operaţie de scriere.

```
integer a;
read a;
write a;
```
Caseta 1_5_1_1

Exemplu: Presupunem că dispozitivul de intrare este tastatura iar cel de ieşire este monitorul. Atunci algoritmul pentru citirea şi apoi afişarea unui număr întreg este descris in caseta 1_5_1_1.

În acest exemplu se observă că pentru început se declară variabila *a*, care este de tip întreg. Apoi, prin instrucţiunea *read a* se aşteaptă introducerea de la tastatură a unei date de tip întreg. După introducerea datei, aceasta este scrisă (afişată) pe monitor.

*Observaţii*

1. După scriere, conţinutul variabilei rămâne nemodificat.
2. La o nouă citire, conţinutul vechi al variabilei se pierde.

```
integer a;
read a;
read a;
write a;
```
Caseta 1_5_1_2

*Exemplu* Presupunem că introducem două numere întregi de la tastatură (de exemplu 7 şi respectiv 20, în această ordine). Se execută secvenţa de instrucţiuni din caseta 1_5_1_2:

În urma primei citiri *a* a primit valoarea 7, în urma celei de a doua citiri *a* a primit valoarea 20, cu care rămâne. Deci se va afişa 20.

3. Se pot citi mai multe variabile cu o singură operaţie *read* şi se pot tipări valorile reţinute de mai multe variabile folosind o singură operaţie *write*.

```
real a,b,c;
write c,a,b;
```
Caseta 1 5 1 3

*Exemplu* Fie secvenţa de instrucţiuni din caseta 1_5_1_3:

Dacă introducem setul de valori numerice *1.2, -0.7, 1.5*, *a* va reţine 1.2, *b* va reţine -0.7 şi *c* va reţine 1.5. În urma execuţiei instrucţiunii *write c,a,b,* pe monitor vor apare 1.5, 1.2, -0.7.

4. O practică des întâlnită în programare este aceea prin care o instrucţiune de citire este precedată de afişarea unui mesaj pe ecran, prin care utilizatorul este atenţionat că se aşteaptă introducerea unei date sau a unui set de date.

### 1.5.2. Atribuiri

Prin *operaţia de atribuire* se reţine o anumită dată într-o variabilă. Această operaţie are 3 forme:

**Forma 1 v:=dată**

unde *v* este numele unei variabile de un tip oarecare, iar *dată* reprezintă o valoare de un tip oarecare.

Tipul variabilei trebuie să coincidă cu tipul valorii atribuite (ex. 1, caseta 1_5_2_1), cu o singură excepție: unei variabile de tip real i se poate atribui o dată de tip întreg (ex. 2, , caseta 1_5_2_1).

| Ex. 1 | Ex. 2 |
|---|---|
| *real b;* | *real d;* |
| *b:=-7.25;* | *d:=7;* |

Caseta 1_5_2_1

**Forma 2**

*v1:=v2*

unde *v1* şi *v2* sunt nume de variabile. Cu o singură excepție (aceeaşi de la forma 1), tipul variabilelor trebuie să coincidă. Efectul operației de atribuire este că variabila *v1* va reține conținutul variabilei *v2*. După efectuarea operației, conținutul variabilei *v2* rămâne nemodificat, iar conținutul inițial al variabilei *v1* se pierde.

| Ex. 1 | Ex. 2 |
|---|---|
| *integer a,b;* | *integer a,b;* |
| *a:=2; b:=3;* | *a:=2; b:=3;* |
| *a:=b;* | *b:=a;* |

Caseta 1_5_2_2

În ex. 1 (caseta 1_5_2_2) inițial *a* conținea valoarea 2, iar după atribuirea *a:=b* atât *a* cât şi *b* vor conține valoarea 3. În ex. 2, după atribuirea *b:=a* atât *a* cât şi *b* vor conține valoarea 2.

Este foarte important de reținut că atribuirea *a:=b* nu are acelaşi efect cu atribuirea *b:=a.*

**Forma 3**

*v:=expresie*

În acest caz, întâi se evaluează expresia, iar valoarea obținută este atribuită variabilei *v*.

*Observație* După atribuire, conținutul variabilelor care sunt operanzi în expresia din membrul drept rămâne nemodificat. Excepție face, eventual, variabila *v* dacă ea figurează şi ca operand în expresie.

*Exemplul 1* După evaluarea expresiei *v:=a+2*b/c*, singura variabilă care se modifică este *v*, conținutul variabilelor *a,b* şi *c* care participă ca şi operanzi în expresie rămânând nemodificat.

Formele 1 şi 2 sunt de fapt particularizări ale formei 3.

*Exemplul 2* Fie o variabilă *v* de tip întreg care conține valoarea 7. După atribuirea *v:=v+1*, variabila *v* va conține valoarea 8. Întâi s-a evaluat valoarea expresiei *v+1=8*, apoi *v* a primit această valoare, distrugându-se, prin suprascriere, vechiul conținut al lui *v*. O eroare des întâlnită este atunci când atribuirea este confundată cu o ecuație, în care se egalează cei doi membri. Pentru cei care fac această confuzie, o operație de forma *v:=v+1* este eronată, conducând la inepția 0=1! ***Atribuirea nu trebuie confundată cu o ecuație.***

**Exemple tipice de utilizare a atribuirii**

```
S:=0;
P:=1;
```

Caseta 1_5_2_3

**1. Inițializări**

Stabilirea valorilor cu care anumite variabile intră în calcule se numeşte *inițializare*. Inițializările se fac cu ajutorul instrucțiunii de atribuire. Două exemple sunt furnizate în caseta 1_5_2_3.

Aceste două exemple sunt tipice pentru cazul în care *S* va reține valoarea unei sume iar *P* va reține valoarea unui produs, pentru că 0 este elementul neutru pentru adunare iar 1 este elementul neutru pentru înmulțire.

**2. Calcule**

Există în principal două forme în care atribuirea intervine în calcule

**Forma directă**

Unei variabile (o vom nota *v*) i se atribuie o expresie în care printre operanzi nu se regăseşte şi ea însăşi.

*Exemplu v:=a+b;*

**Forma indirectă**

Variabilei *v* i se atribuie o expresie în care participă şi ea însăşi ca operand:

*Exemplu v:=a+v*x.*

**3. Copiere**

De multe ori este necesar ca o valoare reținută de o variabilă să fie reținută şi de o altă variabilă.

Un caz particular, des întâlnit, este *interschimbarea conținutului a două variabile.*

Fie *x,y* două variabile al căror conținut dorim să îl interschimbăm. Dacă de exemplu inițial *x* conținea valoarea 3 şi *y* conținea valoarea 4, după interschimbare x va conține valoarea 4 iar y va conține valoarea 3.

| *x:=y;* | *y:=x;* |
|---|---|
| *y:=x;* | *x:=y;* |
| (a) | (b) |

Caseta 1_5_2_4

Un programator neatent ar putea utiliza secvența greşită de instrucțiuni din caseta 1_5_2_4 (a).

Rezultatul execuției acestei secvențe este: după ce se execută *x:=y*, *x* primeşte valoarea 4, pierzându-şi vechiul conținut. Apoi, după ce se execută *y:=x*, *y* primeşte tot valoarea 4, cea memorată actual de *x*. În consecință, în loc să obținem interschimbarea conținutului celor două variabile, obținem egalarea conținutului acestora cu valoarea memorată de *y*.

Dacă am fi scris aşa cum este descris în 1_5_2_4 (b), efectul ar fi fost egalarea conținutului celor două variabile cu *y*!

O secvență corectă de instrucțiuni trebuie să utilizeze o variabilă intermediară *m*, în care să se salveze conținutul uneia dintre variabile înainte ca aceasta să-şi piardă conținutul, urmând ca apoi cealaltă variabilă să folosească valoarea variabilei intermediare.

Ambele posibilități de rezolvare a interschimbării sunt prezentate în cele ce urmează:

| Varianta 1 | Varianta 2 |
|---|---|
| *m:=x;* | *m:=y;* |
| *x:=y;* | *y:=x;* |
| *y:=m;* | *x:=m;* |

## 1.5.3. Operații de decizie

```
if expresie logică
        then
        operația
1
            else

operația 2
        end if
```

În pseudocod şi în majoritatea limbajelor de programare operația de decizie este desemnată folosind cuvântul cheie ***if***.

În pseudocod, forma generală a operației de decizie este descrisă în fig. 1.6. (s-au subliniat cuvintele cheie).

Modul de execuție este următorul:

1. Se evaluează *expresie logică* (o vom nota cu E).
2. Dacă rezultatul evaluării lui E este TRUE, se execută *operația 1* ; dacă acest rezultat este FALSE, se execută *operația 2*.

**Exemplul 1**. Se citesc două numere întregi, *a* şi *b*. Se cere să se tipărească cel mai mare dintre ele. Secvența de instrucțiuni scrisă în pseudocod care realizează acest lucru este descrisă în fig. 1.7:

```
integer a,b;
read a,b;
if a>b
        then
        write a

        else

        write b
endif
```

**Exemplul 2**. Se citesc 4 valori reale *a,b,c,d*. Să se evalueze expresia:

$$E=\begin{cases} a+b & daca\ c+d>0 \\ a-b & daca\ c+d=0 \\ a*b & daca\ c+d<0 \end{cases}$$

Secvența de instrucțiuni scrisă în pseudocod care realizează acest lucru este descrisă în fig. 1.8.

Dacă presupunem ca exemplu numeric setul de date de intrare *a=1, b=2, c=3* şi *d=4*, atunci rezultatul afişat este *3*.

```
real a,b,c,d,E;
read a,b,c,d;
if (c+d)>0
        then
        E:=a+b
        else
                if
(c+d)=0

        then

        E:=a-b

else
```

*Observație*

Testul *if (c+d)<0* lipseşte deoarece atunci când *(c+d)* nu este nici mai mare şi nici egal cu 0, nu mai există decât o singură posibilitate, anume aceea ca *(c+d)* să fie mai mic decât 0.

***Forma particulară a operației if*** se foloseşte atunci când pe ramura *else* nu mai trebuie executată nici o operație. Această formă particulară se codifică aşa cum este descris de fig. 1.9:

```
if E
        then
        operatie
end if
```

Fig. 1.9

Modul de execuție este următorul:

1. Se evaluează *expresia logică* notată cu *E*.
2. Dacă rezultatul evaluării lui E este TRUE, se execută *operație*.

# CAPITOLUL 2. PRINCIPIILE PROGRAMĂRII STRUCTURATE

## 2.1. Generalități

Un algoritm poate fi reprezentat *în mod simplificat* ca în fig. 2.1, ordinea de execuție a operațiilor find indicată de săgeată. O problemă specială apare atunci când o anumită operație trebuie executată de mai multe ori.

În exemplul următor se propune obținerea (prin calcul iterativ) a sumei primelor 1000 de numere naturale. Nu se va aplica formula directă S=n*(n+1)/2.

O primă soluție este prezentată în fig. 2.2. Soluția prezentată în această figură prezintă dezavantajul scrierii repetate. În acest context a apărut *operația de salt* , a cărei sintaxă este

*GOTO etichetă*.

Atunci când, pe parcursul execuției , se întâlneşte această operație, controlul execuției se transferă la linia marcată prin "etichetă".

Folosind GOTO, algoritmul propus ca exemplu se rescrie aşa cum este prezentat în fig. 2.3:

În acest fel am programat *un ciclu*, adică o secvență de operații care se repetă. Utilizarea operației *GOTO* prezintă anumite dezavantaje, un astfel de algoritm fiind greu de urmărit chiar şi pentru cei care l-au elaborat.

Datorită dezavantajelor operației *GOTO* a apărut conceptul de *programare structurată*, care a condus la uşurarea

```
operația 1
operația 2
...
operația n
```

Fig. 2.1

```
integer S;
S:=0;
S:=S+1;
S:=S+2;
.....
S:=S+1000;
write S;
```

1000 operatii

Fig. 2.2

```
        integer S,i;
        S:=0;
        i:=1;
reia:   S:=S+i;
        i:=i+1;
        if  i<=1000 then GOTO reia
        endif
        write S;
```

Fig. 2.3

scrierii algoritmilor, respectiv a citirii (urmăririi) lor.

*Definiție* Prin *structură* înțelegem o anumită formă de îmbinare a operațiilor cu care lucrează algoritmii. În cadrul programării structurate, algoritmii se elaborează prin utilizarea *exclusivă* a anumitor structuri.

Programarea structurată nu înseamnă doar eliminarea operației de salt. Înseamnă de fapt un alt mod de gândire a algoritmilor.

Astfel, problema pentru care se elaborează algoritmul se descompune după structurile permise, rezultatul descompunerii fiind mai multe *subprobleme*. La rândul lor, subproblemele se descompun din nou, după aceleaşi reguli, până se ajunge la operaţii elementare.

## 2.2. Structuri de bază şi descrierea lor în pseudocod

Există 3 tipuri de structuri de bază utilizate în descrierea algoritmilor: liniară, alternativă şi respectiv repetitivă.

### 2.2.1. Structura liniară

Definim structura liniară astfel:

- orice operaţie (citire, scriere, atribuire, decizională) considerată ca un tot unitar se constituie într-o structură liniară;
- dacă S1 şi S2 sunt structuri (de orice tip) atunci secvenţa $\begin{matrix} S1 \\ S2 \end{matrix}$ reprezintă o structură liniară.

Pornind de la această definiţie, se ajunge la forma de structură liniară din fig. 2.1.

Considerăm acum următoarea problemă: se citeşte un număr natural format din 3 cifre. Să se afişeze suma cifrelor sale. (Dacă de exemplu numărul introdus este 217, se va afişa 10, provenit prin însumarea 2+1+7).

Pentru rezolvarea acestei probleme, apelăm la operatorii aritmetici *div* şi *mod*. O soluţie a acestei probleme este reprodusă în cele ce urmează:

| Observaţii | | *Exemplu numeric* |
|---|---|---|
| | *integer x,S;* | |
| | *read x;* | |
| se iniţializează suma | *S:=0;* | S:=0; |
| se adună ultima cifră | *S:=S+x mod 10;* | 217 mod 10=7, deci S:=0+7=7 |
| *se reţin doar primele cifre din număr,* eliminându-se ultima | *x:=x div 10;* | x:=217 div 10, deci x:=21 |
| se adună penultima cifră | *S:=S+x mod 10;* | 21 mod 10=1, deci S:=7+1=8 |
| se reţin doar primele cifre din numărul trunchiat, eliminându-se ultima | *x:=x div 10;* | x:=21 div 10, deci x:=2 |
| se adună prima cifră | *S:=S+x mod 10;* | 2 mod 10=2, deci S:=8+2=10 |
| | *write S;* | |

Se observă caracterul repetitiv al operaţiilor S:=S+x mod 10, respectiv x:=x div 10. Algoritmul poate fi rescris mai eficient, folosind structuri repetitive, descrise în paragraful 2.2.3.

### 2.2.2. Structura alternativă

Definim structura alternativă astfel: dacă S1 şi S2 sunt două structuri şi E este o expresie logică, atunci secvenţa reprezentată în fig. 2.4 este o structură alternativă.

```
if E
   then
      S1
        else
        S2
end if
```

Fig. 2.4.

Mecanismul de execuţie al acestei structuri este următorul:

- se evaluează expresia E
- dacă E are valoarea TRUE, se execută S1, altfel se execută S2.

*Observaţie* Structura alternativă reprezintă o *generalizare a operaţiei decizionale*. Caracterul de generalitate provine din faptul că fie sub *then*, fie sub *else* pot exista mai multe operaţii, nu ca în cazul operaţiei decizionale, când era admisă o singură operaţie.

Ca şi în cazul operaţiei decizionale, şi structura alternativă admite o formă particulară, reprodusă mai jos:

```
if E
    then S
end if
```

```
integer a,b,S1,comanda;
real x,y,S2;
read comanda;
if comanda=0
    then
                read
a,b;
        S1:=a+b;
                write
S1;
        else
                read
x,y;
        S2:=x+y;
                write
```

*Exemplu* Se citeşte un număr natural, care este reţinut în variabila *comanda*. Dacă numărul citit este 0, se vor citi numerele întregi *a* şi *b* şi se va tipări suma lor (reţinută de variabila *S1*). În cazul în care numărul citit este diferit de 0, se vor citi numerele reale *x* şi *y* şi se va tipări suma lor (reţinută de variabila *S2*). Soluţia propusă este reprodusă în fig. 2.5.

### 2.2.3. Structura repetitivă

Structura repetitivă poate apare în mai multe variante, descrise în cele ce urmează:

1. Structură repetitivă *condiţionată anterior* (cu test iniţial), existând două tipuri de astfel de structuri:
a) Structura WHILE DO;
b) Structura FOR.

2. Structură repetitivă *condiţionată posterior* (cu test final), adică structura REPEAT UNTIL.
Singura structură repetitivă indispensabilă este WHILE DO, celelalte putându-se obţine din aceasta.

### 2.2.3.1. Structura WHILE DO

Structura WHILE DO se defineşte astfel: dacă E este o expresie logică şi S este o structură, atunci secvenţa reprezentată în fig. 2.6 reprezintă o structură de tip WHILE DO. Principiul de execuţie al acestei structuri este următorul:

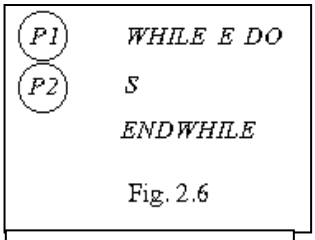


Fig. 2.6

Pasul 1 (marcat în figură prin P1 încercuit): se evaluează expresia logică E şi dacă se obţine TRUE, se trece la pasul 2 (marcat în figură prin P2 încercuit); dacă se obţine FALSE, execuţia structurii se încheie.

Pasul 2: Se execută structura S şi se reia execuţia de la pasul 1.

Deoarece S se execută numai dacă E ia valoarea TRUE, structura face parte dintre structurile condiţionate anterior.

Utilizarea structurii WHILE DO este obligatorie dacă se îndeplinesc simultan două condiţii:

1. S se execută (de câte ori este cazul) *numai dacă este îndeplinită o anumită condiţie* (cea exprimată cu ajutorul expresiei E).
2. Nu se cunoaşte de câte ori se va executa S (este vorba despre un *ciclu cu număr necunoscut de paşi*).

***Exemplu*** Se citeşte un număr natural x, x≠ 0. Să se tipărească suma cifrelor sale (exemplul a mai fost prezentat în paragraful 2.2.1, dar fără a se utiliza structuri repetitive). O soluţie a problemei, folosind structura WHILE DO, este reprodusă în fig. 2.7.

```
integer x,S;
read x;
S:=0;
WHILE x<>0 DO
        S:=S+x mod 10;
        x:=x div 10;
ENDWHILE
write S;
```

### 2.2.3.2. Structura de tip FOR

Fie *i* o variabilă de tip întreg, numită *variabilă de ciclare*. Fie *a* şi *b* două variabile de tip întreg, numite *valoare iniţială*, respectiv *valoarea finală* şi *S* o structură. Atunci secvenţa reprezentată în fig. 2.8 reprezintă o structură de tip FOR. Principiul de execuţie al acestei structuri este următorul:

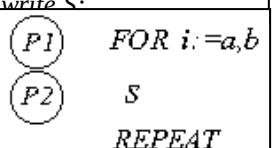


Fig. 2.8

Pasul 1 (marcat în figură prin P1 încercuit): *i* ia valoarea iniţială *a.*

Pasul 2 Dacă $i \le b$, se execută *S*, apoi se incrementează *i* (se efectuează *i:=i+1*) şi se reia execuţia de la pasul 2. În caz contrar (*i>b*) execuţia structurii ia sfârşit.

Structura FOR poate fi simulată folosind WHILE DO aşa cum se reprezintă în fig. 2.9.

```
i:=a;
WHILE  i<=b DO
     S
i:=i+1;
ENDWHILE
```
Fig. 2.9

Structura FOR se utilizează dacă sunt îndeplinite simultan următoarele două condiţii:

- execuţia structurii S se repetă;
- se cunoaşte dinainte de câte ori se execută S.

Se spune că *FOR* este un *ciclu cu un număr cunoscut de paşi.*

***Exemplul 1*** Se citeşte un număr natural *n*. Să se calculeze şi afişeze suma tuturor numerelor naturale mai mici sau egale cu *n* (problema a mai fost tratată în paragraful 2.1). O soluţie a problemei folosind ciclul FOR este prezentată în fig. 2.10.

```
integer n, S, i;
read n;
S:=0;
FOR i:=1,n
S:=S+i;
REPEAT
write S;
```
Fig. 2.10

***Exemplul 2*** Se citeşte un număr natural *n*. Să se calculeze şi afişeze următoarea sumă:

$$S=\sum_{i=1}^{n}\prod_{j=1}^{i} j = 1+1*2+1*2*3+...+1*2*3*...*n$$

Numărul de paşi ai ciclului FOR este *n* (este dat de numărul de termeni ai sumei). Fiecare al *i*-lea termen al sumei se calculează ca un produs de forma 1*2*3*...*i, dar calculul se poate simplifica dacă se calculează pe baza termenului de ordinul (*i-1)* înmulţindu-l pe acest cu *i*.

```
integer n, S, P, i;
read n;
S:=0;
P:=1;
FOR i:=1,n
P:=P*i;
S:=S+P;
REPEAT
write S;
```
Fig. 2.11

O soluţie a problemei folosind ciclul FOR este prezentată în fig. 2.11.

***Exemplul 3*** Se citesc *n* numere întregi. Să se afişeze cel mai mare dintre ele.

O soluţie posibilă a problemei utilizează doar 3 variabile: *n*, care reţine numărul de numere citie, variabila *nr* care reţine ultimul număr citit, şi respectiv variabila *max* în care se reţine maximul dintre numerele citite şi analizate până la momentul respectiv al execuţiei.

O soluţie a problemei folosind ciclul FOR este prezentată în fig. 2.12.

```
integer i, n, max,nr;
read n;
read nr;
max:=nr;
FOR i:=2,n
read nr;
if nr>max
     then
          max:=nr;
end if
REPEAT
write max;
```
Fig. 2.12

### 2.2.3.3. Structura de tip REPEAT UNTIL

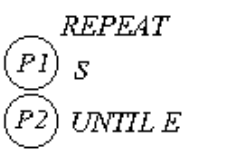


Fig. 2.13

REPEAT UNTIL este o structură repetitivă *condiţionată posterior (cu test final).*

Structura REPEAT UNTIL se defineşte astfel: dacă E este o expresie logică şi S este o structură, atunci secvenţa

reprezentată în fig. 2.13 reprezintă o structură de tip REPEAT UNTIL. Principiul de execuție al acestei structuri este următorul:
Pasul 1: Se execută S;
Pasul 2: Se evaluează expresia logică E. Dacă la evaluare se obține FALSE, se trece la pasul 1. În caz contrar execuția structurii se încheie.
*Observație* Structura S se execută totdeauna cel puțin o dată.

```
WHILE NOT E
S
ENDWHILE
```
Fig. 2.14

Structura REPEAT UNTIL se poate simula cu ajutorul structurii WHILE DO aşa cum se înfățişează în fig. 2.14.
*Exemplu* Să se calculeze şi afişeze suma primelor *n* numere naturale (ex. 1 de la paragraful 2.2.3.2).
O soluție posibilă a acestei probleme folosind structura REPEAT UNTIL este prezentată în fig. 2.15.

```
integer n, i, S;
read n;
i:=1;
S:=0;
REPEAT
S:=S+i;
i:=i+1;
UNTIL i>n;
write S;
```
Fig. 2.15

# CAPITOLUL 3. ELEMENTE DE BAZÁ ALE LIMBAJULUI C

## 3.1. Generalități

Limbajul C++ are câteva caracteristici majore, cele mai importante fiind:
1. Este extrem de *flexibil*, acest lucru însemnând că aceeaşi secvență scrisă în pseudocod poate fi codificată în mai multe feluri.
2. Permite ca anumite secvențe să poată fi scrise "ermetic", dar acest lucru prezintă un mare dezavantaj: după o anumită perioadă, chiar şi cel care le-a scris are dificultăți în înțelegerea lor.
Ex: a+=++b reprezintă o scriere "ermetică" pentru secvența de instrucțiuni: b=b+1; a=a+b.

Limbajul C++ conține, pe lângă setul de instrucțiuni, şi o mulțime de funcții care au rolul de a ajuta programatorul în elaborarea programelor.

Atunci când o firmă (ex. Borland, Microsoft, etc.) vinde un mediu de dezvoltare C++, acesta trebuie să includă toate funcțiile cerute de *standardul pentru limbaj* elaborat de ANSI (American National Standard Institute). Firma poate introduce şi alte funcții, neprecizate de standarde, dar utilizarea lor duce la crearea de programe neportabile.

## 3.2. Structura programelor

Un program C este alcătuit din una sau mai multe funcții. Prin respectarea structurilor (liniară, alternativă, repetitivă), orice algoritm poate fi descompus în alți algoritmi mai mici (subalgoritmi). În general se poate spune că:
1. Fiecărui algoritm sau subalgoritm îi o corespunde o anumită funcție care îl codifică în instrucțiuni.
2. Dacă funcția nu întoarce (nu returnează) nici un rezultat, spunem că tipul rezultatului este *void*. Funcțiile care returnează un rezultat se clasifică în funcții cu rezultat de tip întreg, de tip real, etc.
3. Un program este alcătuit din una sau mai multe funcții. Una dintre acestea este *funcția rădăcină* (numită *main*), care nu poate lipsi. Execuția programului începe automat cu funcția *main*. În general această funcție se foloseşte fără parametri.

La rândul ei, funcția *main* poate apela alte funcții definite de utilizator sau existente în biblioteci.

```
void main()
  {
  }
```
Fig. 3.1

Cel mai simplu program C arată ca în fig. 3.1.Programul e corect, deşi nu efectuează nici o operație. Funcția rădăcină este de tipul *void* - nu întoace nici un rezultat. Parantezele rotunde () se ataşează oricărei funcții. Între ele se scriu, opțional, anumiți parametri. În acest exemplu nu se folosesc parametri.

Între acoladele {} se scrie *corpul funcției*, format din instrucțiuni şi/sau apeluri de funcții. În exemplul nostru, mulțimea instrucțiunilor este vidă.

```
  --------
tip_functie_f  f()
  {                    } Definire functie f
  --------
  }
 void main()
  {
  --------
  f();      /* apelul functiei f */   } Functia principala
  --------
  }
  ----------
```
Fig. 3.2

4.Din punct de vedere sintactic, orice instrucțiune trebuie terminată cu caracterul ";".
*Grupurile de instrucțiuni* pot fi delimitate prin {} pentru a forma unități sintactice noi de tip *bloc*.

Dacă avem un program compus din două funcții, anume funcția principală şi o *funcție apelată f()*, atunci o posibilă structură a acestui program este reprodusă de fig. 3.2.
Observații
1. Se face diferență între litere mici şi litere mari.
2. Nu are importanță nici plasarea cuvintelor pe linie, nici numărul spațiilor dintre ele. Este corectă deci şi o scriere de forma :
*void main( ) { }*
3. Este permis ca tipul funcției să lipsească (să nu fie precizat). În acest caz se presupune că funcția returnează un rezultat întreg (de tipul *int*). La o scriere de tipul *main() {}*, compilarea generează un avertisment: pentru că nu este precizat tipul funcției, se aşteaptă ca în corpul funcției să apară o instrucțiune prin care funcția să returneza un rezultat de tip *int* şi de fapt funcția nu returnează nimic. Avertismentul poate fi ignorat.

# Cursul 3

Standardul ANSI defineşte o s**tructură generală a unui program** care, schematizat, arată ca astfel:

1 includere biblioteci
2 definire macrouri
3 declaratii de variabile globale vizibile în toate funcţiile programului
4 definire funcţii utilizator folosite în program
5 declararea prototipurilor funcţiilor care vor fi definite după apelarea lor sau care nu se afla in fisierul sursa curent
6 definirea funcţiei principale main
**7** void **main(**void**){**
8 declaratii de variabile locale pentru funcţia principală main
**9** descrierea în C a algoritmului funcţiei main**}**
10 definire funcţii utilizator folosite în program , declarate anterior prin prototipuri

În secţiunea "includere biblioteci" apar directive cu forma generală ***#include <nume_fisier.h>***. Acestea precizează compilatorului să adauge la textul programului sursă textul din fişierul *nume_fisier.h*. De obicei directivele *#include* sunt utilizate pentru a include în textele de program ale utilizatorilor **fişierele antet (header)** standard (*stdio.h, conio.h, math.h, ...)*.

În secţiunea "definire macrouri" apar directive cu forma generală *#define nume_macro secventa_de_caractere*. O astfel de directivă defineşte un *identificator utilizator* numit ***macrocomandă*** şi o secvenţă de caractere. La întâlnirea unei astfel de directive, toate apariţiile cu numele *nume_macro* din fişierul sursă al programului vor fi înlocuite cu secvenţa de caractere care îi urmează. În exemplul din fig. 3.3, în urma compilării, macrocomenzile *doi, sase* şi *opt* vor primi valorile 2, 6 şi respectiv 8.

```
#define doi 2
#define sase 3*doi
#define opt sase+doi
```
Fig. 3.3

```
#include<stdio.h>
#define SUMA(x,y) ((x)+(y))
void main(void)
{printf("suma numerelor 3+5 este %d\n",SUMA(3,5));
printf("suma numerelor 9+2 este %d\n",SUMA(9,2));}
```
Fig. 3.4

Macrocomenzile sunt cel mai des utilizate pentru atribuirea de valori speciale unor identificatori folosiţi de program (ca de exemplu valoarea lui pi, dimensiunea maximă a tablourilor, etc.). Dacă se doreşte modificarea acestor valori speciale prin folosirea macrocomenzilor, o va face într-un singur loc, anume în directivele *define* respective.

Macromenzile se mai pot utiliza şi ca alternativă pentru definirea unor funcţii simple. Un exemplu de utilizare a unei astfel de macrocomenzi este prezentat în fig. 3.4.

Acelaşi efect l-am fi obţinut şi cu programul din fig. 3.5.

```
#include<stdio.h>
int SUMA(int x,int y)
{return (x+y)}
void main(void)
{printf("suma numerelor 3+5 este %d\n",SUMA(3,5));
printf("suma numerelor 9+2 este %d\n",SUMA(9,2));}
```
Fig. 3.5

**Declararea variabilelor folosite în program**

Variabilele folosite în program se declară în 3 locuri de bază:

a) În interiorul funcţiilor, caz în care se numesc *variabile locale* şi sunt vizibile numai în interiorul funcţiei unde au fost declarate.

b) În *lista de parametri formali* folosiţi la definirea sau declararea funcţiilor.

c) În exteriorul tuturor funcţiilor, caz în care se numesc *variabile globale* şi sunt vizibile în toate funcţiile, inclusiv în *main*. Declararea acestor variabile se face înaintea definirii sau declarării tuturor funcţiilor programului.

## 3.3. Interfaţa cu dispozitivele periferice

O modalitate de realizare a operaţiilor de intrare-ieşire este prin intermediul unor funcţii din *biblioteca standard I/O*. Programele care utilizează această bibliotecă trebuie să conţină *#include<stdio.h>*.

*Intrarea standard*, în mod implicit, este terminalul utilizatorului de la care s-a lansat programul (de obicei tastatura). *Ieşirea standard*, în mod implicit, este tot terminalul utilizatorului, de obicei ecranul.

Pentru a realiza intrări/ieşiri *sub controlul formatelor* se utilizează funcţiile *scanf* (pentru intrare) şi respectiv *printf* (pentru ieşire).

Pentru a realiza intrări/ieşiri de *caractere*, deci *fără controlul unui format* de la intrarea (respectiv ieşirea) standard se folosesc funcţiile *getchar()* şi respectiv *putchar()*, incluse tot de *stdio.h*.

Atât intrarea standard cât şi ieşirea standard pot fi redirectate spre alte dispozitive periferice decât terminalul standard. De exemplu se poate redirecta ieşirea standard spre imprimantă sau disc, ori se poate redirecta intrarea standard spre un fişier de pe disc.

Biblioteca *stdio.h* conţine mai multe funcţii prin care intrările şi ieşirile standard pot fi şi ele precizate prin denumiri specifice. Astfel, fiecare program are o intrare standard şi două ieşiri standard, acestea având denumirile:*stdin* - intrarea standard;*stdout* - ieşirea standard;*stderr* - ieşirea standard pentru erori.

```
#include <stdio.h>
void main(void)
{ printf("primul program in C\n");}
```
Fig. 3.6

## 3.4.Mediul de ieşire

În exemplul din fig. 3.6 se transmite un mesaj către mediul de ieşire.

Funcţia *printf* are rolul de a transmite pe mediul de ieşire argumentele primite la apelul funcţiei, în conformitate cu un format specificat. În acest exemplu singurul argument primit la apel este şirul de caractere *"primul program in C\n"*.

Într-un şir pot apare caractere afişabile şi caractere neafişabile.

Se observă prezenţa *caracterului neafişabil* ***n*** (care apare, ca toate caracterele neafişabile, precedat de "\", adică în forma \n). În C **\n** reprezintă un *caracter special*, anume caracterul NL - new line. Prezenţa sa are ca efect trecerea la o linie nouă.

*Exemplu* Fie programul din fig. 3.7. În acest caz funcţia *printf* are două argumente:

```
#include <stdio.h>
void main(void)
{ int varsta;
  varsta=19;
 printf("varsta mea este:%d\n",varsta);}
```
Fig. 3.7

a) *"varsta mea este:%d\n"* , care conţine un şir de caractere si un *caracter de conversie* (***%d***). Utilizarea (*%d)* conduce la afişarea unui întreg în format zecimal (reprezentat în baza 10).

Orice caracter de conversie (denumit şi *specificator de format*) începe cu %.
b) variabila întreagă *varsta.*

Forma generală a funcţiei *printf* este: *printf(param_control, parametrul_1, parametrul_2,..., parametrul_n)*

Primul parametru, denumit şi "*parametru de control*", trebuie să fie un şir de caractere. Dacă prin *printf* urmărim numai scrierea unui text, parametrul de control nu va conţine nici un specificator de format.

Ceilalţi parametri pot fi numere, variabile, expresii sau chiar alte şiruri de caractere. Dacă *printf* va afişa o valoare sau o variabilă, în primul parametru se introduce *informaţia aferentă tipului valorii sau variabilei*, folosind specificatori de format.

Un specificator de format începe cu %. În continuare specificatorul conţine şi alte caractere, descrise în continuare (primele 3 categorii sunt opţionale):
- un caracter "-" (minus), când se doreşte ca la afişarea datei corespunzătoare să se folosească alinierea la stânga;
- un şir de cifre zecimale, care defineşte numărul minim de caractere folosite la afişarea datei corespunzătoare ;
- un punct, urmat de cifre zecimale, care vor defini precizia cu care se va tipări data;
+ una sau două litere care definesc tipul de conversie aplicat datei care se scrie. Atunci când se folosesc două litere, prima literă va fi *l* (*long*). Aceasta indică faptul că, atunci când a doua literă este *d,o,x* sau *u*, se face o conversie a variabilelor numerice din tipul de date *long*, nu din tipul de date *int*. Cele mai uzuale valori folosite pentru a doua literă sunt:

***d*** - pentru scrierea numerelor de tip *int* cu caractere zecimale (**d**ecimal);
***o*** - pentru scrierea numerelor de tip *int* convertite în baza 8, prin caractere **o**ctale ;
**x** - pentru scrierea numerelelor de tip *int* convertite în baza 16, cu caractere he**x**azecimale;
***u*** - pentru scrierea numerelor ***u****nsigned* cu caractere zecimale (pentru numere întregi pozitive);
***c*** - valoarea parametrului corespunzător se consideră că reprezintă codul ASCII al unui **c**aracter, acesta scriindu-se;
***s*** - pentru scrierea unui şir de caractere (**s**tring);
***f*** - valoarea parametrului corespunzător (de tip real ***f****loat* sau *double*) se scrie în forma *dd...d* **.** *dd..dd* (unde *d* =cifră zecimală);
***e*** - valoarea parametrului corespunzător (de tip real *float* sau *double*) se scrie în formatul *dd...d* **.** ***e****dd..dd* ;
***g*** - se aplică una din conversiile definite de literele *e* sau *f*, alegându-se cea care se reprezintă pe un număr minim de caractere;

Dacă folosim *d,o,x*, data afişată este convertită din tipul *int*, dacă folosim *f,e* data afişată este convertită din tipul *float* sau *double*.
Exemple:
a) ...int valoare=5; printf("%3d\n",valoare) se afişează " 5".
b) ...int valoare=5; printf("%03d\n",valoare): se afişează 005.
c) ...int valoare=9; printf("%#o\ n",valoare): se afişează O11 (în octal, 9 se reprezintă ca 11 ).
d) ...int val=11; printf("%#x\ n",val): se afişează 0xB (în hexazecimal, 11 se reprezintă ca B ).
e) ...float valoare=1.23456; printf("%8.3f\n",valoare): se afişează 1.234, precedat de 3 spaţii, adică în total 8 caractere.
f) ...float valoare=1.23456; printf("%12.3e\n",valoare): se afişează 12 caractere, adică 1.234e+00 precedat de 3 spaţii.
g) ...int valoare=5; printf("%-3d\n",valoare) conduce la afişarea cifrei 5 pe 3 spaţii, aliniat stânga, sub forma "5 ".
h) ...char a[4]="sir";printf("acesta este un %s\n",a): se afişează "acesta este un sir".

```
#include<stdio.h>
void main(void)
{ float val_inchi=0;
printf("introduceti numarul in inchi\n");
scanf("%f",&val_inchi);  /* se intr. nr. de la tastatura */
printf("val. în inchi este= %f\n",val_inchi);
printf("val. în centimetri este= %f\n",2.54*val_inchi);}
```
Fig. 3.8

## 3.5.Mediul de intrare

Programul din fig. 3.8 realizează conversia unei valori exprimate în inchi într-o valoare exprimată în centimetri.

În acest exemplu s-a utilizat funcţia *scanf*, care permite citirea datelor sub controlul formatelor. s*canf* este foarte asemănătoare cu funcţia *printf*, realizând conversii inverse. Literele *d,o,x* şi *u* pot fi precedate de litera *l* şi în acest caz conversia se realizează spre *long*, nu spre *int*. Specificatorul de format *f* poate fi precedat de litera *l* pentru a face conversii spre formatul *double*.

Formatul general al funcţiei este: *scanf(param_control, parametrul_1, parametrul_2,..., parametrul_n)*

Fiecare *parametrul_i* reprezintă adresa unei zone de memorie unde se va memora data citită. Un astfel de parametru este de forma ***&****nume_variabila*. A doua literă a specificatorului de format poate fi: *d,o,x,u,c,s,f.*

```
#include<iostream.h>
void main(){
int a;
cin>>a:
cout<<a;}
```
Fig. 3.9

## 3.6. Funcţii de intrare/ieşire furnizate în C++ de biblioteca *iostream.h*

În C++ citirile/scrierile se mai pot face şi cu ajutorul unor funcţii speciale, furnizate de biblioteca *iostream.h*, pentru care există o modalitate specială de apel. Pentru citire se foloseşte *cin>>*, iar pentru scriere se foloseşte *cout<<*, un exemplu de utilizare fiind prezentat în fig. 3.9.

Funcţia *cin* are sintaxa: $cin>>a_1>>a_2>>...>>a_n$. $a_1,a_2,..,a_n$ sunt variabile de un tip oarecare.

Valorile corespunzătoare fiecărei variabile care se citeşte se introduc pe rând, apăsând tasta *Enter* după fiecare valoare.

*Observaţie* Dacă pentru o anumită variabilă se introduce o valoare care nu coincide cu tipul ei (de exemplu se introduce o

valoare reală pentru o variabilă întreagă), citirea se blochează. Acest lucru semnifică faptul că următoarele variabile nu mai sunt citite, iar calculatorul execută instrucțiunea următoare, *fără a semnaliza această blocare*.

```
#include<iostream.h>
void main(){
int nrlat;
cout<<"introduceti numar de laturi";
cin>>nrlat:
cout<<"numarul de laturi este "<<nrlat;}
```

Fig. 3.10

Funcția *cout* are sintaxa: $cout<<a_1<<a_2<<...<<a_n$ unde $a_1,a_2,..,a_n$ sunt variabile de un tip oarecare.

Constantele de tip şir de caractere vor fi scrise între apostroafe (de exemplu "un şir"). În mod normal datele se scriu una după alta, fără a lăsa spațiu între ele. Dacă însă se doreşte separarea datelor la scriere prin mai multe spații, aceste spații se tipăresc separat, ca în exemplul următor: *cout<<a<<" "<<b*.

Dacă vrem ca după scrierea lui $a_i$ , scrierea lui $a_{i+1}$ să se facă pe rândul următor, se foloseşte constanta *endl*, adică se foloseşte o scriere de forma:

$cout<<a_1<<a_2<<... a_i<<endl<< a_{i+1} <<...<<a_n.$

În exemplul din fig. 3.10 se citeşte de la tastatură şi apoi se afişează variabila *nrlat*.

## 3.7. Vocabularul limbajului

Vocabularul oricărui limbaj de programare este format din: setul de caractere, identificatori, separatori şi comentarii.

*Setul de caractere al limbajului C* reprezintă ansamblul de caractere cu ajutorul cărora se poate realiza un program în acest limbaj. Este alcătuit din:

- **literele** mici şi mari ale alfabetului englez (a...z, A...Z), precum şi **cifrele** din baza 10 (0...9);
- **caractere speciale**: +,-,*,/,=,^,<,>,(,),[,],{,} ,", ' , . , , , ; , : , # , $ , @ , _ , ,%,&,?.

***Un identificator*** reprezintă o succesiune de litere, cifre sau caracterul **_** (adică liniuța de subliniere). Primul caracter nu poate fi o cifră. Cu ajutorul identificatorilor se asociază nume constantelor, variabilelor, funcțiilor, etc.

Exemple de identificatori corecți: *a1*, *tasta*, *un_număr*. Exemple de identificatori eronați: *2ae* (nu poate începe cu o cifră), *mt&* (ultimul caracter nu este nici cifră, nici literă, nici _), *doua cuvinte* (nu poate conține spațiu), *a*b* (interpretează a X b).

Dacă un identificator este format din două sau mai multe cuvinte, pentru o mai bună lizibilitate se pot folosi literele mari pentru prima literă din fiecare cuvânt component (ex. *SumaPozitivelor*) sau liniuța de subliniere (ex. *Suma_Pozitivelor*).

*Observație* Numele care încep de obicei cu litera de subliniere (_) sunt folosite pentru facilități speciale în contextul execuției, a.î. nu este recomandabil să se folosească astfel de nume în programele de aplicație.

O categorie specială de identificatori este dată de *cuvintele cheie* ale limbajului. Acestea au un înțeles bine definit şi nu pot fi folosite în alt context. Exemple de cuvinte cheie: nume de instrucțiuni (*if, while, ...*), nume de tipuri de date (*int, float,...*), etc.

*Literalii* reprezintă valori concrete corespunzătoare unor tipuri de date cum ar fi: valori întregi sau reale (numite şi *numere*), caracterele ASCII, şirurile de date, valorile logice, etc.

Cele mai simple elemente alcătuite din caractere cu semnificație lingvistică se numesc *unități lexicale*. Acestea se separă între ele, după caz, prin unul sau mai multe blanc-uri (spații libere), caracterul "sfârşit de linie" (introdus de tasta *Enter*), tab sau caracterul ";". Se pot considera separatori şi caracterele din fig. 3.11.

parantezele () - delimitează lista argumentelor unei funcții sau părți ale unei expresii;
parantezele {} - delimitează corpul unei instrucțiuni compuse sau corpul unei funcții;
parantezele [] - delimitează dimensiunea unui tablou sau indicii elementelor tabloului;
ghilimelele "" - delimitează un literal şir de caractere;
apostroafele ' ' - delimitează un literal de tip caracter;
perechea "/* */" - delimitează un comentariu

Fig. 3.11

Pentru ca un program să fie uşor de înțeles, se folosesc *comentariile*, care se pot plasa oriunde în program.

Un comentariu care ocupă mai multe linii trebuie încadrat de perechile de caractere /* ...*/. Un comentariu scris pe o singură linie se poate marca prin plasarea a două linii oblice // înaintea sa. În fig. 3.12 sunt prezentate exemple de utilizare a comentariilor.

```
/* comentariu
   pe două linii */
S=0; //initializare suma.
```

Fig. 3.12

## 3.8. Tipuri de date

Toate variabilele trebuie declarate înaintea utilizării lor. Fiecare identificator dintr-un program C are asociat *un tip*. Prin **tip de date** se înțelege: (a) o mulțime de valori; (b) o regulă de codificare a valorilor (modul în care se reprezintă în memorie); (c) o mulțime de operații definite pe mulțimea valorilor.

Se spune că variabilele au *tip standard* dacă acesta este cunoscut de către limbaj fără a fi definit în cadrul programului. Programatorul poate însă şi să îşi definească propriile tipuri, pentru ca apoi să declare variabile cu tipurile definite de el.

```
#include<iostream.h>
void main(){
char a;
int b;
a='y';b=71;
cout<<a<<" "<<a+b;}
```

Fig. 3.13

În C există **5 tipuri de date de bază**: *character, integer, floating-point, double-floating-point* şi *valueless*, cărora le corespund cuvintele cheie: *char, int, float, double* şi *void*.

*Observație* În C nu există tipul boolean, prezentat în pseudocod. În schimb se foloseşte convenția:
- orice valoare diferită de zero, a oricărui tip de date, este considerată ca fiind TRUE;
- orice valoare egală cu zero, a oricărui tip de date, este considerată ca fiind FALSE.

*Observație* În C tipul *character* este asimilat tipurilor întregi.

Un *bit* (***B**inary Dig**it*** - cifră binară) reprezintă cea mai mică unitate de informație pe care o poate manipula calculatorul. Un bit poate lua doar valorile 0 şi 1. Un *octet* (*byte*) reprezintă o unitate de memorare care reține 8 biți.

Considerăm exemplul din fig. 3.13. În acest exemplu operația *a+b* are sens pentru că se adună codul ASCII aferent

valorii variabilei *a* cu valoarea întreagă *b*. Conform codului ASCII, fiecare caracter se memorează pe câte un octet. În consecință rezultatul afişat în urma executării secvenței de cod din exemplu este 192, deoarece codul ASCII corespunzător literere i y este 121.

Caracteristicile tipurilor de bază sunt descrise în tabelul din fig. 3.14, cu observația că variabilele de tip *float* şi *double* pot lua şi valori negative, la rubrica "rang" precizându-se valorile pe care le poate modulul lor.

| Tipul | Nr. biți | Rang |
|---|---|---|
| *char* | 8 | -128÷127, adică ($-2^7 \div 2^7$-1) |
| *int* | 32 | -2147483648÷2147483647 |
| *float* | 32 | 0, (3.4E-38÷3.4E+38) |
| *double* | 64 | 0, 1.7E-308÷1.7E+308 |
| *void* | 0 | fără valoare |

Fig. 3.14

Exceptând tipul *void*, tipurile de bază pot avea anumiți *modificatori* care le preced. Tipurile *char* şi *int* pot fi prefixate cu *signed* şi *unsigned*, tipul *int* poate fi prefixat cu *short* şi *long*, iar *double* poate fi prefixat cu *long*.

Tipurile *char*, *short int*, *int*, *long int* sunt folosite pentru a reprezenta numere întregi de diferite dimensiuni, tipurile *float*, *double* şi *long double* pentru a reprezenta numere reale, iar tipurile *unsigned char*, *unsigned short int*, *unsigned int*, *unsigned long int* pentru a reprezenta numere întregi, fără semn, valori logice, vectori de biți, etc. De exemplu o variabilă de tipul *int* poate lua valori în plaja de valori precizată în fig. 3.14 (limita stânga fiind egală cu $-2^{31}$, iar cea din dreapta fiind egală cu $2^{31}$-1, numărul total de valori din plajă fiind egal cu $2^{32}$), pe când o variabilă de tip *unsigned int* va lua doar valori pozitive, plecând din 0 până la valoarea 4294967295 (adică $2^{32}$). Modificatorul *short* are ca efect înjumătățirea numărului de biți pe care se reprezintă variabila. De exemplu domeniul în care va lua valori o variabilă de tip *short int* va fi ($-2^{15} \div 2^{15}-1$), adică (-32768 ÷ 32767), deci numărul total de valori posibile este egal cu $2^{16}$. Din contră, modificatorul *long* are ca efect dublarea numărului de biți pe care se reprezintă variabila.Astfel, domeniul în care va lua valori o variabilă de tip *long int* va fi ($-2^{63} \div 2^{63}-1$), adică valorile extreme ale intervalului sunt în modul aproximativ egale cu $9.2233372036854776 \times 10^{18}$, deci există un număr total de valori posibile egal cu $2^{64}$.

```
char a='y',b=75,c;
int d,e=178;
```
Fig. 3.15

```
char a=99,b='c';
cout<<a<<" "<<b;
```
Fig. 3.16

Se poate renunța la sufixul *int* de la combinațiile de mai multe nume. Astfel *long* semnifică *long int*, iar *unsigned* semnifică *unsigned int*. În general, atunci când lipseşte un tip într-o declarație, se consideră că este de tipul *int*. Se fac observațiile:

1. La declarare este permisă şi inițializarea variabilelor, ca în exemplul din fig. 3.15.
2. La tipărirea unei variabile *de tip caracter*, se tipăreşte caracterul care îi corespunde şi nu codul ASCII al acestuia. Astfel, în exemplul din fig. 3.16 se tipăreşte de două ori caracterul 'c'.

## 3.9. Constante

Constantele se referă la valori fixe, nemodificabile prin program. Se definesc tipurile:

1) ***Constante întregi***, care se clasifică în 3 categorii, după cum urmează:

a) *Constante zecimale* (numere întregi scrise în baza 10). Exemple: 23 sau 541, etc.

b) *Constante octale* (numere întregi scrise în baza 8). Când se declară o astfel de constantă, ea trebuie precedată de o cifră zero (0) nesemnificativă. De exemplu folosind *0123* se declară constanta octală cu valoarea $(123)_8$.

c) *Constante hexazecimale* (numere întregi scrise în baza 16). Când se declară o astfel de constantă, ea trebuie precedată de 0x sau 0X. De exemplu folosind *0x1A2* se declară constanta hexazecimală $(1A2)_{16}$.

O constantă întreagă poate valori ca şi tipul *unsigned long int* şi *este totdeauna pozitivă*. Dacă se foloseşte semnul - (de exemplu -123), este vorba despre o *expresie constantă*, care este evaluată (în acest caz semnul - este tratat ca şi operator unar).

2) ***Constante caracter***. O astfel de constantă este delimitată cu ajutorul apostroafelor şi este memorată utilizând tipul *char* şi codul ASCII. De exemplu '1' se memorează sub forma $(49)_{10}$. Constantele caracter pot fi folosite în calcule exact ca şi întregii.

O constantă caracter mai poate fi declarată şi sub forma unei *secvențe escape*. O *secvență escape* începe prin caracterul \.

*Exemplu* Caracterul 'a' are codul ASCII $(97)_{10}$. Dar $(97)_{10}=(141)_8=(61)_{16}$. Programul din fig. 3.17 tipăreşte de 4 ori caracterul *'a'*.

```
#include<iostream.h>
void main() {
char x1=97, x2='\141', x3='\x61', x4='a';
cout<<x1<<x2<<x3<<x4;}
```
Fig. 3.17

**Secvențele escape** sunt utile în cazul caracterelor care nu se pot declara în mod obişnuit (între două apostroafe) pentru că nu se pot tasta. De exemplu caracterul *newline* (salt la linie nouă) are codul ASCII egal cu 10 şi poate fi declarat '\10' , '\n' sau '\xa'. Exemple uzuale de secvențe escape: \b - caracte-rul backspace , \0 - caracterul NULL,\t – caracterul horizontal TAB, etc.

Un rol aparte este deținut de *caracterele albe* (whitespaces): blanc, \t, \v, \n sau \r, care au un rol special pentru operațiile de citire/scriere. De exemplu tab-ul orizontal poate fi folosit la operațiile de scriere, el determinând saltul cursorului cu 8 poziții.

```
#include<istream.h>
void main(){
char a='\t',b='i';
cout<<a<<b;}
```
Fig. 3.18

De exemplu programul din fig. 3.18 tipăreşte litera i precedată de 8 spații.

3. ***Constante reale*** Ex: -45.6, 1. (reține 1.0), .2 (memorează 0.2), 0.3, -2.5E-12 (memorează $-2{,}5 \times 10^{-12}$).

4. ***Constante şir de caractere*** Acestea se folosesc pentru reprezentarea şirurilor de caractere. O astfel de constantă se declară prin încadrare între caractere ". Exemplu: *"acesta este un text"*.

```
const int numar=10;
const float pi2=3.14;
```
Fig. 3.19

Constantele şir de caractere, spre deosebire de alte tipuri de constante, au o locație în memoria calculatorului. Pentru a da nume constantelor folosim cuvântul *const*. Forma generală a unei astfel de declarații este: *const [tip] nume_constantă=valoare*. În fig. 3.19 sunt prezentate două exemple.

Constantele întregi mai pot căpăta nume şi prin mecanismul care are la bază *tipul enum*. În tabelul următor sunt prezentate câteva exemple de utilizare a acestui mecanism.

| Instrucțiune | Efect | Explicații |
|---|---|---|

| | | |
|---|---|---|
| *enum timp(ieri,azi,maine)* | *ieri* = 0, *azi* = 1,*maine*=2 | Prima constantă ia implicit valoarea 0, celelalte iau valori consecutive |
| *enum timp(ieri,azi=3,maine=7)* | *ieri* = 0, *azi* = 3,*maine*=7 | Ultimele două inițializări sunt explicite |
| *enum timp(ieri,azi=3,maine=azi)* | *ieri* = 0, *azi* = 3,*maine*=3 | *azi* şi *maine* sunt inițializate cu aceeaşi valoare, adică *3*. |
| *enum timp(ieri=7, azi)* | *ieri=7, azi=8* | Când o constantă este inițializată, celelalte sunt inițializate *implicit* pornind de la valoarea sa, la care se adaugă, pe rând, câte o unitate. |

# Cursul 4

## 3.10. Expresii

### 3.10.1. Generalităţi

Expresiile se construiesc folosind operatori, constante, variabile şi funcţii. Operanzii din componenţa unei expresii pot avea tipuri diferite. De exemplu doi operanzi legaţi de un operator binar pot fi de tipuri diferite. În acest caz compilatorul converteşte tipurile diferite la acelaşi tip, mai exact la tipul operandului mai mare, aplicând *regula conversiilor implicite*.

Se aplică următoarele 8 conversii, *în ordine*, de la prima până la ultima:

2. **A** *long double*, **B** *long double*.
3. **A** *double*, **B** *double*.
4. **A** *float*, **B** *float*.
5. **A** *unsigned long*, **B** *unsigned long*.
6. **A** *long*, **B** *long*.
7. **A** *unsigned*, **B** *unsigned*.
8. În celelalte cazuri, compilatorul tratează ambii operanzi ca fiind de tipul *int*.

Fig. 3.20

1. Exceptând valorile de tip *unsigned short*, toate valorile întregi mici sunt convertite (promovate) la *int*. Tipul *unsigned short* este promovat la *unsigned int*.

Pentru a descrie următoarele 7 conversii (fig. 3.20), vom folosi următoarea schemă de scriere prescurtată: notăm prin **A** exprimarea : "*Dacă unul dintre operanzi este de tipul*", şi prin **B** exprimarea : "*compilatorul promovează pe celălalt la tipul*".

După execuţia acestor paşi, cei doi operanzi sunt de acelaşi tip, iar rezultatul va fi de tipul comun lor. Se observă o regulă: se promovează tipul celei mai mici valori.

*Exemplu* Se consideră programul reprodus în fig. 3.21. Acest program, datorită conversiei de la *char* la *int* tipăreşte *97* şi nu caracterul *'a'*.

```
include#<iostream.h>
void main(void){
int a=0; char b='a';
cout<<a+b;}
```

Fig. 3.21

La evaluarea unei expresii se ţine cont de *asociativitatea operatorilor*, care este de două feluri: de la stânga la dreapta (o vom nota prin -->) şi respectiv de la dreapta la stânga (<--).

În fig. 3.22 se prezintă un exemplu pentru modul în care sunt convertiţi operanzi de tipuri diferite care participă într-o expresie formată din operanzi legaţi prin operatori aritmetici. În exemplu se folosesc declaraţiile: *char ch; int i; float f; double d, rezultat;*

Pentru a înţelege noţiunea de *asociativitate*, pornim de la o expresie în care operanzii

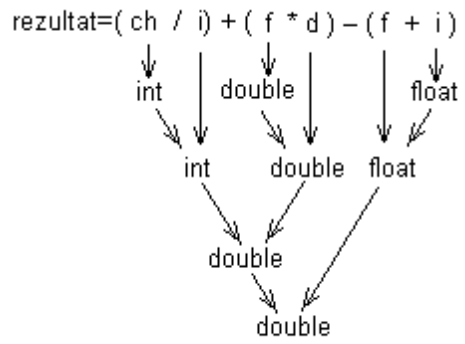



Fig. 3.22

sunt legaţi prin operatori de aceeaşi prioritate. Dacă asociativitatea operatorilor este -->, prima operaţie care se efectuează este cea corespunzătoare primului operator din stânga, a doua este cea corespunzătoare celui de-al doilea operator din stânga, ş.a.m.d. Dacă asociativitatea operatorilor este <--, prima operaţie care se efectuează este cea corespunzătoare primului operator din dreapta, ş.a.m.d.

Există câţiva operatori binari (din construcţii de forma *O1* OB *O2*) pentru care nu se garantează ordinea de evaluare a expresiilor care îi leagă. Aici apare o problemă *nedecelabilă*, adică nu se poate spune cine se evaluează întâi: *O1* sau *O2*. Dacă se scriu expresii de acest tip, rezultatul evaluării expresiei depinde de tipul compilatorului utilizat, caz în care se obţin *programe neportabile*.

### 3.10.2. Operatori C++

#### 3.10.2.1. Operatori aritmetici

Există două categorii de operatori aritmetici: a) Operatori unari: + şi -; b) Operatori binari: +, -, *, /, %.

*Observaţii*

1. Operatorul / acţionează în mod diferit în funcţie de operanzii pe care îi leagă, după cum urmează:
a) Dacă ambii sunt de tip întreg, rezultatul este de tip întreg şi are semnificaţia de împărţire întreagă (se obţine câtul).
b) Dacă cel puţin un operand este de unul din tipurile reale, rezultatul este real (se efectează împărţirea obişnuită).
2. Operatorul % se aplică doar la operanzi de tip întreg. Rezultatul evaluării expresiei *O1 % O2* este restul împărţirii lui *O1* la *O2*.
3. Rezultatul evaluării unei expresii de forma *O1 / O2* sau *O1 % O2* este matematic corect numai dacă *O1* şi *O2* sunt pozitive.

*Exemplul 1* Fie variabila *a* de tip *int* cu valoarea 10. Atunci expresia *4*a/3* este de tipul *int* şi la evaluare se obţine [4*a/3]=13. (Prin paranteze drepte s-a simbolizat partea întreagă). În aceleaşi condiţii, expresia *4*(a/3)* are ca rezultat valoarea 12.
*Exemplul 2* Fie declaraţia *float a*=10; Atunci expresiile *4*a/3* şi respectiv *4*(a/3)* au ca rezultat valoarea de tip *float* 13.3333...
*Exemplul 3* Fie declaraţia *int a=-10*. Atunci, evaluând expresia *a*-3* obţinem 30.
*Exemplul 4*. Fie declaraţiile: *int a=10; char b=2; float c=5*.; Expresia *a+b+c* are ca rezultat valoarea de tip float *17.000000*.

#### 3.10.2.2. Operatori relaţionali

În C există următorii operatori relaţionali: <, <=, >, >=. Întrucât în C nu există valorile logice TRUE şi FALSE din pseudocod, *rezultatul evaluării unei expresii este 1 dacă valoarea de adevăr a expresiei este TRUE şi 0 dacă este FALSE*.
*Exemplu* La evaluarea expresiei *3+7>=11-1* se obţine 1.

#### 3.10.2.3. Operatori de egalitate

În C se definesc următorii operatori de egalitate: == pentru egalitate, respectiv != pentru inegalitate. Şi în cazul aplicării acestor operatori se utilizează convenţia prezentată anterior: 1 semnifică TRUE şi 0 semnifică FALSE.
*Exemplu* Dacă se evaluează 3!=4 se obţine 1.

#### 3.10.2.4. Operatori de incrementare şi decrementare

Operatorii de incrementare şi decrementare sunt operatori *unari* care acţionează asupra conţinutului unei *variabile* şi au rolul de a incrementa (caz în care se adună 1) sau decrementa (caz în care se scade 1). Se foloseşte ++ pentru incrementare,

respectiv -- pentru decrementare.

Operatorii pot fi:

- prefixați - dacă sunt aplicați în fața operandului (exemplu ++*a*, --*a*)
- postfixați - dacă sunt aplicați după operand (exemplu *a*++, *a*--).

Dacă operandul este prefixat, variabila este incrementată (decrementată) *înainte* ca valoarea reținută de ea să intre în calcul. În cazul postfixării, variabila este incrementată (decrementată) *după ce* valoarea reținută de ea a intrat în calcul.

În tabelul din fig. 3.23 se prezintă exemple de aplicare a operatorilor de incrementare și decrementare, considerând că variabilele *a* și *b* care apar în expresiile prezentate ca exemple sunt de tip întreg și au valorile inițiale: *a=1, b=3*. Eroarea care apare la evaluarea expresiei *1+++a* se datorează următorului fapt: compilatorul a considerat că operatorul ++ este aplicat postfixat pentru *constanta* 1 și nu prefixat pentru *variabila a*, iar incrementarea este permisă numai asupra unei variabile.

| Expresie | Rezultat evaluare | *a* și *b* după evaluare |
|---|---|---|
| 1+a++ | 2 | a=2 |
| 1-++a | -1 | a=2 |
| 1+++a | Eroare sintactică | |
| a++*b++ | 3 | a=2; b=4 |
| ++a*++b | 8 | a=2; b=4 |

Fig. 3.23

În cazul utilizării formei prefixate, apar cazuri în care nu se poate decide care este valoarea produsă de expresie la evaluare. La operatorii postfixați e posibil ca incrementarea să se facă în oricare dintre momentele următoare:
*momentul 1* (îl vom nota cu M1)*:* imediat ce valoarea a intrat în calcul *sau*
*momentul 2* (îl vom nota cu M2)*:* după ce a fost calculată întreaga expresie *sau*
*momentul 3* (îl vom nota cu M3): între momentele M1 și M2.

Specificațiile C++ nu prevăd cum se procedează în astfel de cazuri, iar momentul în care are loc incrementarea poate conduce la generarea de rezultate diferite.

### 3.10.2.5. Operatori logici

Există 3 operatori logici: ! semnifică negarea logică, || se utilizează pentru operația SAU logic și && semnifică ȘI logic. Aceşti operatori se pot aplica oricărei variabile sau constante.
*Operatorul !* acționează astfel: dacă operandul este o valoare diferită de zero, rezultatul este 0. Altfel rezultatul este 1.
*Operatorul* || funcționează așa cum s-a descris în pseudocod: numai dacă ambii operanzi sunt 0 se obține 0, altfel se obține 1.
*Operatorul &&* funcționează așa cum s-a descris în pseudocod: doar când ambii operanzi sunt diferiți de 0 se obține 1, altfel se obține 0.
În tabelul din fig. 3.24 sunt reproduse câteva exemple de evaluare a unor expresii în care intervin operatori logici.

| Operanzi | Expresie | Valoare expresie |
|---|---|---|
| *float a=2* | *!a* | *0* |
| *int a=1, b=3* | *a&&b* | *1* |
| *int a=0, b=3* | *a&&b* | *0* |
| *int a=0, b=3* | *a\|\|b* | *1* |
| *int a=0, b=0* | *a\|\|b* | *0* |

Fig. 3.24

*Observație* Operatorii logici binari garantează modul în care se tratează operanzii: întâi cel din stânga, apoi, *dacă este cazul*, cel din dreapta.

Astfel, într-o expresie de forma *O1* || *O2*, dacă $O1 \neq O2$, O2 nu mai este evaluat pentru că rezultatul este evident 1. Analog, într-o expresie de forma *O1 && O2*, dacă O1 este 0, nu se mai evaluează O2, pentru că rezultatul este evident 0. De aceea, în cazul unei expresii de forma *a && b--*, dacă *a=0*, nu se mai evaluează *b*, deci nu se mai face decrementarea postfixată.

### 3.10.2.6. Operatori logici pe biți

Operatorii logici pe biți acționează numai asupra operanzilor de tip întreg. Există 4 tipuri de astfel de operatori:

a) Operatorul << apare în expresii de forma *O1 << O2*. Aplicarea acestui operator are ca efect deplasarea către stânga a tuturor biților operandului *O1* cu un număr de poziții egal cu valoarea lui *O2*. Pozițiile lui *O1* rămase libere (în dreapta) vor reține valoarea 0. Dacă *O2* reține o valoare pe care o notăm cu *m*, o astfel de deplasare este echivalentă cu înmulțirea cu $2^m$ (dacă m este mai mic decât numărul de biți pe care este reprezentat *O1*).
b) Operatorul >> apare în expresii de forma *O1 >> O2*. Efectul său este analog cu cel al operatorului <<, dar deplasarea se face la dreapta. Există două cazuri distincte:
- dacă *O1* este de tip întreg fără semn, pozițiile rămase libere (în stânga) se completează cu 0, și o astfel de deplasare este echivalentă cu împărțirea cu $2^m$ ;
- dacă *O1* este de tip întreg cu semn, pozițiile rămase libere în stânga se completează cu valoarea reținută de *bitul de semn* (0 pentru numere pozitive, 1 pentru numere negative).

c) Operatorii *binari pe biți*: &, | și ^.

Aceşti operatori apar în expresii de forma *O1 OB O2*. Rezultatul evaluării unei expresii de forma *E= O1 OB O2* se obține aplicând pentru fiecare pereche de biți aflați pe aceeași poziție regulile prezentate în tabelul de la paragraful 1.4.3.3.
*Observație* Trebuie făcută distincția față de operatorii logici, care folosesc notațiile dublate && și respectiv ||. Operatorii logici au aceleași denumiri, dar ei tratează întregul operator ca pe o singură valoare, adevărată sau falsă.
d) Operatorul unar ~ (negare pe biți). Acest operator are rolul de a inversa conținutul biților (dacă un bit conține valoarea 0, după evaluare el va conține 1 și invers).

Pentru exemplificare să considerăm declarațiile: *int a=0 x 000f, b=0 x 0f03*. Fiecare cifră hexazecimală se poate exprima în baza 2 conform tabelului din fig. 3.25 (unde prin H s-a notat cifra hexazecimală iar prin E s-a notat echivalentul său în baza 2).

| H | E | H | E | H | E | H | E |
|---|---|---|---|---|---|---|---|
| 0 | 0000 | 4 | 0100 | 8 | 1000 | C | 1100 |
| 1 | 0001 | 5 | 0101 | 9 | 1001 | D | 1101 |
| 2 | 0010 | 6 | 0110 | A | 1010 | E | 1110 |
| 3 | 0011 | 7 | 0111 | B | 1011 | F | 1111 |

Fig. 3.25

Valorile hexazecimale ale variabilelor *a* și *b* din exemplul nostru pot fi scrise în baza 2 astfel:
a= 0000 0000 0000 1111; b= 0000 1111 0000 0011.
Atunci :
a & b= 0000 0000 0000 0011=0 X 0003
a | b=0000 1111 0000 1111=0 X 0F0F
a^b=0000 1111 0000 1100 = 0X 0F0C

a<<2=0000 0000 0011 11**00**= 0 X 003C
a>>2= **00**00 0000 0000 0011= 0X 0003
~a=1111 1111 1111 0000= 0 X FFF0.
Exemplul următor realizează un SAU şi un ŞI pentru două caractere:
'a' | 'c'= 0110 0001 | 0110 0011 = 01100011='c'
'a' & 'c'= 0110 0001 & 0110 0011 = 0110 0001='a'

**3.10.2.7. Operatori de atribuire**

Spre deosebire de limbajele de tip pseudocod, *în C atribuirea este operator*. În C există mai mulţi operatori de atribuire.

**1. Operatorul "="** apare într-o expresie de forma : *v=expresie*, iar principiul de execuţie este următorul: se evaluează expresia, apoi variabilei *v* i se atribuie valoarea obţinută (dacă este cazul se face şi conversia de tip necesară).

Se pot face şi ***atribuiri multiple,*** de forma: $v=v_1=v_2=...=v_n=expresie$. In acest caz principiul de execuţie poate fi descris de următorii paşi: (a) se evaluează *expresie*; (b) valoarea obţinută în urma evaluării precedente se atribuie variabilei $v_n$; (c) conţinutul variabilei $v_n$ se atribuie variabilei $v_{n-1}$ ; (d) Se continuă prin atribuiri prin care conţinutul variabilei $v_i$ se atribuie variabilei $vi_{-1..}$.pentru ca în final conţinutul variabilei $v_1$ să i se atribuie variabilei *v*. La fiecare pas se fac conversiile necesare.

**2. Operatorii de atribuire compuşi**

Operatorii de atribuire compuşi apar în construcţii sintactice de forma: ***v op= expresie.*** Această construcţie are acelaşi efect ca şi scrierea *v=v op expresie.* Se pot forma următorii operatori de atribuire compuşi: +=, -=, *=, /=, %=, <<=, >>=, &=, |=, ^=.

**Exemplul 1** *int a=3; a*=2* va produce rezultatul 6.

*Observaţie* Dacă unei variabile de tip întreg i se atribuie conţinutul unei variabile de tip real, acesta este trunchiat.

**Exemplul 2***: int a; float b=-1.9;* După ce se execută *a=b* , *a* va reţine -1.

*Observaţie* Trunchierea nu înseamnă "parte întreagă". Dacă lui *a* i s-ar fi atribuit partea întreagă a lui *b*, *a* ar fi trebuit să reţină -2.

**3.10.2.8. Operatorul virgulă**

*int a=1, b=5; float c;*
Expresia
*c=a=b+1, a=c+2, b=b+1*
se evaluează astfel:
a=b+1=6
c=a=6
a=c+2=6+2=8
b=b+1=6.

Fig. 3.26

În C se pot scrie mai multe expresii separate prin virgulă, astfel: $expr_1$, $expr_2$, ..., $expr_n$. Deoarece operatorul "," se asociază "-->", expresiile se evaluează în ordinea $expr_1$, $expr_2$, ..., $expr_n$.

Dacă notăm $E= expr_1, expr_2, ..., expr_n$, prin convenţie, E produce ca rezultat valoarea obţinută în urma evaluării *ultimei expresii*, tipul acestei valori fiind şi tipul lui E. În fig. 3.26 se redă un exemplu de utilizare a operatorului virgulă. Expresia din exemplu, în ansamblul ei, este de tipul *int* si produce valoarea 6.

**3.10.2.9. Operatorul condiţional (sau ternar)**

Operatorul condiţional apare în expresii cu forma generală: *expr1 ? expr2: expr3*
Principiul de execuţie este următorul:
- se evaluează *expr1;*
- dacă în urma evaluării se obţine o valoare diferită de 0 (TRUE), se evaluează *expr2* şi *expr3* este ignorată;
- dacă în urma evaluării se obţine o valoare egală cu 0 (FALSE), se evaluează *expr3* şi *expr2* este ignorată.

În ansamblu, o expresie E de forma E= *expr1 ? expr2: expr3* este de tipul *expr2* sau *expr3* (în funcţie de care dintre ele este evaluată) şi produce în mod corespunzător valoarea expresiei *expr2* sau pe cea a expresiei *expr3*.

```
#include<iostream.h>
void main(void){
float x;
cout<<"x";
cin>>x;
cout<<(x>0?x;-x);}
```

Fig. 3.27

*Exemplu* Programul din fig. 3.27 citeşte o variabilă *x* de tip *float* şi afişează modulul acesteia.

*Observaţie* Ultima linie din program se putea înlocui cu secvenţa de instrucţiuni din fig. 3.28:

```
if (x>0)
cout<<x;
else
cout<<-x;
```

Fig. 3.28

**3.10.2.10. Operatorul sizeof**

Operatorul *sizeof* are rolul de a returna numărul de octeţi utilizaţi pentru memorarea unei valori şi poate fi utilizat într-una dintre următoarele forme: *sizeof(expresie)* sau *sizeof(tip)*.

Expresia *expresie* care apare în prima dintre forme nu este evaluată, deci nu apar efecte secundare.

*Exemplul 1*: Fie declaraţia *float a*. Atunci *size(a)=4* (pentru că *float* se memorează pe 32 biţi, adică 4 octeţi). (Dacă se evaluează *size(float)* se obţine 4).

*Exemplul 2*: *int a; double b; size(a+b)=8*.

**3.10.2.11. Operatorul cu conversie explicită (operatorul cast)**

Atunci când se doreşte ca unul sau mai mulţi operanzi să intre în calcul convertiţi la un *anumit* tip de date, specificat de programator, şi nu aşa cum se realizează în mod implicit, se foloseşte o construcţie sintactică de forma:

*(tip_dorit)operand_convertit*

*Exemplul 1: float x=-1.9;* Atunci *(int)x=-1* (s-a făcut conversia de la *float* la *int* prin trunchiere).

*Exemplul 2: float x=-1.9;* Atunci *(int)(++x+3)=2*.

**3.10.2.11. Operatori utilizaţi în manipularea pointerilor**

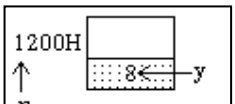
*Un pointer* este în esenţă adresa de memorie a unei variabile de un anumit tip. O *variabilă pointer* este o variabilă pentru care s-a precizat în mod explicit că memorează un pointer. Există 2 operatori pentru manipularea pointerilor:

**1. Operatorul adresă (ampersand), simbolizat prin &**

Când într-un program trebuie să se determine adresa unei varibile, se utilizează *operatorul adresă*. Acesta este un operator unar, ce returnează adresa de memorie a operandului specificat. Considerăm construcţia sintactică: *adresa_de_memorie=&variabila*.


```
int x=100,*p;
p=&x;
```

Fig. 3.29




```
int v,y=8,*p;
v=*p;
```

Fig. 3.30

În această construcție *variabila* reprezintă identificatorul unei variabile de un anumit tip iar *adresa_de_memorie* este o variabilă de tip pointer care, după execuția atribuirii de mai sus, primește ca valoare adresa de memorie a variabilei specificate.

În exemplul din fig. 3.29 se efectuează operațiunile: (a) declarare variabilă *p* de tip pointer, care va memora o adresă de memorie a unei variabile de tip *int*; (b) memorare în *p* a adresei de memorie a varibilei *x*.

În acest exemplu, *x* s-a memorat la adresa 3460H; după execuția *p=&x*, *p* a primit valoarea 3460H.

Deci, pentru a declara o variabilă de tip pointer trebuie precizat tipul valorii pe care o indică pointerul și un asterisc înaintea numelui variabilei de tip pointer. Astfel *float *x* are ca efect declararea variabilei de tip pointer *x* care va memora o adresă a unei variabile de tip real (*va indica spre* o variabilă de tip real).

**2. Operatorul de redirectare (asterisc) simbolizat prin ***

Utilizând adresa pe care o conține un pointer se poate determina informația conținută la adresa indicată de el. *Dereferențierea unui pointer* reprezintă procesul de accesare a informației conținute la adresa indicată de pointer. Operatorul * este tot un operator unar, complementar operatorului &.

Considerăm construcția sintactică următoare: *variabila = *adresa_de_memorie*.

În această construcție *adresa_de_memorie* este o variabilă de tip pointer care conține adresa unei variabile de un anumit tip, iar *variabila* este identificatorul unei variabile de același tip cu tipul variabilei a cărei adresă este memorată în *adresa_de_memorie*.

În urma executării acestei operații de atribuire variabila specificată prin *variabila* va primi valoarea variabilei a cărei adresă de memorie este memorată de *adresa_de_memorie*.

Considerăm exemplul din fig. 3.30. În acest exemplu variabila *y* de tip *int* memorează valoarea 8. Presupunem că sistemul de operare a memorat-o la adresa *1200H*. După executarea *v=*p*, *v* primește valoarea 8.

**3.10.2.12. Operatori utilizați la referirea termenilor individuali ai structurilor și uniunilor**

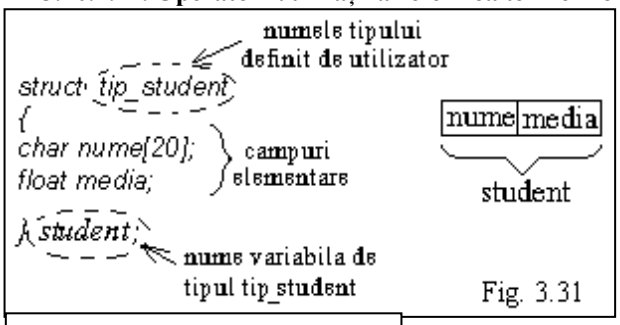



Fig. 3.31

Structurile și uniunile sunt *tipuri colective de date* (*structurate*) care pot fi definite și utilizate sub un singur nume. Pentru referirea termenilor individuali ai acestora se folosesc 2 operatori.

Operatorul "**.**" (punct) se folosește atunci când se lucrează cu o structură sau uniune pentru a ne referi la câmpuri elementare din aceasta. Operatorul "->" (săgeată dreapta) se folosește pentru a ne referi la câmpuri elementare reperate prin intermediul unor pointeri care le indică.

Considerăm exemplul din fig. 3.31. Aici utilizatorul declară un tip colectiv de date, mai exact *o structură*. Aceasta este formată din două câmpuri elementare, *char nume[20]* (vector cu maxim 21 de componente, fiecare având tipul *char*), respectiv *float media*. După declararea structurii se declară și o variabilă de acest tip, identificată prin nume - *student*.

```
struct tip_student *p=&student;
student.media=7.20;
p->media=7.20;
```

Fig. 3.32

Considerăm exemplul din fig. 3.32. Aici se declară și inițializează variabila pointer *p*. Ea va reține adresa unde se memorează variabila *student* care are tipul *tip_student*. Instrucțiunile *student.media=7.20* și respectiv *p->media=7.20* au același efect, respectiv reactualizarea valorii reținute de câmpul *media*, astfel încât în acest câmp să se memoreze valoarea 7.20.

# CAPITOLUL 4. INSTRUCTIUNILE LIMBAJULUI C

## 4.1. Instrucțiunea expresie

C-ul utilizează o instrucțiune care nu există în pseudocod, anume *instrucțiunea expresie*, cu sintaxa: *expresie;*

Dacă expresia este vidă (nu are nici un operand și nici un operator), obținem *instrucțiunea vidă*. Instrucțiunea expresie este des utilizată în atribuiri.

*Observație* Este permisă scrierea unor instrucțiuni expresie ca cea din exemplul următor: *7+2;*. Efectul execuției unor instrucțiuni de acest tip este următorul: se evaluează expresia, dar rezultatul obținut nu este utilizat în nici un fel.

În exemplul din fig. 4.1 se citesc numerele întregi *a* și *b* și se afișează media lor aritmetică.

```
#include <iostream.h>
void main(){
int a,b;
float medie;
cout<<"a=?"; cin>>a;
cout<<"b=?"; cin>>b;
medie=(float)(a+b)/2; // instr. expresie
cout<<"media="<<medie;}
```

Fig. 4.1

## 4.2. Instrucțiunea *if*

Ca și în pseudocod, și în C instrucțiunea *if* prezintă două forme, prezentate în fig. 4.2

```
Forma 1
if(expresie) instrucțiune_1;
        else
instrucțiune_2;
Forma 2
if(expresie) instrucțiune_1;
```

Efectul execuției instrucțiunii *if* reprezentate prin prima formă este următorul: se evaluează *expresie*. Dacă valoarea obținută este diferită de 0, se execută *instrucțiune_1*. În caz contrar (valoarea obținută este 0) se execută *instrucțiune_2*. La a doua formă nu se mai execută nimic atunci când în urma evaluării expresiei se obține o valoare egală cu 0.

*Exemplu* (fig. 4.3). Se citește o valoare întreagă. Dacă aceasta este pară, se afișează mesajul "par". Altfel nu se afișează nimic.

```
#include <iostream.h>
void main(){
int a;
cout<<"a=?"; cin>>a;
if(a%2==0)
cout<<"par";}
```

Fig. 4.3

```
{
i1;
i2;
...
in;
}
```
Fig. 4.4

## 4.3. Instrucțiunea compusă

Pentru a putea scrie mai multe instrucțiuni care să fie interpretate de compilator ca una singură se utilizează *instrucțiunea compusă*, cu forma generală din fig. 4.4, unde *i1, i2, ...in* reprezintă instrucțiunile din care este format corpul instrucțiunii compuse. Un exemplu de utilizare este reprodus în fig. 4.7 (instrucțiunile îngroşate).

## 4.4. Instrucțiunea *switch*

Instrucțiunea *switch* are forma generală din fig. 4.5.

În această formă generală, *expresie* reprezintă o expresie *de tip int*. *expr1, expr2, ..., exprn* reprezintă *expresii constante de tip int*.

```
switch(expresie){
case expr1: secvență_de_instrucțiuni_1; break;
case expr2: secvență_de_instrucțiuni_2; break;
....
case exprn: secvență_de_instrucțiuni_n; break;
[default: secvență_de_instrucțiuni_n+1];
}
```
Fig. 4.5

Principiul de execuție este următorul:
- se evaluează *expresie*;
- dacă în urma evaluării se obține o valoare egală cu cea produsă de oricare dintre celelalte expresii (să considerăm că este egală cu cea produsă de *expri*), atunci se execută, în ordine, instrucțiunile care formează *secvență_de_instrucțiuni_i*. Se iese apoi de sub controlul instrucțiunii *switch*;
- în caz contrar (atunci când în urma evaluării se obține o valoare care nu este egală cu nici una dintre valorile produse de expresiile *expri*), dacă eticheta *default* este prezentă, se execută *secvență_de_instrucțiuni_n+1*.

*Observație* Alternativa *default* este facultativă. În absența ei, în cazul în care în urma evaluării se obține o valoare care nu este egală cu nici una dintre valorile produse de expresiile *expri*, nu se execută nimic.

*Exemplu* Programul reprodus în fig. 4.6 execută următoarele operațiuni: citeşte valoarea variabilei *i*. Dacă aceasta este 1 sau 2, se afişează un mesaj de forma : "am citit ...", altfel se afişează mesajul "am citit altceva".

```
#include <iostream.h>
void main(){
int a;
cout<<"a=?"; cin>>a;
switch(a)
{
case 1:cout<<"am citit 1";break;
case 2:cout<<"am citit 2";break;
default: cout<<"am citit altceva";
}
```
Fig. 4.6

## 4.5. Instrucțiunea *while*

Instrucțiunea *while* reproduce structura WHILE DO prezentată la pseudocod. Are următoarea formă generală: *while(expresie) instrucțiune*.

***Varianta 1***
```
#include <iostream.h>
void main(){
int n,S=0;
cout<<"n=?"; cin>>n;
while(n) {
S=S+n % 10;
n=n / 10;}
```

***Varianta 2***
```
#include <iostream.h>
void
main(){
int n,S=0;
cout<<"n=?"; cin>>n;
while(S+=n % 10, n /= 10);
cout<<"S="<<S;
}
```

Principiul de execuție al acestei instrucțiuni este următorul:
Pasul 1: Se evaluează *expresie*.
Pasul 2: Dacă rezultatul obținut la pasul precedent este diferit de 0 (expresia are valoarea de adevăr TRUE), atunci se execută *instrucțiune* şi apoi se revine la pasul 1. Altfel (s-a obținut 0), se iese de sub controlul instrucțiunii *while*.

*Exemplu* (fig. 4.7) Se citeşte un număr natural *n*. Să se calculeze suma cifrelor sale (de exemplu, dacă numărul citit este 426, se va afişa 12). În pseudocod – fig. 2.7.

## 4.6. Instrucțiunea *do while*

```
do
instrucțiune
while(expresie)
```
Fig. 4.8

Efectul execuției instrucțiunii *do while* este asemănător cu cel prezentat la pseudocod, la structura REPEAT UNTIL. Forma generală a acestei instrucțiuni este reprodusă în fig. 4.8, iar principiul de execuție al acestei instrucțiuni este următorul:
Pasul 1: Se execută *instrucțiune*.
Pasul 2: Se evaluează *expresie*. Dacă se obține valoarea 0, execuția instrucțiunii *do while* se încheie, altfel se reia execuția de la pasul 1.

***Varianta 1***
```
#include <iostream.h>
void main(){
int n,S=0,i=1;
cout<<"n=?"; cin>>n;
do {
S=S+i;
i++;
```

***Varianta 2***
```
#include <iostream.h>
void
main(){
int n,S=0,i=1;
cout<<"n=?"; cin>>n;
do
S+=i++;
```

*Observație* Secvența se execută cel puțin o dată, indiferent de valoarea pe care o produce *expresie*.

*Exemplul 1* (fig. 4.9) Se citeşte un număr natural *n>=1*. Să se calculeze suma primelor *n* numere naturale.

*Exemplul 2* (fig. 4.10) Dându-se o valoare reală *x* să se calculeze suma *S* a următoarei serii: $S = \frac{x}{1!} - \frac{x}{3!} + \frac{x}{5!} - \ldots$

Se doreşte calculul cu o precizie relativă *eps* dată de expresia: $\left| \frac{x}{(2i-1)!} \right| < eps$

*Observație* Pentru uşurarea calculului un termen *t* al seriei se poate calcula aplicând următoarea relație de recurență: $t = -t * \frac{1}{i(i-1)}$

```
#include<iostream.h>
#include<math.h>
void main(void){
float x,S,t,eps;
int i;
cout<<"introduceti x";cin>>x;
cout<<"introduceti eps";cin>>eps;
S=x;t=x;i=1;
do {
i=i+2;
t=-t / i / (i-1);
S += t; }
while( abs(t) >= eps);
cout<<"S="<<S;}
```

Fig. 4.10

## 4.7. Instrucțiunea *for*

Instrucțiunea *for* are următoarea formă generală:

*for( expr_inițializare ; expr_test ; expr_incrementare ) instrucțiune*

Expresia *expr_inițializare* se foloseşte de regulă pentru inițializarea *variabilei de ciclare*. Expresia *expr_test* se foloseşte pentru a testa dacă se va executa *instrucțiune*. Doar dacă *expr_test* produce o valoarea diferită de 0 se va executa *instrucțiune*. Expresia *expr_incrementare* se foloseşte de obicei pentru incrementarea variabilei de ciclare.

Principiul de execuție al acestei instrucțiuni este următorul:

Pasul 1: Se evaluează *expr_inițializare*.

Pasul 2: Se evaluează *expr_test*. Dacă se obține o valoare diferită de 0, se execută *instrucțiune*, apoi se trece la pasul 3. În caz contrar se iese de sub controlul instrucțiunii *for*.

Pasul 3: Se evaluează *expr_incrementare* şi se revine la pasul 2.

*Observație* Oricare dintre cele 3 expresii din forma generală a lui *for* poate fi vidă.

În exemplul din fig. 4.11 se citeşte un număr natural *n*. Se afişează produsul numerelor naturale mai mici decât *n*.

```
#include <iostream.h>
void main(){

unsigned long produs=1;
int i,n;
cout<<"n=?"; cin>>n;
for(i=1; i<=n; i++)
produs=produs*i;
cout<<"produs="<<produs;}
```

**Probleme particulare relative la utilizarea instrucțiunii *for***

Nu totdeauna sunt necesare toate cele 3 expresii care apar în forma generală a instrucțiunii *for*.

De exemplu, dacă *variabila de ciclare* a fost inițializată înainte de intrarea în ciclu, expresia de inițializare nu mai este necesară, în acest caz folosindu-se următoarea formă particulară: *for( ; expr_test ; expr_incrementare) instrucțiune* .

În fig. 4.12 (a), cu ajutorul instrucțiunii *for* se afişează numerele naturale de la 0 la 9999. Variabila *contor* s-a inițializat înainte de intrarea în ciclu cu valoarea 0 .

```
for( ; contor<1000; contor++)
printf("%d",contor);
```

(a)

```
for( ; contor<1000 ; )
    printf("%d",contor++);
```

(b)

In fig. 4.12 (b) nu se foloseşte nici *expr_incrementare*, acest lucru realizându-se în corpul ciclului prin operatorul de postincrementare, iar instrucțiunea *for(;;)* va produce o ciclare infinită.

Instrucțiunea *for* acceptă ca variabila contor să aibă şi un tip de date diferit de *int*. In exemplul din fig. 4.13 se utilizează 3 instrucțiuni *for*: prima afişează literele mari de la A la Z, a doua afişează literele mici în ordine inversă, de la z la a şi ultima afişează numerele 0,0.1, 0.2,...,0.9.

```
#include<stdio.h>
void main(void){
char litera;
float procent;
for(litera='A';litera<='Z';litera++)
putchar(litera);
for(litera='z';litera<='a';litera--)
putchar(litera);
putchar(' \n');
for(procent=0; procent <1;procent+=0.1)
printf("%3.1f\n", procent);}
```

Fig. 4.13

## 4.8. Instrucțiuni de salt

În C există 4 instrucțiuni de salt necondiționat, care determină continuarea execuției programului din altă parte, alta decât cea corespunzătoare secvenței care ar fi urmat în derularea normală a execuției. Aceste 4 instrucțiuni sunt descrise în cele ce urmează. În plus, un salt necondiționat se mai poate realiza şi cu funcția *exit*.

### 4.8.1. Instrucțiunea *break*

Instrucțiunea *break* are două utilizări. Prima constă în terminarea unui *case* în cadrul unei instrucțiuni *switch*. A doua utilizare constă în terminarea imediată a unui ciclu, fără a se mai ține cont de testul condițional normal al ciclului. Dacă într-un ciclu se întâlneşte o instrucțiune *break*, controlul programului se transferă imediat la prima instrucțiune de după ciclu.

În fig. 4.14 se prezintă un program prin care se afişează numerele naturale de la 0 până la 10. În absența lui *break* s-ar fi afişat şi numerele de la 10 la 99.

```
#include<iostream.h>
void main(void){
int i;
for(i=0;i<100;i++){
cout<<"i"<<i<<" ";
if(i==10) break;}
}
```

Fig. 4.14

### 4.8.2. Instrucțiunea *continue*

Dacă în interiorul unui ciclu se execută instrucțiunea *continue*, acest lucru are ca efect oprirea iterației curente şi trecerea imediată la iterația următoare.

În exemplul din fig. 4.15, datorită utilizării instrucțiunii *continue* nu se afişează toate numerele de la 0 la 99, aşa cum s-ar derula execuția programului în mod obişnuit, ci se afişează numai numerele pare. De exemplu la iterația aferentă valorii *i=3* , la execuția instrucțiunii *continue* se abandonează celelalte instrucțiuni care trebuiau efectuate pentru *i=3* (adică nu se mai execută *cout*) şi se trece la execuția grupului de instrucțiuni aferente iterației *i=4*.

```
#include<iostream.h>
void main(void){
int i;
for(i=0;i<100;i++){
if( i % 2) continue;
cout<<i:}
}
```

Fig. 4.15

În cazul instrucțiunilor *while* şi *do while*, o instrucțiune *continue* determină trecerea direct la testul condițional. În cazul instrucțiunii *for* se realizează mai întâi operația de incrementare a variabilei de control a ciclului, apoi se testează condiția de continuare a ciclului.

```
#include<iostream.h>
void main(void){
      int i,n,S;
      cout<<"n";cin>>n;
      i=1;S=0;
etich: S+=i;i++;
      if(i<=n) goto etich;
      cout<<S; }
```
Fig. 4.16

#### 4.8.3. Instrucţiunea de salt *goto*

Instrucţiunea de salt *goto* are următoarea sintaxă: *goto etichetă;*. *etichetă* serveşte la localizarea unei alte instrucţiuni. Efectul execuţiei este următorul: instrucţiunea următoare care se va executa este cea etichetată cu ajutorul *etichetă*. În programarea structurată nu există situaţii unde folosirea instrucţiunii *goto* este imperios necesară. În exemplul din fig. 4.16 se calculează suma primelor *n* numere naturale.

#### 4.8.4. Instrucţiunea *return*

Instrucţiunea *return* este folosită în definirea unei funcţii şi poate preciza valoarea pe care o "întoarce" funcţia. Un exemplu de utilizare este dat în fig. 4.17, unde se defineşte funcţia *f*, care întoarce (returnează) pătratul variabilei furnizate ca parametru. Apoi, în programul principal, se citesc valorile variabilelor *x* şi *y* şi se afişează suma pătratelor lor, apelând funcţia *f*.

```
#include<iostream.h>
int f(int x){
return(x*x);}
void main(void){
int x,y;
cout<<"x";cin>>x;
cout<<"y";cin>>y;
cout<<f(x)+f(y);}
```
Fig. 4.17

# CAPITOLUL 5. FUNCTII

## 5.1. Generalităţi

În limbajul C o funcţie este o structură de program, care descrie un subalgoritm al unui algoritm şi poate fi apelată de mai multe ori pe parcursul execuţiei unui program. O funcţie poate utiliza unul sau mai mulţi *parametri de intrare* numiţi si *argumente* (sunt însă cazuri când nu se foloseşte nici un argument). Funcţia poate *returna*, în urma apelării şi executării sale *cel mult o valoare de ieşire* (sau *rezultat*) numit *valoarea funcţiei*. Dar există şi funcţii care nu returnează nimic. În practică, de multe ori, valorile returnate de funcţie sunt ignorate.

Orice funcţie care se utilizează în program trebuie *descrisă* sau *declarată* totdeauna înaintea apelării sale de către funcţia principală (*main*) sau de către o altă funcţie, definită de utilizator.

## 5.2. Definirea sau descrierea funcţiilor

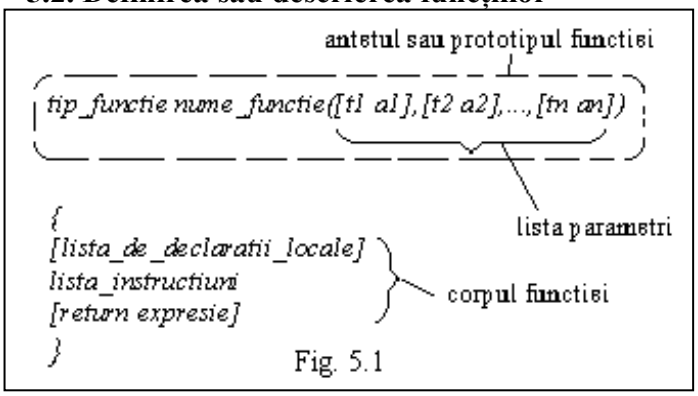


Fig. 5.1

Din punctul de vedere al proprietarului funcţiilor se cunosc:

a) *Funcţii predefinite* (sau *standard*), care sunt livrate împreună cu mediul de programare C şi se găsesc în diferite fişiere sau biblioteci (ex. *stdio.h, iostream.h, math.h,* etc.)

b) *Funcţii utilizator*, care sunt definite de utilizator şi se pot găsi:

- în fişierele si *bibliotecile utilizator* unde au fost introduse anterior de către utilizator;
- în fişierul sursă al programului curent, caz în care trebuie *definite(descrise)* sau *declarate* înaintea apelării lor.

*Sintaxa de definire(descriere) a unei funcţii utilizator* este descrisă în fig.5.1, putându-se evidenţia următoarele elemente:

- *tip_functie* poate fi orice tip de date valid, adică poate fi un tip *de bază* (char, int, float, double, etc.) sau un *tip derivat* (pointer, structură, etc.). Nu poate fi un tablou. Dacă funcţia nu returnează nici un rezultat, atunci *tip_functie* trebuie să fie *void*, iar dacă nu este specificat, se consideră implicit tipul *int*;
- *nume_functie* reprezintă un identificator utilizator, prin intermediul căruia se apelează funcţia pentru execuţie;
- *a1, a2, .., an* reprezintă parametrii formali sau argumentele formale ale funcţiei, care în general reprezintă datele formale de intrare în funcţie (reprezentate prin *variabile formale* sau prin *adrese de variabile formale*). La o funcţie fără argumente *lista de parametri* se înlocuieşte cu *void*;

```
int max(int a,int b)
{
if(a>b)
return(a);
else
return(b);
}
```
Fig. 5.2

*t1, t2, .., tn* reprezintă tipurile parametrilor formali (*int, char, float* şi derivatele lor). Când tipul unui parametru formal lipseşte, el este implicit considerat ca fiind *int;*

- *lista_de_declaraţii_locale* , opţională, conţine toate declaraţiile locale ale funcţiei, valabile în interiorul acesteia (aici intră declararea variabilelor locale);
- *lista_instrucţiuni* reprezintă toate instrucţiunile executabile care descriu subalgoritmul funcţiei;
- *return*, opţional, precizează, prin evaluarea expresiei pe care o precede, valoarea returnată de către funcţie către structura de program apelantă. Această valoare trebuie să fie de acelaşi tip cu tipul funcţiei.

În fig. 5.2 se prezintă un exemplu de funcţie care returnează cel mai mare dintre numerele întregi *a* şi *b* furnizate ca parametri.

## 5.3. Declararea funcţiilor

Dacă atât antetul cât şi corpul unei funcţii utilizator se află în fişierul sursă înaintea oricărui apel al funcţiei, avem de-a face cu *definirea* sau *descrierea funcţiei.* În cazul în care funcţia utilizator respectivă este definită *după apelul său* ori atunci când aceasta nu se află în fişierul sursă curent, este necesară *declararea* funcţiei înaintea oricărui apel la aceasta.

Declararea funcţiei presupune precizarea antetului său.

## 5.4. Variabilele locale

Aşa cum reiese din sintaxa generală a unei funcţii, şi în C (ca şi în alte medii de programare), în cadrul unei funcţii se

```
#include<iostream.h>
void val_local(void){
int a=1,b=2,c=3;
cout<<"a="<<a<<"b="<<b<<"c="<<c;}
void main(void){
cout<<"a="<<a<<"b="<<b<<"c="<<c;}
```

Fig. 5.3

pot declara *variabile locale*. Numele şi valoarea unei astfel de variabile sunt valabile doar în cadrul funcţiei care conţine declaraţia sa. În fig. 5.3 se prezintă un exemplu în care, în interiorul funcţiei *val_local* se declară 3 variabile locale: *a,b* şi *c*. Apoi, tot în cadrul acestei funcţii, se atribuie valori acestor variabile locale. Funcţia *main* încearcă să tipărească valoarea fiecărei variabile, dar compilatorul generează erori care anunţă că simbolurile *a,b* şi *c* sunt nedefinite.

## 5.5. Variabilele globale

Pe lângă variabilele locale programele mai pot folosi şi *variabile globale*, ale căror nume, valori şi existenţă sunt recunoscute în întregul program. Fig. 5.4 - exemplu de utilizare a acestora.

```
#include<iostream.h>
int a=1,b=2,c=3; //variabile globale
void val_global(void){
cout<<"a="<<a<<"b="<<b<<"c="<<c;}
void main(void){
val_global();}
```

Deoarece valoarea unei variabile globale se poate modifica din orice punct al programului care o utilizează, un alt programator care modifică un program în care sunt variabile globale le-ar putea modifica din greşeală. Ca regulă, funcţiile trebuie să modifice doar acele variabile care le sunt transmise ca parametri.

```
#include<iostream.h>

int a=1,b=2,c=3; //variabile globale

void conflict_a(void){
int a=100;
cout<<"a="<<a<<"b="<<b<<"c="<<c<<endl;}
void main(void){
                                    a =100 b=2 c=3

                                                    a =1 b=2 c=3
```

In ceea ce priveşte **rezolvarea conflictelor dintre numele variabilelor locale şi ale celor globale,** se pot specifica următoarele: este posibil ca numele unor variabile globale să fie identic cu cel al unor variabile locale. În exemplul din fig. 5.5 (a) se utilizează variabilele globale *a,b* şi *c*. Funcţia *conflict_a* foloseşte o variabilă locală numită *a* şi variabilele globale *b* şi *c*. În urma executării programului, pe ecran se vor afişa valorile reprezentate în fig. 5.5. (b)

Când numele unei variabile globale intră în conflict cu numele unei variabile locale se va folosi întotdeauna variabila locală. Se spune că *variabila locală "o ascunde" pe cea globală*.

Alegând locul în care se defineşte o variabilă globală se pot stabili funcţiile care se pot referi la variabila respectivă. Altfel spus, se poate controla ***domeniul de valabilitate al variabilei (*numit şi *scope*)**. Atunci când programul declară o variabilă globală, orice funcţie *care urmează declaraţiei* poate face referinţă la această variabilă, până la sfârşitul fişierului sursă. Funcţiile *definite înaintea unei variabile globale* nu o pot accesa.

## 5.6. Apelarea unei funcţii

Orice funcţie definită sau declarată de utilizator poate fi apelată de o altă funcţie definită de utilizator sau de funcţia *main*.

Când apelează o funcţie, programatorul trebuie să precizeze numele ei şi lista de parametri *efectivi* (numiţi şi *actuali* sau *reali*). La executarea funcţiei, aceşti parametri vor înlocui parametrii formali precizaţi la definirea sau declararea funcţiei.

Apelarea unei funcţii se poate face în două moduri:

a) *Apelarea independentă* (de sine stătătoare), cu următoarea sintaxă: *nume_funcţie([ae1],[ae2],...,[aen]),* unde *ae1,ae2,...,aen* reprezintă *datele efective de intrare* (literali, constante, variabile, funcţii sau expresii) sau *adresele datelor efective de intrare*. Pentru o *funcţie fără argumente* nu se pune nimic în locul listei de parametri efectivi.

Dacă o funcţie *apelată în mod independent* returnează o valoare, aceasta nu are nici o contribuţie la desfăşurarea algoritmului. De exemplu în instrucţiunea *f(2,9.5,8);* se apelează în mod independent funcţia $f$.

b) *Apelarea dependentă* , cu următoarea sintaxă: *e=...nume_funcţie([ae1],[ae2],...,[aen])...*

Exemplu: *e=9.6+3.2*f(7,3.1)*.

In cazul apelării dependente în cadrul unei instrucţiuni de atribuire, funcţia apelată în mod dependent trebuie să returneze o valoare de un anumit tip, compatibilă cu celelalte date şi cu operatorii din instrucţiunea de atribuire în care apare.

**Transferul parametrilor între funcţia apelantă şi funcţia apelată**

Transferul parametrilor poate fi de două tipuri: prin valorile parametrilor şi respectiv prin adresele parametrilor (prin referinţă).

1. **Transferul parametrilor prin valoare**

În cazul transferului parametrilor prin valoare la apelul unei funcţii, funcţiei i se transmite *o copie a valorilor parametrilor efectivi*. Orice modificare pe care funcţia o efectuează asupra parametrilor efectivi *transmişi prin valoare* pe parcursul execuţiei sale *este valabilă numai în interiorul funcţiei* (modificările se operează numai asupra unor cópii ale parametrilor efectivi). După terminarea execuţiei funcţiei apelate şi revenirea în funcţia apelantă, valorile efective ale parametrilor rămân neschimbate (ele sunt aceleaşi cu valorile avute înaintea apelării funcţiei). De aceea în acest caz se returnează funcţiei apelante numai valorile prevăzute în instrucţiunile *return* din corpul funcţiei apelate precum şi valorile *parametrilor modificaţi transmişi prin adresele lor*. În exemplul din fig. 5.6 se transmit 3 parametri (variabilele *a,b* şi *c*) funcţiei *afis_si_modif*, care va afişa valorile, le va adăuga valoarea 100 şi apoi va afişa rezultatul. După execuţia funcţiei, programul va afişa valorile variabilelor. Deoarece se foloseşte *apelul prin valoare*, funcţia nu modifică valorile variabilelor în

cadrul funcției apelante.

```
#include<iostream.h>
void afis_si_modif(int prima, int a_doua, int a_treia ){
cout<<"valorile originale ale parametrilor functiei sunt"<<prima<< a_doua<<a_treia<< endl;
prima+=100;
a_doua+=100;
a_treia+=100;
cout<<"valorile originale ale parametrilor functiei sunt"<<prima<< a_doua<<a_treia<<endl;}
void main(void){
int a=1,b=2,c=3;
afis_si_modif(a,b,c);
cout<<"valorile finale ale parametrilor in main sunt"<<a<< b<<c<<endl;}
```

Fig. 5.6

În urma executării programului, pe ecran se vor afișa rezultatele din fig. 5.7. Se observă că modificările variabilelor sunt vizibile numai în ca-drul funcției. După execuția *main*, valorile variabilelor rămân nemodificate.

```
valorile originale ale parametrilor functiei sunt 1 2 3
valorile finale ale parametrilor functiei sunt 101 102 103
valorile finale ale parametrilor in main sunt 1 2 3.
```

Fig. 5.7

**2. Transferul parametrilor prin adrese sau referințe**

Dacă se utilizează transferul parametrilor prin adrese sau referințe, atunci când se apelează funcția, acesteia i se transmit *adresele parametrilor efectivi*.

În acest caz, pe parcursul execuției sale, orice modificare pe care o efectuează funcția asupra parametrilor efectivi este valabilă atât în interiorul funcției cât și în exteriorul său. După terminarea execuției funcției apelate și revenirea în funcția apelantă, valorile efective ale parametrilor rămân cele modificate în timpul execuției funcției apelate.

Ca urmare, în urma apelării și executării unei funcții ai cărei parametri au fost transmiși prin adresă, se returnează funcției apelante:

- valorile expresiilor prevăzute de *return* din corpul funcției;
- valorile parametrilor modificați, transmiși prin adresele lor.

Deci, pentru ca în urma apelării și execuției sale o funcție să poată modifica valorile unor parametri efectivi, trebuie ca:

- la *definirea* sau declararea funcției să se indice *adresele* acestor parametri formali (adică variabile de tip pointer spre variabilele respective);
- la *apelarea* funcției să se transmită *adresele parametrilor efectivi* (folosind operatorul &).

```
#include <iostream.h>
void modif_prim(int *prim, int al_doilea){
// atribuie primului param. val. celui de-al 2-lea param.
*prim=al_doilea;
al_doilea=100;}
void main(void){
int a=0,b=5;
modif_prim(&a,b);
cout<<"a="<<a<<", b="<<b;}
```

Fig. 5.8

Spre deosebire de apelul prin valoare, la apelul prin adresă al unui parametru efectiv se transmite funcției apelate *adresa de memorie* a variabilei ce reprezintă parametrul efectiv. La apelul prin adresă, funcția efectuează modificarea variabilei nu într-o copie ci *direct în zona de memorie în care se salvează variabila*, modificarea menținându-se și după execuția funcției și revenirea în funcția apelantă.

În fig. 5.8 este redat un program în care se definește și apelează funcția *modif_prim*. În antetul funcției apar doi parametri: primul, numit *prim*, este transmis *prin referință*, iar al doilea, numit *al_doilea* este transmis *prin valoare*. În urma execuției programului, se afișează valorile celor doi parametri, adică se va afișa: *a=5, b=5*.

Între parametrii formali utilizați la definirea sau declararea funcției și respectiv parametrii efectivi utilizați la apelul funcției trebuie să existe o *compatibilitate biunivocă referitoare la tipul parametrilor și respectiv la numărul lor*. În plus, pozițiile ocupate de aceștia în lista de definire a parametrilor formali trebuie să fie aceleași cu cele ocupate în lista parametrilor efectivi. Dacă tipurile nu sunt compatibile, compilatorul nu va semnala eroare, dar apar rezultate imprevizibile după apelarea și executarea funcțiilor.

```
....
f1(){
char c;
cout<<"continuam?Y/N"<<endl;
cin>>c;
if(c=='y'){
char s[80]; //
cout<<"introduceti numerele"<<endl;
.....
}
```

Fig. 5.9

De obicei variabilele locale utilizate în interiorul unei funcții se declară la începutul blocului de cod al acesteia. Acest lucru nu este însă neapărat necesar, declararea putându-se face oriunde în interiorul blocului, dar înainte de a fi folosite. În exemplul din fig. 5.9, funcția *f1* creează variabila locală *s* doar după intrarea în blocul aferent lui *if* .

# Cursul 6

## 5.7. Specificatori de clasă de memorie

Atunci când se declară o variabilă, înaintea tipului său se poate preciza şi un *specificator de clasă de memorie*. Sintaxa *generală* utilizată pentru declararea unei liste de variabile este:

*[specificator_de_clasă_de_memorie] specificator_de_tip lista_de_variabile*;

**Fisier sursa 1**

```
#include<iostream.h>

int i;
        extern float a,b;
float a,b;
        void g(){
//declarare prototip functie f
void f();
        i*=100;//i=100*30=3000
void main(void){
        a*=100;//a=50.25*100=5025
i=10;
        b*=100;//b=49.75*100=4975
a=20.25;
        }
b=79.75;
//se afiseaza i=10,a=20.25,b=79.75
cout<<"i="<<i<<",a="<<a<<",b="<<b;
f();
//se afiseaza i=30,a=50.25,b=49.75
cout<<"i="<<i<<",a="<<a<<",b="<<b;
// se va apela functia g din fisierul sursa 2
g();
//se afiseaza i=3000,a=5025.0,b=4975.0
```

**Fisier sursa 2**

```
extern int i;

{
```

Specificatorul de clasă de memorie specifică clasa de memorie căreia îi aparţin variabilele din lista de variabile şi poate fi: *auto, extern, static* şi *register*.

1. Variabilele declarate cu ajutorul specificatorului de clasă de memorie *auto* sunt variabile *automate*. O astfel de variabilă va fi locală unei funcţii sau unui bloc. Dacă la declararea unei variabile declarată într-o funcţie sau într-un bloc nu s-a făcut nici o declarare de clasă de memorie, atunci aceasta se consideră implicit *auto*.

2. Variabilele locale declarate explicit folosind cuvântul cheie *extern* sunt de fapt tot variabile globale. Dacă programul utilizatorului este compus din mai multe fişiere sursă, atunci variabilele externe se declară ca globale într-un singur fişier sursă şi ca externe în celelalte fişiere sursă.

Cuvântul cheie *extern* anunţă compilatorul că variabila respectivă a fost declarată extern (în afara fişierului sursă curent) de un alt program.

În exemplul din fig. 5.10 se utilizează *un proiect* format din două fisiere sursă. În functia *main* din fisierul sursa 1 se apelează funcţiile *f* şi *g*. *f* modifică valorile variabilelor *i,a,b*, care sunt declarate ca şi variabile globale în fisierul sursa 1. Modificarea acestora este resimţită şi de functia *g*, pentru că *i, a* şi *b* sunt declarate cu specificatorul de clasă *extern* în fisierul sursa 2. La rândul său, atunci când este apelată, *g* modifică aceste variabile, modificarea respectivă resimţindu-se în funcţia *main*.

3. Variabilele statice sunt declarate cu specificatorul de clasă *static*. O astfel de variabilă *îşi păstrează valoarea pe parcursul executării programului*.

Atunci când se declară o variabilă ca statică într-o funcţie, la primul apel, compilatorul va iniţializa variabila cu valoarea indicată. La un nou apel ulterior al aceleiaşi funcţii, nu se mai face iar iniţializarea.

Variabilele statice pot fi interne sau externe.

a) Variabilele statice interne sunt locale funcţiilor, dar spre deosebire de cele declarate cu *auto* îşi păstrează valoarea pe parcursul executării programului. Aceste variabile sunt importante în definirea funcţiilor de sine-stătătoare atunci când se doreşte păstrarea valorii lor între două apelări. Un exemplu de astfel de funcţii este cel al funcţiilor utilizate la generarea seriilor de numere care calculează o valoare plecând de la valoarea precedentă. O altă soluţie care se poate adopta pentru păstrarea valorii anterioare constă în utilizarea unei variabile globale, care ar trebui să fie declarată de fiecare dată. În plus, trebuie avut grijă ca aceasta să nu interfereze cu alte variabile globale deja amplasate în alte fişiere sursă.

```
#include<iostream.h>
int serie(int q) {
static int termen_serie=10;
termen_serie+=q;
return (termen_serie);}
void main(void){
int ratia,n,i;
cout<<"dati numarul de termeni"<<endl;
cin>>n;
cout<<"dati ratia"<<endl;
cin>>ratia;
for(i=1;i<=n;i++)
cout<<serie(ratia)<<endl;}
```

Fig. 5.11

În exemplul din fig. 5.11 se generează seria aritmetică cu *n* numere întregi care are primul termen 10.

b) Variabilele statice externe se declară în afara oricărei funcţii (sunt globale), dar sunt recunoscute doar de către toate funcţiile care fac parte *din acelaşi fişier sursă*. Dacă se doreşte ca anumite variabile globale dintr-un anumit fişier să nu poată fi accesate de funcţii din afara acestuia (din alte fişiere), trebuie ca respectivele variabile să fie declarate utilizând *static*.

4. Variabilele registru sunt declarate cu specificatorul de clasă *register*. Aceste variabile sunt locale funcţiilor şi blocurilor. Iniţial specificatorul de clasă *register* cere compilatorului să păstreze valoarea variabilei de tip *register* într-un registru al procesorului în loc de a o stoca în memoria internă (aşa cum procedează în mod obişnuit). Acest lucru determină ca operaţiile aplicate asupra unei variabile de acest tip să se desfăşoare cu o viteză mult

mai mare în raport cu o variabilă normală. Într-un program, numărul permis de variabile *register* care beneficiază de optimizarea vitezei este determinat atât de mediul de programare cât şi de varianta de implementare a compilatorului.

```
#include<iostream.h>
int factorial(int val){
if (val==1) return (1);
else return (val*factorial(val-1));}
void main(void){
int i;
for(i=1;i<=5;i++)
cout<<"factorial de"<<i<<"="<<factorial(i)<<endl;}
```

Fig. 5.12

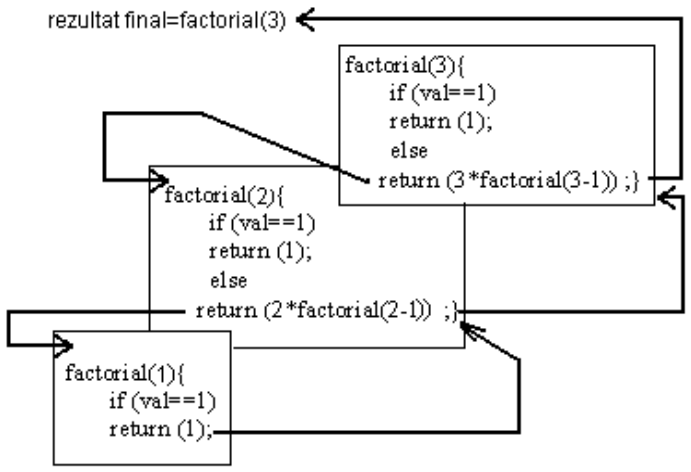



Fig.5.13

## 5.8. Funcţii recursive

O *funcţie recursivă* este o funcţie care se apelează pe sine însăşi pentru a efectua o anumită operaţie, iar procesul în care o funcţie se apelează pe sine însăşi se numeşte *recursivitate*.

În exemplul din fig. 5.12 este definită o funcţie *factorial*, care calculează factorialul numărului întreg furnizat ca parametru. Apoi, în funcţia *main* , se apelează *factorial* într-un ciclul *for* pentru a afişa factorialul primelor 5 numere naturale. Se observă ca funcţia *factorial* returnează un rezultat care se bazează pe propriul ei rezultat. În fig. 5.13 se descrie lanţul de apelări recursive şi de returnări de valori la apelul *factorial(3)*.

O funcţie recursivă este oarecum asemănătoare cu o structură ciclică, deoarece trebuie precizată o *condiţie de încheiere*. Dacă nu se face această precizare, funcţia nu se sfârşeşte niciodată. În acest exemplu, condiţia de încheiere este *factorial(1)* care, prin definiţie, este 1.

Dacă funcţia se apelează pe sine însăşi pentru a îndeplini o sarcină, funcţia execută un *proces recursiv direct* . Forma mai dificilă a recursivităţii, *recursivitatea indirectă*, are loc atunci când o funcţie (să o notăm cu A) apelează o altă funcţie (să o notăm cu B), care la rândul său o apelează pe A. Acest tip de recursivitate poate conduce la realizarea unui cod dificil de înţeles, aşa că de obicei trebuie evitată utilizarea sa.

Funcţiile recursive sunt lente, pentru că la fiecare apel se introduce în program o *suprasarcină de apel*. De aceea se recomandă pe cât posibil evitarea utilizării recursivităţii. Orice funcţie care poate fi scrisă într-o formă recursivă poate fi de asemenea scrisă într-o structură ciclică. În exemplu din fig. 5.14 se prezintă rezolvarea problemei din fig. 5.12, dar fără a utiliza recursivitatea.

```
#include<iostream.h>
int factorial(int val){
int rezultat=1;
int contor;
for(contor=2;contor<=val;contor++)
rezultat*=contor;
return (rezultat);}
void main(void){
int i;
for(i=1;i<=5;i++)
cout<<"factorial de"<<i<<"="<<factorial(i)<<endl;}
```

Fig. 5.14

În exemplul din fig. 5.15 se prezintă un program în care se calculează şi afişează suma, diferenţa şi produsul a două numere reale, furnizate sub formă de date de intrare de către utilizator. Se utilizează o funcţie *f* care returnează suma. Diferenţa şi produsul sunt transmise către funcţia care o apelează pe *f* deoarece în lista de parametri formali ai lui *f* se utilizează doi parametri transmişi prin adresă: *xy* va memora produsul iar *x_y* va memora diferenţa.

Un alt exemplu de utilizare a funcţiilor este următorul: se dau numerele naturale *n* şi *k* ($n \geq k$). Să se determine combinări de *n* luate câte *k*. Se utilizează formula: $C^k_n = n!/((n-k)!*k!)$. Se va folosi o funcţie definită de utilizator pentru calculul unui produs factorial (fig. 5.16).

În acest exemplu s-a arătat şi cum se poate face validarea datelor introduse de utilizator. Astfel, dacă se dau valori greşite, se reia citirea datelor.

```
#include<iostream.h>
float f(float x, float y, float *xy, float *x_y){
*xy=x*y;
*x_y=x-y;
return(x+y);}
void main(void){
float a,b,s,d,p;
cout<<"a="<<endl;cin>>a;
cout<<"b="<<endl;cin>>b;
s=f(a,b,&p,&d);
cout<<"suma="<<s<<",dif.="<<d<<",prod="<<p;}
```

Fig. 5.15

# CAPITOLUL 6. TABLOURI SI SIRURI DE CARACTERE

## 6.1. Declararea şi iniţializarea unui tablou. Referirea elementelor unui tablou

Un tablou reprezintă o variabilă care poate să păstreze mai multe valori de acelaşi tip. Pentru a declara un tablou, trebuie specificat tipul elementelor care îl compun precum şi numărul lor. Sintaxa de declarare a unui tablou este: *tip_elemente_tablou nume_tablou[dim_1] [dim_2],...,[dim_n]*, unde:

```
#include<iostream.h>
int fact(int val){
int rezultat=1;
int contor;
for(contor=2;contor<=val;contor++)
rezultat*=contor;
return (rezultat);}
void main(void){
int n,k,combinari;
do {
cout<<"'n";cin>>n; cout<<"'k";cin>>k;}
while((k>n) || (k<1) || (n<1));
combinari=fact(n) / fact(n-k) / fact(k);
cout<<"comb. de"<<n<<"luate cate"<<k<<"="<<combinari;}
```

Fig. 5.16

- *tip_elemente_tablou* precizează tipul elementelor tabloului (*int, char, float* sau derivate ale lor);
- *nume_tablou* este un identificator utilizator care reprezintă numele tabloului prin intermediul căruia vor fi referite componentele (elementele) tabloului;
- *dim_i* (i=1,...n) sunt date întregi, strict pozitive (literali, constante) care precizează mărimea tabloului.

Numărul datelor *dim_i* determină *dimensiunea tabloului*. Pentru tablourile unidimensionale (vectori, şiruri) se precizează o singură dată *dim_1*, pentru matrici bidimensionale (dreptunghiulare) se precizează două date *dim_1, dim_2*, ş.a.m.d.

*Adresarea unei componente* se face prin indice, încadrat de paranteze drepte.

Tabloul declarat prin *int v[100]* poate conține maxim 100 de componente de tip *int*. Indicii componentelor lui *v* pot lua valori în gama [0,99]. De exemplu, când vrem să adresăm componenta a doua, scriem v[1].

De exemplu prin *float a[10][9]* se declară o matrice de 10 linii, 9 coloane, cu elemente de tip *float*. Indicii liniilor pot avea valori în plaja [0,9], iar cei aferenți coloanelor pot lua valori în plaja [0,8]. Primul element este *a[0,0]*, iar ultimul este *a[9,8]*.

*Observație* În C nu se poate declara o variabilă de tip tablou cu un număr *variabil* de componente. De aceea, când nu se cunoaşte dinainte valoarea la care poate ajunge un indice, se rezervă un număr maxim de componente, cel care este necesar pentru acea execuție a programului în care indicele este maxim. Astfel, de multe ori, o parte din componente rămân neutilizate.

Tablourile pot fi *inițializate*. Când se atribuie valori inițiale, acestea trebuie încadrate între acolade.

De exemplu prin *int a[5]={3,2,5,10,77}* s-a inițializat tabloul *a* a.î.: *a[0]=3; a[1]=2; a[2]=5;a[3]=10;a[4]=77*.

Următoarea instrucțiune atribuie 3 valori unui tablou care poate păstra 20! *float s[20]={2.0,-3.0,0.0}*

În funcție de compilator, se poate atribui valoarea 0 elementelor pentru care programul nu atribuie în mod explicit o valoare. Ca regulă însă, nu trebuie să se presupună că va iniția compilatorul celelalte elemente. Mai mult, dacă nu se precizează dimensiunea tabloului, compilatorul va aloca atâta memorie cât este necesară pentru a păstra numai valorile specificate.

Pentru a se evita, prin inițializare, depăşirea dimensiunilor tablourilor şi deci apariția erorilor legate de această situație, se folosesc *tablouri adimensionale*, conform sintaxei: *tip_element_tablou nume_tablou[ ][ ]...={lista_de_valori};*

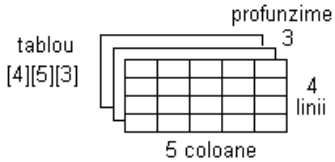
```
#include <stdio.h>                                  #include
<stdio.h>
void main(void) {                                   #define
DIMENSIUNE 5
int val[5]={4,2,19,5,7};                        void main(void) {

int i;                                                          int
val[DIMENSIUNE]={4,2,19,5,7};
for(i=0;i<5;i++)                                    int i;
```

Compilatorul va crea automat tablouri suficient de mari pentru a memora toți literalii de inițializare.

Exemplu: *int e[ ]={1,5,12}*.

Atunci când se lucrează cu un tablou trebuie specificată dimensiunea acestuia. În programul din fig. 6.1 a) se declară un tablou cu 5 valori şi apoi se utilizează un ciclu *for* pentru a afişa valorile sale. Dacă ulterior se doreşte modificarea acestui program a.î. acesta să accepte 10 valori, trebuie operate două modificări, atât în dimensiunea declarată a tabloului cât şi în ciclu. Cu cât se aduc mai multe modificări programului, cu atât va creşte şi şansa de a se produce erori. O alternativă este utilizarea de constante pentru dimensiunile tablourilor, ca în fig. 6.1 b). Dacă ulterior trebuie modificate dimensiunile tabloului se pot modifica numai valorile constantelor utilizate pentru dimensiuni.




```
#include <iostream.h>
void main(void){
int v[100],n,i;
cout<<"introduceti numarul de componente"; cin>>n;
for(i=0;i<n;i++) {
cout<<"v["<<i<<"]=";
cin>>v[ i ];}
for(i=0;i<n;i++) cout<<v[ i ]<<endl; }
```

Fig. 6.3

Relativ la **inițializarea elementelor intr-un tablou bidimensional**, se utilizează o tehnică asemănătoare cu cea utilizată la inițializarea unui tablou cu o singură dimensiune, ca în exemplul următor:

```
int tabel[2][3]={{2,3,1},
                 {3,-2,5}};
```

Tablourile pot avea şi mai mult de două dimensiuni (exemplu în fig. 6.2).

Pentru a înțelege cum se inițializează un tablou cu 3 dimensiuni, se prezintă exemplul de mai jos.

```
int b[2][3][4]={
                {
{1,2,3,4},{5,6,7,8},{2,4,2,1}},
                { {9,2,3,1},{3,6,8,8},{2,4,6,2}}
```

```
}
```

```
#include <iostream.h>
void main(void){
float a[100], b[100];
int n,i;
cout<<"introduceti numarul de componente"; cin>>n;
for(i=0; i<n; i++){
cout<<"a["<<i<<"]="; cin>>a [ i ]; b[ i ]=a[ i ];}
for(i=0; i<n ;i++) cout<<b[ i]<<endl; }
```

Fig. 6.4

Se observă că fiecare inițializare a unui tablou este redată între acoladele exterioare. Între cele două acolade exterioare, fiecare dintre elementele tabloului este definit între alte acolade.

**Exemple de programe care utilizează tablouri**

Exemplul 1: Programul din fig. 6.3 citeşte şi apoi tipăreşte componentele unui vector v[n].

Exemplul 2: În programul din fig. 6.4 se citeşte vectorul *a*, care are componente de tip *float*. Apoi se copiază valorile sale, prin atribuire, în componentele vectorului *b*.

*Observație* Dacă *a* şi *b* sunt tablouri, nu sunt permise atribuiri de forma *b=a*. Atribuirea trebuie să se facă pe componente.

```
#include <iostream.h>
void main(void){
int m,n,i,j,a[9][9];
cout<<"introduceti nr. de linii"; cin>>m;
cout<<"introduceti nr. de coloane"; cin>>n;
for(i=0;i<m;i++)
        for(j=0;j<n;j++)
    {cout<<"a["<<i<<"]["<<j<<"]=";
     cin>>a[ i ][ j ];}
for(i=0;i<m;i++){
        for(j=0;j<n;j++)
     cout<<a[ i ][ j ]<<" ";
    cout<<endl;}
}
```

Fig. 6.5

Exemplul 3: În programul din fig. 6.5 se citesc şi tipăresc elementele unui tablou (*m x n*). La afişare fiecare linie a tabloului apare pe o linie separată.

## 6.2. Tablouri de structuri ce conțin alte tablouri de structuri

Tablourile şi structurile permit gruparea informațiilor înrudite. Se pot crea tablouri de structuri şi se pot utiliza tablouri ca membri ai unor structuri. De exemplu declarația din fig. 6.6 creează un tablou de 100 de structuri de angajați. În cadrul fiecărei structuri există un alt tablou, cu 3 componente care sunt structuri de tip *Date*, acestea corespunzând datei de angajare a salariatului, datei corespunzătoare primei şi respectiv ultimei sale examinări.

Pentru accesarea membrilor şi elementelor matricei se va lucra de la stânga la dreapta, începând din exterior spre interior. În fig. 6.7 se prezintă un exemplu prin care se atribuie data de angajare a celui de al 11-lea salariat memorat de tabloul personal.

```
struct Angajat{
 char nume[30];
 int varsta;
 char numar_asig[11];
 int categ_salarizare;
 float salariu;
 unsigned nr_angajat;
 struct Date {
  int luna;
  int ziua;
  int anul;
  } date_angajat[3];
} personal[100];
```

Fig.6.6

## 6.3.Uniunea (UNION)

Structurile permit păstrarea de informații legate între ele. Uneori trebuie stocate într-o structură informații care nu vor consta numai din una sau două valori. De exemplu să presupunem că un program urmăreşte valorile a două date speciale pentru fiecare angajat. Pentru persoanele angajate în prezent interesează numărul de zile lucrate de angajat. Pentru persoanele care nu mai lucrează în cadrul societății interesează data ultimei sale zile de lucru. O cale pentru a memora o astfel de informație poate fi utilizarea unei structuri, ca în fig. 6.8 (a). Deoarece programul va utiliza fie membrul *zile_lucru*, fie membrul *ultima_zi*, memoria care păstrează informația neutilizată se va irosi. Ca alternativă, compilatorul permite utilizarea *union*-ului, care alocă numai memoria cerută de cel mai mare dintre membrii structurii, aşa cum se prezintă în fig. 6.8 (b). Accesarea membrilor structurii se face ca şi în cazul structurilor, folosind operatorul "**.** " (*dot*). Spre deosebire de structură, uniunea păstrează numai valoarea unui membru. Fig. 6.9 ilustrează modul în care compilatorul alocă memorie pentru structură şi uniune (s-a presupus că pentru memorarea unei variabile de tip *int* se utilizează 4 octeți).

```
personal[10].date_angajat[0].luna=11;
personal[10].date_angajat[0].ziua=3;
personal[10].date_angajat[0].anul=2001;
```

Fig. 6.7

```
struct date_angajat{                    union
date_angajat{  int zile_lucru;
          int zile_lucru;
 struct Ultima_data                     struct
Ultima_data
          {
            {
           int luna;
int luna;
           int ziua;
          int ziua;
```

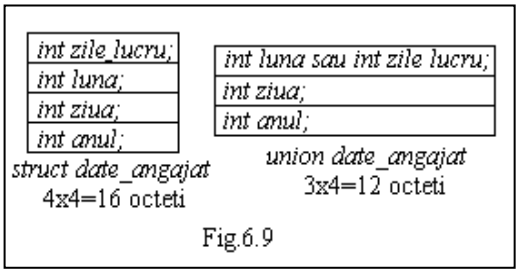



Fig.6.9

## Cursul 7

## 6.4. Structurile câmp de biți

```
struct data
        {
        int
luna;
        int
ziua;
     int anul;
```

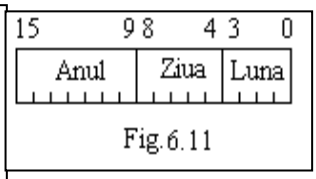


Fig.6.11

Atunci când biții care compun o valoare au o semnificație specifică, programele pot utiliza *operatorii C pe biți* pentru a extrage biții respectivi. Să presupunem de exemplu că într-un program se manipulează 10000 de date calendaristice. O primă abordare a problemei este cea în care se utilizează o structură de date de tip *data*, ca în fig. 6.10. O altă soluție este cea care utilizează *biți specifici*, aceştia fiind reuniți în cadrul unei valori *unsigned short int* în scopul memorării celor 3 câmpuri, ca în fig. 6.11 (s-a presupus că o variabilă de tipul *int* se memorează pe 32 biți).

```
unsigned data;
...
data=luna;// se atribuie cei mai din dreapta 4 biti pentru luna
data=data | (ziua<<4); // se face OR pe biti pentru campul ziua deplasat la stanga cu 4 pozitii
// corespunzatoare memorarii campului luna
data=data | (anul<<9); // se face OR pe biti pentru campul anul  deplasat la stanga cu 9 pozitii
// corespunzatoare memorarii campurilor luna si ziua
...
```

Fig. 6.12

```
struct Data                          data.luna=12;
 {                                          data.ziua=31;
 unsigned luna:4;          data.anul=94;
 unsigned ziua:5;
          cout<<data.luna<<data.ziua<<data.anul;
 unsigned anul:7;
} data;
```

(a)

*Observație* Numărul de biți necesari pentru memorarea fiecărei zone de biți s-a calculat astfel: pentru memorarea câmpului *luna* , care poate atinge valoarea maximă 12 (adică $2^4$ -4) sunt necesari 4 biți, deci se rezervă primii 4 biți (0...3) .Pentru memorarea câmpului *ziua* , care poate atinge valoarea maximă 31 (adică $2^5$ -1) sunt necesari 5 biți, deci se rezervă următorii 5 biți (4...8). Pentru memorarea câmpului *anul* , care va reține doar ultimele două cifre şi poate atinge valoarea maximă 100 (adică $2^7$ -28) sunt necesari 7 biți, deci se rezervă următorii 7 biți (9...15).

Apoi, ori de câte ori programul urmează să atribuie data, el poate executa operațiile corecte pe biți, ca în fig. 6.12.

Pentru a face programele mai uşor de înțeles, compilatorul de C permite crearea unei ***structuri câmp de biți***. Atunci când se declară o astfel de structură, se defineşte o structură care specifică înțelesul biților corespunzători, aşa cum se prezintă în fig. 6.13 a). Programele vor face referire individuală la câmpul de biți, după cum se exemplifică în fig. 6.13 b).

*Observație* Atunci când se declară o structură câmp de biți, fiecare membru al structurii trebuie să fie o valoare de tip *unsigned int*.

## 6.5. Funcții care folosesc ca argumente tablouri

Atunci când **se apelează** o funcție care are ca argument un tablou, acesteia i se va transmite *un pointer la **primul element** al tabloului* (acest pointer reține adresa de la care începe memorarea tabloului).

*Observație* În C numele unui tablou unidimensional urmat de două paranteze drepte între care nu se specifică nimic reprezintă un pointer la primul element al tabloului. (Ex. *v [ ]*). Asemănător *v [ ] [ ]* reprezintă un pointer spre primul element al tabloului bidimensional *v*, etc.

Relativ la **declararea** unui parametru formal al unei funcții care urmează să primească un pointer la un tablou, se poate spune că în C există 3 modalități prin care se poate realiza acest lucru, prezentate în cele ce urmează.

a) Declararea parametrului formal ca şi *tablou dimensionat*

Exemplificăm acest caz prin programul din fig. 6.14 , în care în funcția *main* se generează elementele unui vector *a* cu formula: *a[i]=3*i*. Apoi se apelează o funcție definită de utilizator, numită *afiseaza*, care are ca parametru un *tablou dimensionat*. Funcția afişează componentele vectorului *a* generat în funcția *main*.

```
#include<iostream.h>

void afiseaza(int v[15]) {

int k;

cout<<"compon. vect."<<endl;
for(k=0;k<15;k++)

cout<<"v["<<k<<"]="<<v[k]<<" ";}
void main(void) {
int a[15],i;
```

b) Declararea parametrului formal ca şi *tablou nedimensionat* Considerăm acelaşi program prezentat la punctul a). Se înlocuieşte *v [15]* cu *v [ ]*. În acest caz compilatorul converteşte automat pe *v [ ]* la un pointer spre o valoare de tip întreg, valoare care este prima componentă a tabloului.

c) Declararea parametrului formal *ca pointer*

Considerăm acelaşi program prezentat la punctul a). Se înlocuieşte *v[15]* cu **v*. Acest tip de declarație este permis deoarece orice pointer poate fi indexat folosind paranteze drepte, asemănător unui tablou.

Toate cele 3 metode de declarare a unui tablou ca parametru produc acelaşi rezultat: un pointer.

*Exemplu* Se dau matricile *a,b* şi *c* de dimensiuni *n x n*. Să se calculeze matricea *s=a*b+c*a*. Să se afişeze *a,b,c,s*. Soluția este prezentată în fig. 6.15.

O constantă de tip şir de caractere se reprezintă prin încadrarea caracterelor şirului între ghilimele, ca în exemplul

următor:

```
#include<iostream.h>
#define nmax 10
void citeste(int n, char nume, float x[nmax][nmax]) {
int i,j;
for(i=0;i<n;i++)
        for(j=0;j<n;j++) {
        cout<<nume<<"["<<i<<"]["<<j<<"]=";
        cin>>x[i][j];}}
void scrie(int n, float x[nmax][nmax]) {
        int i,j;
        for(i=0;i<n;i++) {
         for(j=0;j<n;j++)
         cout<<x[ i ][ j ]<<" ";
         cout<<endl;}}
void produs(int n, float x[nmax][nmax],float y[nmax][nmax],float xy[nmax][nmax])
 {
int i,j,k;
        for(i=0;i<n;i++)
          for(j=0;j<n;j++){
            xy[ i ][ j ]=0;
            for(k=0;k<n;k++)
                xy[ i ][ j ]= xy[ i ][ j ]+x[ i ][ k ]*y[ k ][ j ];}}
void main(void) {
int n,i,j;
float a[nmax][nmax], b[nmax][nmax], c[nmax][nmax];
float ab[nmax][nmax], ca[nmax][nmax], s[nmax][nmax];
do {
cout<<"n=";cin>>n;}
while((n<1) || (n>nmax)); //validare introducere date
citeste(n,'a',a); citeste(n,'b',b); citeste(n,'c',c);
produs(n,a,b,ab); produs(n,c,a,ca);
        for(i=0;i<n;i++)
                for(j=0;j<n;j++)
                s[ i ][ j ]=ab[ i ][ j ]+ca[ i ][ j ];
cout<<"matricea a"<<endl;scrie(n,a);
cout<<"matricea b"<<endl;scrie(n,b);
cout<<"matricea c"<<endl;scrie(n,c);
cout<<"matricea s"<<endl;scrie(n,s);}
```

Fig. 6.15

```
#include <stdio.h>
void main(void){
char sir[256];
int i;
for(i=0;i<26;i++)
sir[i]='A'+i;
sir[i]=NULL;
printf("sirul de caractere contine %s\n",sir);}
```

Fig. 6.17

```
#include <stdio.h>
void main(void){
char sir[256];
int i;
printf("introduceti sirul si apoi apasati Enter:\n" );
gets(sir);
for(i=0;sir[i]!=NULL;i++)
putchar(sir[i]);
printf("numarul de caractere in sir este %d\n",i);}
```

Fig.6.18

*"Sir de caractere 234"*

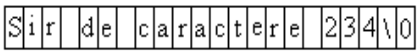



Fig. 6.16

Atunci când se utilizează o constantă de acest tip într-un program, compilatorul C va adăuga automat caracterul NULL (\0) la sfârşitul şirului. Constanta din exemplul nostru va fi stocată în memorie aşa cum reiese din fig. 6.16.

### 6.6.2. Stocarea unui şir de caractere

Unele programe utilizează pe scară largă şiruri de caractere. De exemplu unele programe utilizează şiruri de caractere pentru a citi fişiere sau date de la tastatură, precum şi pentru alte operaţii. Ín C un şir de caractere este un tablou de caractere terminat cu NULL. Pentru a crea un şir de caractere, se declară pur şi simplu un tablou de caractere, ca în exemplul următor: *char sir[256]*.

Compilatorul va crea pentru şirul din acest exemplu un şir capabil să păstreze 256 de caractere, pe care le va indexa începând cu *sir[0]* până la *sir[255]*. Deoarece şirul poate conţine mai puţin de 256 de caractere, se utilizează NULL pentru a reprezenta ultimul caracter al şirului.

În exemplul din fig. 6.17 se defineşte un şir de caractere de dimensiunea 256 şi apoi se atribuie primelor 26 de locaţii literele mari ale alfabetului latin.

*Observaţie* Atunci când se lucrează cu caractere se poate utiliza valoarea numerică a caracterului în reprezentarea ASCII sau se poate plasa caracterul respectiv între apostrofuri (ex. 'A'). Dar atunci când se utilizează ghilimelele (ex. "A"), compilatorul creează un şir de caractere care conţine litera specificată şi termină şirul cu caracterul NULL.

Ín concluzie, deoarece sunt stocate în mod diferit, constantele de tip caracter şi cele de tip şir de caractere nu sunt similare şi trebuie tratate diferit în cadrul programelor.

*Exemplu* În fig. 6.18 se prezintă un program care realizează următoarele: utilizatorul este invitat să introducă de la tastatură un şir de caractere, iar după ce a introdus şi ultimul caracter trebuie să introducă *Enter*. Apoi, printr-un ciclu *for* se afişează unul câte unul caracterele şirului, până când condiţia *sir[i]=NULL* este evaluată ca fiind falsă.

### 6.6.3. Funcţii mai importante pentru lucrul cu şiruri de caractere

**1. Calculul lungimii unui şir**

În exemplul din fig. 6.18 nu s-au utilizat funcţii specifice lucrului cu şiruri de caractere. Există însă o bibliotecă (*string.h*) în care sunt implementate numeroase astfel de funcţii.

De exemplu funcţia *strlen* returnează numărul de caractere al şirului furnizat ca parametru. În exemplul din fig. 6.19 se prezintă

un exemplu de utilizare a acestei funcții.

```
#include <stdio.h>
#include <string.h>
void main(void){
char sir[ ]=”exemplu de sir”;
printf(“%s contine %d caractere \n”,sir,strlen(sir));}
```

Fig. 6.19

Atunci când se execută programul, se afişează: *exemplu de sir contine 14 caractere*.

**2. Copierea unui şir de caractere in altul**

Cu ajutorul funcției *strcpy* se poate copia conținutul unui şir de caractere (*parametrul sursă*) în alt şir de caractere (*parametrul destinație*). Funcția *strcpy* returnează un pointer care indică începutul şirului destinație şi are următorul prototip:

```
#include <stdio.h>
#include <string.h>
void main(void){
char sir_copiat[ ]="exemplu de sir";
char sir_destinatie[128];
strcpy(sir_destinatie,sir_copiat);
printf("sirul destinatie are continutul %s",sir_destinatie);}
```

Fig. 6.20

*char *strcpy(char *destinatie, const char *sursa);*

În programul din fig. 6.20 se prezintă un exemplu de copiere a unui şir (memorat de variabila cu numele *sir_copiat*) într-un alt şir (memorat de variabila cu numele *sir_destinatie*).

**3. Alte funcții utile pentru lucrul cu şiruri**

Biblioteca *string.h* este vastă, conținând multe funcții utile. Spațiul cursului nu permite explicarea tuturor. Merită menționate ca mai uzuale:

- *strchr* si *strrcchr* care primesc 2 parametri: primul este un sir în care se efectuează căutarea caracterului precizat ca al doilea parametru. Prima funcție returnează pointer la prima apariție a caracterului, a doua la ultima apariție a caracterului. Dacă acesta nu este găsit, se returnează NULL.
- *strcmp* primeste ca parametri două siruri *s1* si *s2* care sunt comparate (conform ordinii lexicografice). Returnează o valoare care este: negativă dacă s1<s2, egală cu 0 dacă s1=s2 si respectiv pozitivă dacă s1>s2.
- *strrev* inversează sirul primit ca parametru;
- *strstr* primeste 2 parametri: sir1 si sir2. Se returnează pointer spre acel element din sir1 de la care începe sir2 (dacă sir1 include pe sir2) sau NULL în caz contrar.

# CAPITOLUL 7. FUNCȚII MATEMATICE

În fig. 7.1. este prezentat conținutul bibliotecii *math.h* aferente versiunii de Turbo C cu care se lucrează la laborator (asa cum îl prezintă *help*-ul).

```
math.h
abs         acos        asin        atan
atan2       atof        cabs        ceil
cos         cosh        exp         fabs
floor       fmod        frexp       hypot
labs        ldexp       log         log10
matherr     modf        poly        pow
pow10       sin         sinh        sqrt
tan         tanh
```

Fig. 7.1

| Nume functie | Argumente | Observatii |
|---|---|---|
| *asin* | x | Rezultat in radiani, in [-pi/2,pi/2] |
| *acos* | x | Rezultat in radiani, in [0,pi] |
| *atan* | x | Rezultat in radiani, in [-pi/2,pi/2] |
| *atan2* | x,y | Arctang(x/y) |
| *sinh,cosh,tanh* | x | X in radiani |
| *ceil* | x | Returneaza cel mai mic intreg>=x |
| *floor* | x | Returneaza cel mai mare intreg<=x |
| *frexp* | x | Calculeaza mantisa si exponentul |

Tabelul 7.1

```
#include <stdio.h>
#include <math.h>
void main(void) {
double x;
printf(“x   acos(x)  asin(x)   atan(x)\n”);
for(x = - 0.5;x<=0.5; x+=0.2)
printf(“%f  %f  %f  %f \n”, x, acos(x), asin(x), atan(x));}
```

Fig. 7.2

În tabelul 7_1 se prezintă informații sumare despre cele mai importante dintre acestea. Funcțiile matematice incluse pot fi grupate astfel:

1) **Funcții trigonometrice inverse**. Forma generală a prototipului acestor funcții este:

*double **nume_funcție**(double expresie),*

unde *nume_funcție* poate fi: *asin, atan, acos*

Excepție face *atan2*, care are prototipul:

*double atan2(double y, double x).*

```
#include <stdio.h>
#include <math.h>
void main(void) {
double numarator=10.0;
double numitor=3.0;
printf(“fmod(10,3) este %f \n”,fmod(numarator,numitor));}
```

Fig. 7.3

În exemplul din fig. 7.2 se calculează şi afişează valorile funcțiilor arccos, arcsin şi atan pentru argumentul funcției variind în intervalul [-0.5,0.5], cu pasul de incrementare 0.2.

2) **Funcții trigonometrice directe si hiperbolice**. Forma generală a prototipului acestor funcții este:

*double **nume_funcție**(double expresie),*

unde *nume_funcție* poate fi: *sin, tan, cos, sinh, tanh, cosh.*

3) **Funcții pentru rotunjirea numerelor reale în virgulă mobilă**. Prototipul lor este la fel cu cel al funcțiilor trigonometrice directe, *nume_funcție* putând să fie: *floor* si respectiv *ceil*.

4) **Funcție pentru determinarea restului real al împărțirii a două numere de tip double.**

Are prototipul : *double* ***fmod****(double x, double y)* si returnează restul real al împărţirii lui *x* la *y*.

Programul din fig. 7.3 afişează restul real al împărţirii numărului real 10.0 la numărul real 3.0. Programul afişează: *fmod(10,3) este 1.000000*

```
#include <stdio.h>
#include <math.h>
void main(void) {
double val=1.2345;
double parte_intreaga;
double fract;
fract=modf(val,&parte_intreaga);
printf("valoare  %f, parte intreaga %f parte fractionara %f\n",val, parte_intreaga, fract);}
```

Fig. 7.4

5) **Funcţie pentru determinarea părţii întregi si respectiv a celei fracţionale dintr-un număr real**. Pentru a determina partea întreagă si respectiv fracţionară dintr-un număr real se poate utiliza funcţia *modf*, care are prototipul:

*double* ***modf****(double x, double *ipart)*

Efectul execuţiei funcţiei este următorul: returnează partea fracţionară si memorează partea întreagă în **ipart*.

Atunci când se execută programul din fig. 7.4, se afişează:

*valoare 1.234500 parte intreaga 1.000000 parte fractionara 0.234500*

6) **Funcţii pentru lucrul cu funcţia exponenţială si cu logaritmi**.

Forma generală a prototipului acestor funcţii este:

*double* ***nume_funcţie****(double expresie)*, unde *nume_funcţie* poate fi: *exp*, *log* si *log10*.

```
#include <stdio.h>
#include <math.h>
void main(void) {
float val;
printf(" argument   exponentiala\n");
for(val = 0.0; val<=1.0; val+=0.1)
printf ("%f  %f \n", val, exp(val));}
```

Fig. 7.5

Programul din fig. 7.5 calculează şi afişează $e^x$ pentru x parcurgând domeniul [0,1], cu pasul de 0.1.

7) **Funcţii pentru lucrul cu puteri**

Funcţia *pow* returnează rezultatul evaluării expresiei $x^N$ . Prototipul este:

*double pow(double x, double N)*

Funcţia *pow10* returnează rezultatul evaluării expresiei $10^x$. Prototipul este: *double pow10(double x)*

```
#include <stdio.h>
#include <math.h>
void main(void) {
int putere;
printf(" p    2 la puterea p\n");
for(putere = -1; putere<=2; putere++)
printf ("%d  %f \n", putere, pow(2.0,putere));
printf(" p    10 la puterea p\n");
for(putere = -1; putere<=2; putere++)
printf ("%d  %f \n", putere, pow10(putere));}
```

Fig. 7.6

Programul din fig. 7.6 calculează şi apoi afişează puterile întregi ale lui 2, respectiv 10, puterile fiind cuprinse în plaja de valori [-1,2].

8) **Funcţii pentru calculul modulelor**

Valoarea absolută (modulul) unei expresii de tip întreg se calculează folosind funcţia *abs*, implementată în biblioteca *stdlib.h*.

Prototipul funcţiei este : *int abs(int expresie)*.

În *math.h* este implementată si funcţia *labs*, care primeste si returnează valori de tip *longint*.

```
#include <stdio.h>
#include <math.h>
void main(void) {
float val;
printf(" argument    modul\n");
for(val = - 1.0; val<=1.0; val+=0.1)
     printf ("%f  %f \n", val, fabs(val));}
```

Fig. 7.7

Pentru modulul unei expresii reale se utilizează *fabs* (din *math.h*) , cu prototipul:

*float fabs(float expresie)*

În programul din fig. 7.7 se calculează şi afişează valorile absolute ale numerelor reale din domeniul [-1,1], parcurs cu pasul 0.1.

*Obţinerea modulului unui număr complex*

În biblioteca *complex.h* se defineste structura *complex* cu ajutorul căreia se poate reprezenta un număr complex. Prin funcţia *complex* acesta poate fi creat, dacă funcţia primeste ca parametri partea sa reală si respectiv pe cea imaginară. Modulul unui număr complex se poate calcula folosind funcţia *abs* din biblioteca *complex.h* , cu prototipul: *double abs(struct complex valoare)* .

```
#include <stdio.h>
#include <math.h>
void main(void) {
struct complex numar_complex;
numar_complex=complex(3,4);
printf("modulul lui (3,4) este %f\n", abs(numar_complex));}
```

Fig. 7.8

În fig. 7.8 se prezintă un exemplu de utilizare a acestei funcţii. După executarea programului, se va afişa: *modulul lui (3,4) este 5*.

```
#include <stdio.h>
#include <math.h>
void main(void) {
double val;
for(val = 0.0; val<=10.0; val+=0.1)
printf ("valoarea %f  radacina patrata %f \n", val, sqrt(val));}
```

Fig. 7.9

9) **Calculul rădăcinii pătrate a unei valori reale** Funcţia *sqrt*, implementată în *math.h*, se utilizează pentru extragerea rădăcinii pătrate a valorii precizate ca parametru. Prototipul funcţiei este: *double sqrt(double val);*

În fig. 7.10 se prezintă un program utilizat pentru calcularea şi afişarea rădăcinilor pătrate ale numerelor din [0,10], echidistanţate printr-un pas de 0.1.

10) **Calculul valorii unui polinom** este posibil datorită funcţiei *poly*, cu prototipul: *double poly(double x,int grad,double coeficienţi[])*. Această funcţie returnează valoarea polinomului precizat prin coeficienţii si gradul său, pentru valoarea argumentului *x*.