PROGRAMARE

Conf. univ. dr. COSTEL BĂLCĂU

2020

Tematica

Evaluare

- Activitate laborator: 30%
- Teme de casă: 20%
- Examen final: 50% (Probă de laborator)

Bibliografie

[1] A.V. Aho, J.E. Hopcroft, J.D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, Massachusetts, 2009.

[2] Gh. Barbu, V. Păun, *Programarea în limbajul C/C++*, Editura Matrix Rom, București, 2011.

[3] Gh. Barbu, V. Păun, *Calculatoare personale și programare în C/C++*, Editura Didactică și Pedagogică, București, 2005.

[4] Gh. Barbu, I. Văduva, M. Bolosteanu, *Bazele informaticii*, Editura Tehnică, București, 1997.

[5] C. Bălcău, *Combinatorică și teoria grafurilor*, Editura Universității din Pitești, Pitești, 2007.

[6] E. Cerchez, M. Șerban, *Programarea în limbajul C/C++ pentru liceu. Vol. 2: Metode și tehnici de programare*, Ed. Polirom, Iași, 2005.

[7] T.H. Cormen, *Algorithms Unlocked*, MIT Press, Cambridge, 2013.

[8] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, *Introduction to Algorithms*, MIT Press, Cambridge, 2009.

[9] H. Georgescu, *Tehnici de programare*, Editura Universității din București, București, 2005.

[10] C.A. Giumale, *Introducere în analiza algoritmilor. Teorie și aplicații*, Ed. Polirom, Iași, 2004.

[11] D.E. Knuth, *The Art Of Computer Programming. Vol. 4A: Combinatorial Algorithms*, Addison-Wesley, Massachusetts, 2011.

[12] L. Livovschi, H. Georgescu, *Sinteza și analiza algoritmilor*, Editura Științifică și Enciclopedică, București, 1986.

[13] D. Logofătu, *Algoritmi fundamentali în C++: Aplicații*, Ed. Polirom, Iași, 2007.

[14] D. Lucanu, M. Craus, *Proiectarea algoritmilor*, Ed. Polirom, Iași, 2008.

[15] D.A. Popescu, *Bazele programării - Java după C++*, `ebooks.infobits.ro`, 2019.

[16] R. Sedgewick, P. Flajolet, *An Introduction to the Analysis of Algorithms*, Addison-Wesley, New Jersey, 2013.

[17] R. Sedgewick, K. Wayne, *Algorithms*, Addison-Wesley, Massachusetts, 2011.

[18] R. Stephens, *Essential Algorithms: A Practical Approach to Computer Algorithms*, Wiley, Indianopolis, 2013.

[19] ***, *Revista MATINF*, editată de Departamentul de Matematică-Informatică, Universitatea din Pitești, Editura Universității din Pitești, 2018-2020, `http://matinf.upit.ro`.

Tema 1

Algoritmi

1.1 Noțiunea de algoritm

Definiția 1.1.1. *Un **algoritm** pentru rezolvarea unei probleme este o secvență finită de **propoziții** (**declarații** ale datelor utilizate și **instrucțiuni** aplicabile acestor date) prin aplicarea cărora se trece de la **informația inițială** (**datele de intrare**, **parametrii problemei**) la **informația finală** (**datele de ieșire**, **rezultatele problemei**), cu respectarea următoarelor **caracteristici**:*

1) ***generalitatea**: algoritmul este aplicabil pentru orice set de valori ale datelor de intrare, compatibile cu problema;*

2) ***unicitatea** (**neambiguitatea**): pentru orice set fixat de valori ale datelor de intrare (din domeniul de aplicabilitate al problemei) succesiunea instrucțiunilor aplicate este univoc determinată de algoritm (adică nu depinde de alți factori precum locul sau momentul aplicării);*

3) ***finitudinea**: pentru orice set de valori ale datelor de intrare aplicarea algoritmului se termină după un număr finit de pași, adică după executarea unei succesiuni finite de instrucțiuni.*

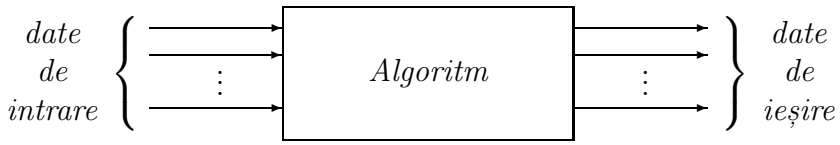


Figura 1.1.1: O ilustrare grafică a noțiunii de algoritm, privit ca o cutie neagră (din punct de vedere al instrucțiunilor)

1.2 Modalități de descriere a algoritmilor

Principalele **modalități de descriere a algoritmilor** sunt:

- **limbajul natural**: instrucțiunile sunt exprimate, concis, în limba română;
- **schema logică**: diagrame ce ilustrează grafic succesiunea instrucțiunilor;
- **limbajele de tip pseudocod**: descrieri simplificate apropiate de (intermediare pentru) limbajele de programare;
- **limbajele de programare**: instrucțiunile sunt efectuate (executate) de calculator; un algoritm descris într-un limbaj de programare se numește **program**.

Exemplul 1.2.1. Vom descrie algoritmul binecunoscut pentru rezolvarea ecuației de gradul al doilea

$$ax^2 + bx + c = 0,\ a, b, c \in \mathbb{R},\ a \neq 0,\ x \in \mathbb{R}.$$

Datele de intrare sunt a, b, c.
Datele de ieșire sunt soluțiile reale ale ecuației.

- Descrierea în limbajul natural (matematic):
  - Se calculează determinantul $\Delta = b^2 - 4ac$.
  - Dacă $\Delta > 0$ ecuația are două soluții reale

$$x_1 = \frac{-b + \sqrt{\Delta}}{2a},\ x_2 = \frac{-b - \sqrt{\Delta}}{2a}.$$

- Dacă $\Delta = 0$ ecuația are o soluție reală $x_1 = x_2 = -\dfrac{b}{2a}$.
- Dacă $\Delta < 0$ ecuația nu are soluții reale.

- Schema logică:

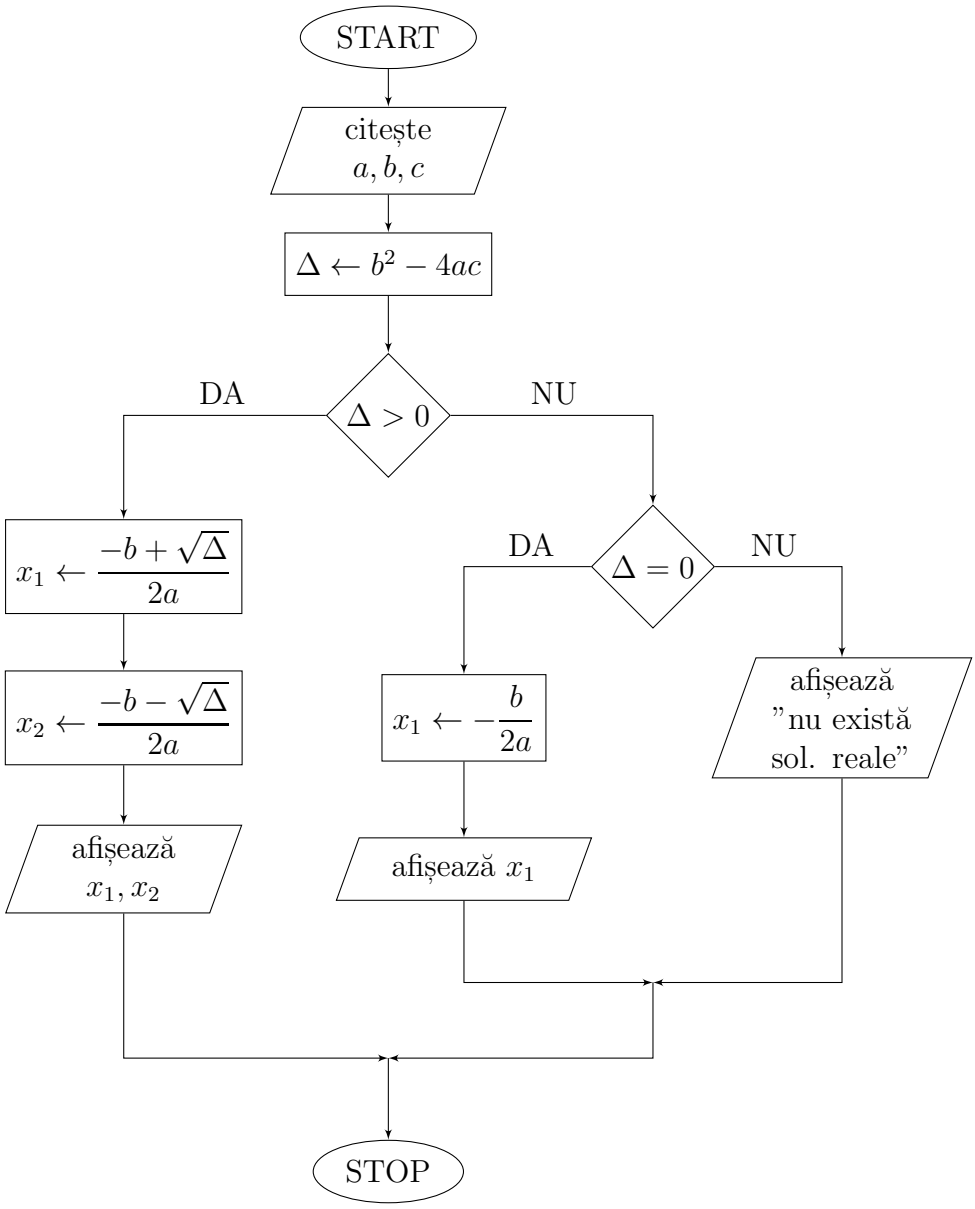


Figura 1.2.1:

- Pseudocod:

Date de intrare: a, b, c;
Date de ieșire: x_1, x_2;
Variabile: $a, b, c, x_1, x_2, \Delta \in \mathbb{R}$;
citește a, b, c;
$\Delta \leftarrow b^2 - 4ac$;
dacă $\Delta > 0$ **atunci**
  $x_1 \leftarrow \dfrac{-b + \sqrt{\Delta}}{2a}$;
  $x_2 \leftarrow \dfrac{-b - \sqrt{\Delta}}{2a}$;
  afișează x_1, x_2;
altfel
  dacă $\Delta = 0$ **atunci**
    $x_1 \leftarrow -\dfrac{b}{2a}$;
    afișează x_1;
  altfel
    afișează "ecuația nu are soluții reale";

- Programul corespunzător în limbajul de programare *C++*:

```
#include<iostream>
#include<math.h>
using namespace std;
int main(void)
{ float a,b,c,delta,x1,x2;
  cin>>a>>b>>c;
  delta=b*b-4*a*c;
  if (delta>0)
  { x1=(-b+sqrt(delta))/(2*a);
    x2=(-b-sqrt(delta))/(2*a);
    cout<<"x1="<<x1<<" x2="<<x2;
  }
  else
    if (delta==0)
    { x1=-b/(2*a);
      cout<<"x1="<<x1;
    }
    else cout<<"Ecuatia nu are solutii reale";
  return 0;
}
```

1.3 Analiza algoritmilor

Definiția 1.3.1. *Un algoritm este* ***corect*** *dacă pentru orice set de valori ale datelor de intrare (din domeniul de aplicabilitate) rezultatele obținute la terminarea aplicării algoritmului sunt corecte.*

Definiția 1.3.2. *a) Verificarea corectitudinii rezultatelor, în ipoteza că algoritmul se încheie după un număr finit de pași, reprezintă* ***problema corectitudinii parțiale****.*

b) Verificarea atât a corectitudinii parțiale, cât și a finitudinii algoritmului, reprezintă ***problema corectitudinii totale****.*

Observația 1.3.1. Corectitudinea invocată în definițiile anterioare este denumită și **corectitudine semantică**, pentru deosebirea de **corectitudinea sintactică**, aceasta din urmă însemnând respectarea regulilor de scriere ale limbajului folosit la descrierea algoritmului.

Definiția 1.3.3. ***Analiza performanțelor unui algoritm*** *constă în:*

1) Estimarea ***resurselor de calcul****, în funcție de* ***dimensiunea datelor de intrare****:*

- *Calculul* ***necesarului de resurse hardware*** *(memorie internă, externă etc.).*
- *Evaluarea* ***timpului de execuție****, adică a numărului de operații elementare (primitive) care se execută (operații aritmetice, operații logice, comparații, atribuiri etc.). În urma acestei evaluări se obține* ***ordinul de complexitate al algoritmului*** *(****complexitatea algoritmului****).*

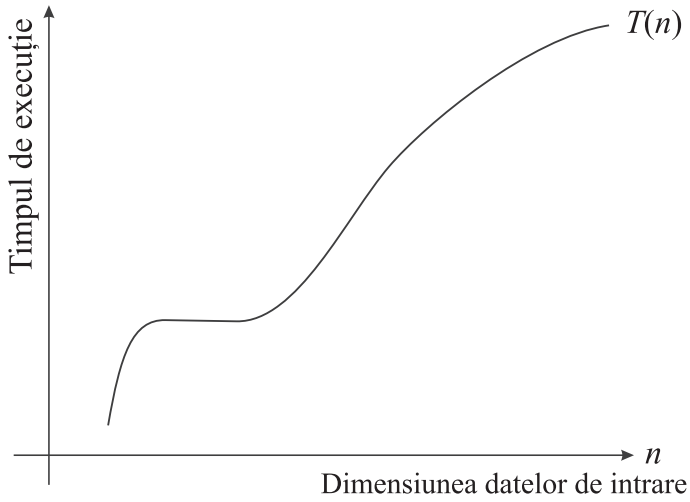


Figura 1.3.1: Complexitatea temporală a unui algoritm (cum depinde *timpul de execuție* de *dimensiunea datelor de intrare*)

2) *Studiul* ***eficienței algoritmului****, în comparație cu alți algoritmi care rezolvă aceeași problemă. Dacă un algoritm utilizează mai puține resurse de calcul (resurse hardware sau timp de execuție) decât un altul, atunci el este considerat ca fiind* ***mai eficient****.*

   Studiul eficienței poate eventual justifica ***optimalitatea algoritmului*** *(dacă este cazul). Un algoritm este considerat* ***optim*** *dacă se demonstrează că nu poate exista un algoritm mai eficient decât el, pentru rezolvarea problemei date.*

Observația 1.3.2. **Dimensiunea datelor de intrare** poate fi exprimată prin:

- numărul de componente (valori reale, valori întregi, caractere etc.) ale datelor de intrare;
- numărul de octeți sau de biți necesari stocării datelor de intrare.

Observația 1.3.3. Ideal este ca un algoritm să folosească resurse hardware cât mai puține și să necesite un timp de execuție cât mai mic.

De cele mai multe ori, însă, micșorarea necesarului de resurse hardware face ca timpul de execuție să crească și, invers, minimizarea timpului de execuție poate duce la o creștere considerabilă a necesarului de resurse hardware.

Exemplul 1.3.1. Calculul produsului a două numere complexe $z_1, z_2 \in \mathbb{C}$. Fie

$$\begin{aligned} z_1 &= a_1 + b_1 \cdot i, \\ z_2 &= a_2 + b_2 \cdot i, \text{ cu } a_1, a_2, b_1, b_2 \in \mathbb{R}, \end{aligned}$$

sau, altfel scris,

$$\begin{aligned} z_1 &= (a_1, b_1), \\ z_2 &= (a_2, b_2). \end{aligned}$$

Atunci

$$\begin{aligned} z_1 \cdot z_2 &= (a_1 + b_1 \cdot i) \cdot (a_2 + b_2 \cdot i) = \\ &= \underbrace{a_1 \cdot a_2 - b_1 \cdot b_2}_{\text{partea reală}} + \underbrace{(a_1 \cdot b_2 + a_2 \cdot b_1)}_{\text{partea imaginară}} \cdot i, \end{aligned}$$

sau, altfel scris,

$$z_1 \cdot z_2 = (\underbrace{a_1 \cdot a_2 - b_1 \cdot b_2}_{p}, \underbrace{a_1 \cdot b_2 + a_2 \cdot b_1}_{q}).$$

Varianta 1:
Date de intrare: a_1, b_1, a_2, b_2;
Date de ieșire: p, q;
Variabile: $a_1, b_1, a_2, b_2, p, q, t_1, t_2 \in \mathbb{R}$;
citește a_1, b_1, a_2, b_2;
$t_1 \leftarrow a_1 \cdot a_2$;
$t_2 \leftarrow b_1 \cdot b_2$;
$p \leftarrow t_1 - t_2$;
$t_1 \leftarrow a_1 \cdot b_2$;
$t_2 \leftarrow a_2 \cdot b_1$;
$q \leftarrow t_1 + t_2$;
afișează p, q;

Analiza algoritmului:

a) resurse hardware: $4 + 2 + 2 = 8$ locații de memorie;

b) operații efectuate:

- 4 înmulțiri;
- 2 adunări și scăderi;
- 6 atribuiri;

Varianta 2: Rescriem expresiile pentru părțile reală și imaginară astfel:

$$\begin{aligned} p &= a_1 \cdot a_2 - b_1 \cdot b_2 = (a_1 + b_1) \cdot a_2 - b_1 \cdot (a_2 + b_2), \\ q &= a_1 \cdot b_2 + a_2 \cdot b_1 = (a_1 + b_1) \cdot a_2 + a_1 \cdot (b_2 - a_2). \end{aligned}$$

Date de intrare: a_1, b_1, a_2, b_2;
Date de ieșire: p, q;
Variabile: $a_1, b_1, a_2, b_2, p, q, t_1, t_2, t_3 \in \mathbb{R}$;
citește a_1, b_1, a_2, b_2;
$t_1 \leftarrow a_1 + b_1$;
$t_2 \leftarrow t_1 \cdot a_2$;
$t_1 \leftarrow a_2 + b_2$;
$t_3 \leftarrow b_1 \cdot t_1$;
$p \leftarrow t_2 - t_3$;
$t_1 \leftarrow b_2 - a_2$;
$t_3 \leftarrow a_1 \cdot t_1$;
$q \leftarrow t_2 + t_3$;
afișează p, q;

Analiza algoritmului:

a) resurse hardware: $4 + 2 + 3 = 9$ locații de memorie;

b) operații efectuate:

- 3 înmulțiri;
- 5 adunări și scăderi;
- 8 atribuiri;

Observăm că Varianta 2 utilizează mai puține înmulțiri, dar mai multe locații de memorie, mai multe adunări și scăderi și mai multe atribuiri.

Observația 1.3.4. Pentru simplificarea calculelor, de cele mai multe ori ne vom concentra atenția numai asupra anumitor operații, semnificative pentru algoritmii respectivi, rezumându-ne astfel numai la estimarea (numărarea) acestora.

Tema 2

Limbajul pseudocod

2.1 Descrierea limbajului

Definiția 2.1.1. *Orice **limbaj** pentru descrierea algoritmilor este caracterizat prin trei elemente:*

1) ***alfabet**: mulțimea caracterelor utilizate;*
2) ***vocabular**: mulțimea unităților lexicale (cuvinte) utilizate;*
3) ***sintaxă**: mulțimea regulilor de folosire a unităților lexicale (cuvintelor) pentru a forma propoziții (declarații și instrucțiuni) corecte.*

Alfabetul limbajului

Caracterele limbajului pseudocod sunt:

- *litere mari*: A, B, ..., Z;
- *litere mici*: a, b, ..., z;
- *cifre*: 0, 1, ..., 9;
- *simboluri speciale*:
  - *operatori*:
    - *aritmetici*: $+, -, \times, \cdot, /, -, \sqrt{\ }$;
    - *relaționali*: $=, \neq, <, >, \leq, \geq, \in$;
    - *de atribuire*: $\leftarrow$;
  - *delimitatori*: $(,), [,], \{, \}, \lceil, \rceil$, , , ; , ., :, ′ , ”, |, //;
  - *simboluri matematice*: $\mathbb{N}, \mathbb{Z}, \mathbb{R}$.

Date

După *semnificația* lor, datele pot fi:

- *de intrare*;
- *de ieșire*;
- *de lucru* (*intermediare*).

După *tipul* lor, datele pot fi:

- *simple*:
  - *reale* (de exemplu 12.5 sau -13.26);
  - *întregi* (de exemplu 120 sau -326);
  - *logice* (*booleene*): adevăr (1), fals (0);
  - *caractere*: orice caracter cuprins între apostrofuri (de exemplu 'a' sau '6');
- *compuse*:
  - *șiruri de caractere* (*stringuri*): orice șir de caractere cuprins între ghilimele (de exemplu "maximul este" sau "Ecuația dată nu are soluții.");
  - *tablouri*:
    - *vectori* (*tablouri unidimensionale*), de exemplu

$$v = (v_1, v_2, \ldots, v_n);$$

    - *matrice* (*tablouri bidimensionale*), de exemplu

$$A = \begin{pmatrix} A_{11} & A_{12} & \ldots & A_{1n} \\ A_{21} & A_{22} & \ldots & A_{2n} \\ \ldots & & & \\ A_{m1} & A_{m2} & \ldots & A_{mn} \end{pmatrix};$$

    - *tablouri k-dimensionale* ($k \in \mathbb{N}^*$), de forma

$$A = (a_{i_1 i_2 \ldots i_k})_{i_1 \in I_1, i_2 \in I_2, \ldots, i_k \in I_k},$$

      unde $I_1, I_2, \ldots, I_k$ sunt mulțimi de indici.

După *natura* lor, datele pot fi:

- *constante* (au valori nemodificabile);
- *variabile* (au valori modificabile). Orice variabilă este identificată printr-un *nume* (*identificator*).

Operații și *funcții*:

- pentru datele reale:
  - *aritmetice*:
    - *operații*: $+, -$ (scădere), $\times, \cdot, /, -$ (împărțire);
      De exemplu, $32.0/5 = \dfrac{32.0}{5} = 6.4$.
    - *funcții*: $|x|$, $[x]$ (partea întreagă a numărului x), $\lceil x \rceil$ (partea întreagă superioară a numărului x), $\sqrt{x}$, e^x, $\ln x$, $\sin x$, $\cos x$, $\operatorname{tg} x$, $\arcsin x$, $\arccos x$, $\operatorname{arctg} x$;
  - *relaționale*: $=, \neq, <, >, \leq, \geq$;
- pentru datele întregi:
  - cele de la datele reale:
  - *DIV* și *MOD*:
    - x DIV y reprezintă câtul împărțirii lui x la y;
    - x MOD y reprezintă restul împărțirii lui x la y;
      De exemplu, 32 DIV 5 = 6 și 32 MOD 5 = 2.
- pentru datele logice: *non* (negația, $\neg$), *și* (conjuncția, $\wedge$), *sau* (disjuncția, $\vee$):

$$\text{non } 0 = 1,\ \text{non } 1 = 0;$$

$$0 \text{ și } 0 = 0,\ 0 \text{ și } 1 = 0,\ 1 \text{ și } 0 = 0,\ 1 \text{ și } 1 = 1;$$

$$0 \text{ sau } 0 = 0,\ 0 \text{ sau } 1 = 1,\ 1 \text{ sau } 0 = 1,\ 1 \text{ sau } 1 = 1;$$

- pentru datele șiruri de caractere: + (concatenarea, alipirea). De exemplu, "Ion"+"Ela"+" Gigel"="IonEla Gigel".

Vocabularul limbajului

Unitățile lexicale ale limbajului pseudocod sunt:

- *identificatorii*: nume date variabilelor, constantelor, funcțiilor, declarațiilor, instrucțiunilor. Ei pot fi:

  - *standard* (*predefiniți*): **Date de intrare**, **Date de ieșire**, **Variabile**, **Constante**, **caractere**, **șir de caractere** (**stringuri**), **funcția**, **citește**, **afișează** (**scrie**), **pentru**, **execută**, **dacă**, **atunci**, **altfel**, **cât timp**, **repetă**, **returnează**, adevăr, fals, ln, sin, cos, tg, arcsin, arccos, arctg, DIV, MOD, non, și, sau;
  - *definiți de programator*;
- *constantele*;
- *operatorii*.

Sintaxa limbajului

Separatorii limbajului pseudocod, utilizați pentru separarea unităților lexicale, sunt:

- *spațiul*;
- *trecerea la linie nouă*;
- *delimitatorii*;
- *comentariile*: texte doar cu rol explicativ, precedate de o pereche de simboluri `//` (*dublu slash*).

Propozițiile limbajului pseudocod sunt:

- *declarații* ale datelor și funcțiilor utilizate;
- *instrucțiuni* ce operează cu și asupra acestor date și funcții.

Structura unui algoritm în limbajul pseudocod este următoarea:

```
⋮              // declararea datelor de intrare și de ieșire
                // declararea variabilelor și a constantelor
                                  // definite de programator
⋮                    // declararea și construirea de funcții
                                  // definite de programator
⋮                                              // instrucțiuni
```

Declararea datelor de intrare și de ieșire are forma:

```
Date de intrare: ...              // lista datelor de intrare
Date de ieșire: ...                // lista datelor de ieșire
```

Componentele unei liste sunt separate prin virgulă.

Exemplul 2.1.1.

Date de intrare: a, b, c;
Date de ieșire: x_1, x_2;

Declararea variabilelor și constantelor are forma:

Variabile: ... `// lista variabilelor, // cu precizarea tipului lor`
Constante: ... `// lista constantelor, // cu precizarea valorilor lor, // sub forma` *nume*`=`*valoare*

Exemplul 2.1.2.

Constante: $DIMMAX = 100000$
$PI = 3.1415$
Variabile: $m, n \in \mathbb{Z}$
$x, y, z \in \mathbb{R}$
$v = (v_1, \ldots, v_n) \in \mathbb{R}^n$

Declararea și construirea (definirea) unei funcții are forma:

Funcția *nume*(*lista parametrilor*) **:** `// antetul funcției // (declararea funcției)`

⋮ `// corpul funcției (construirea funcției), // ce conține declararea de variabile și instrucțiuni`

Dacă funcția calculează o valoare (rezultat), atunci în antetul funcției se precizează și tipului acesteia. În acest caz valoarea finală calculată este specificată prin instrucțiunea

returnează *rezultat*.

Observația 2.1.1.

Executarea instrucțiunii **returnează** implică încheierea executării funcției în care este utilizată!

Exemplul 2.1.3. Următoarea funcție calculează modulul unui număr complex $z = a + b \cdot i,\ a, b \in \mathbb{R}$, conform formulei

$$|z| = |a + b \cdot i| = \sqrt{a^2 + b^2}.$$

Funcția $MODUL(a, b) \in \mathbb{R}$ **:** `//` $a, b \in \mathbb{R}$
Variabile: $r \in \mathbb{R}$; `//` $r = |a + b \cdot i|$
$r \leftarrow \sqrt{a^2 + b^2}$;
returnează r;

Instrucțiunile limbajului pseudocod sunt:

- *instrucțiunea de citire*;
- *instrucțiunea de scriere (afișare)*;
- *instrucțiunea de atribuire (calcul)*;
- *instrucțiunea de decizie (selecție)*;
- *instrucțiunea repetitivă cu contor (cu număr fixat de pași)*;
- *instrucțiunea repetitivă cu test inițial*;
- *instrucțiunea repetitivă cu test final*;
- *instrucțiunea* ***returnează***.

Descriem în continuare aceste instrucțiuni.

2.2 Instrucțiunea de citire

- *Sintaxa*:

  citește *listă de variabile*

  Variabilele din listă sunt separate prin virgulă.

- *Efect*: se introduc valori pentru variabilele din listă, respectând ordinea lor în cadrul listei. Pentru fiecare variabilă, valorea introdusă trebuie să fie compatibilă cu tipul variabilei.

Exemplul 2.2.1. Considerăm următorul algoritm.
Variabile: $n \in \mathbb{Z},\ x \in \mathbb{R}$;
citește n, x;

Introducerea valorilor 10 30 (la executarea instrucțiunii **citește** n, x) conduce la $n = 10,\ x = 30$.
Introducerea valorilor 30 10 conduce la $n = 30,\ x = 10$.
Introducerea valorilor 8 5.32 conduce la $n = 8,\ x = 5.32$.
Introducerea valorilor 5.32 8 este incorectă, deoarece ar conduce la $n = 5.32$, imposibil, deoarece $n \in \mathbb{Z}$.

Observația 2.2.1. Instrucțiunea de citire se folosește cu precădere la introducerea (citirea) valorilor inițiale ale datelor de intrare, cel mai adesea la începutul algoritmului.

2.3 Instrucțiunea de scriere (afișare)

- *Sintaxa*:

  afișează *listă de expresii* `// sau` **scrie** *listă de expresii*

  Expresiile din listă sunt separate prin virgulă.

- *Efect*: se afișează valorile curente ale expresiilor din listă, respectând ordinea lor în cadrul listei.

Observația 2.3.1. Instrucțiunea de scriere (afișare) se folosește cu precădere la afișarea (scrierea) valorilor finale ale datelor de ieșire, cel mai adesea la sfârșitul algoritmului.

Astfel adesea algoritmii au următoarea ***formă simplificată***:

... `// declararea datelor și funcțiilor utilizate`
citește *lista variabilelor de intrare*;
⋮ `// calculul datelor de ieșire (rezultatelor)`
afișează *lista datelor de ieșire*;

2.4 Instrucțiunea de atribuire (calcul)

- *Sintaxa*:

  variabilă $\leftarrow$ *expresie*

- *Efect*: se calculează valoarea expresiei (pentru valorile curente ale variabilelor componente) și această valoare este atribuită variabilei, adică variabila primește drept valoare valoarea calculată a expresiei.

  Tipul expresiei trebuie să fie compatibil cu tipul variabilei.

  Vechea valoare a variabilei se pierde.

Exemplul 2.4.1. Pentru următorul algoritm, efectul fiecărei instrucțiuni este scris la comentarii.

Variabile: $x, y, z \in \mathbb{Z}$;
$x \leftarrow 100$; `//` $x = 100$
$y \leftarrow 8$; `//` $y = 8$
$z \leftarrow 2 \cdot x + 3 \cdot y$; `//` $z = 2 \cdot 100 + 3 \cdot 8 = 224$
$x \leftarrow x - 10$; `//` $x = 100 - 10 = 90$
$z \leftarrow z + 1$; `//` $z = 224 + 1 = 225$
$y \leftarrow x - z - y$; `//` $y = 90 - 225 - 8 = -143$
afișează x, y, z; `// se afișează valorile` $90, -143, 225$

Exemplul 2.4.2. Algoritmul următor conține instrucțiuni incorecte (evidențiate la comentarii), deci este incorect sintactic.

Variabile: $x, y, z, t \in \mathbb{R},\ a, b, c, d \in \mathbb{Z}$;
$x \leftarrow 8$; // $x = 8$
$y \leftarrow 8.5$; // $y = 8.5$
$a \leftarrow 8$; // $a = 8$
$b \leftarrow 8.5$; // instrucțiune incorectă, deoarece $b \in \mathbb{Z}$, // iar $8.5 \in \mathbb{R}$
$z \leftarrow a$; // $z = 8$
$c \leftarrow x$; // instrucțiune incorectă, deoarece $c \in \mathbb{Z}$, // iar $x \in \mathbb{R}$ (***tipul*** variabilei x este real!)
$t \leftarrow a + x$; // $t = 8 + 8 = 16$
$d \leftarrow a + x$; // instrucțiune incorectă, deoarece $d \in \mathbb{Z}$, // iar $a + x \in \mathbb{R}$ (***tipul*** expresiei $a + x$ este real!)

Exemplul 2.4.3. Calculul radicalului dintr-un număr întreg n, $n \geq 0$.

Următorul algoritm conține o eroare sintactică.

Date de intrare: n;
Date de ieșire: r; // $r = \sqrt{n}$
Variabile: $n, r \in \mathbb{Z}$;
citește n; // se introduce valoarea lui n
$r \leftarrow \sqrt{n}$; // instrucțiune incorectă, deoarece $r \in \mathbb{Z}$, // iar $\sqrt{n} \in \mathbb{R}$ (***tipul*** expresiei $\sqrt{n}$ este real!)
afișează r;

Algoritmul devine corect dacă declarăm variabila r de tip real:

Date de intrare: n;
Date de ieșire: r; // $r = \sqrt{n}$
Variabile: $n \in \mathbb{Z},\ r \in \mathbb{R}$;
citește n; // se introduce valoarea lui n
$r \leftarrow \sqrt{n}$; // se calculează $r = \sqrt{n}$
afișează r; // se afișează valoarea calculată a lui r

Exemplul 2.4.4. Să se calculeze suma cifrelor unui număr natural de patru cifre $n \in \{1000, 1001, \ldots, 9999\}$ dat.

De exemplu, pentru $n = 5704$ suma este $S = 5 + 7 + 0 + 4 = 16$.

Pentru rezolvare utilizăm următoarele două proprietăți din aritmetică.

1. Ultima cifră a unui număr natural n este egală cu n MOD 10 (restul împărțirii lui n la 10);
2. Dacă $n \geq 10$, atunci numărul care se obține din n prin eliminarea ultimei cifre este egal cu n DIV 10 (câtul împărțirii lui n la 10).

Putem astfel calcula succesiv cifrele lui n, de la dreapta la stânga (de la ultima cifră până la prima). Obținem următorul algoritm pentru rezolvarea problemei date.

Date de intrare: n;
Date de ieșire: S;
Variabile: $n, S, m, c \in \mathbb{Z}$;
citește n; // se introduce valoarea lui n
$S \leftarrow 0$; // înainte de a însuma termenii doriți, valoarea // inițială a unei sume este 0; // spunem că *inițializăm variabila* S *cu* 0
$m \leftarrow n$; // $m =$ numărul curent, căruia îi calculăm // de fiecare dată ultima cifră c
$c \leftarrow m$ MOD 10; // $c =$ ultima cifră a lui $m = n$
$S \leftarrow S + c$; // o adunăm la S
$m \leftarrow m$ DIV 10; // eliminăm ultima cifră din m, // m rămâne de trei cifre
$c \leftarrow m$ MOD 10; // $c =$ ultima cifră a lui m, deci // penultima cifră a lui n
$S \leftarrow S + c$; // o adunăm la S
$m \leftarrow m$ DIV 10; // eliminăm din nou ultima cifră din m, // m rămâne de două cifre
$c \leftarrow m$ MOD 10; // $c =$ ultima cifră a lui m, deci // antepenultima cifră a lui n
$S \leftarrow S + c$; // o adunăm la S
$m \leftarrow m$ DIV 10; // eliminăm din nou ultima cifră din m, // m rămâne de o cifră, și anume $m =$ prima cifră a lui n
$S \leftarrow S + m$; // o adunăm la S
afișează S; // se afișează valoarea calculată a lui S

De exemplu, pentru $n = 5704$ efectul fiecărei instrucțiuni a algoritmului

anterior este descris în continuare, la comentarii.

citește n; // se introduce valoarea $n = 5704$
$S \leftarrow 0$; // $S = 0$
$m \leftarrow n$; // $m = 5704$
$c \leftarrow m$ MOD 10; // $c = 5704$ MOD $10 = 4$
$S \leftarrow S + c$; // $S = 0 + 4 = 4$
$m \leftarrow m$ DIV 10; // $m = 5704$ DIV $10 = 570$
$c \leftarrow m$ MOD 10; // $c = 570$ MOD $10 = 0$
$S \leftarrow S + c$; // $S = 4 + 0 = 4$
$m \leftarrow m$ DIV 10; // $m = 570$ DIV $10 = 57$
$c \leftarrow m$ MOD 10; // $c = 57$ MOD $10 = 7$
$S \leftarrow S + c$; // $S = 4 + 7 = 11$
$m \leftarrow m$ DIV 10; // $m = 57$ DIV $10 = 5$
$S \leftarrow S + m$; // $S = 11 + 5 = 16$
afișează S; // se afișează valoarea lui S, adică 16

Exemplul 2.4.5. Interschimbarea valorilor a două variabile a și b (având același tip).

Varianta 1: Cu memorie suplimentară, pentru o variabilă aux (**regula celor trei pahare**):

Date de intrare: a, b;
Date de ieșire: a, b;
Variabile: $a, b, aux \in \mathbb{R}$;
citește a, b;
$aux \leftarrow a$;
$a \leftarrow b$;
$b \leftarrow aux$;
afișează a, b;

Analiza algoritmului:

a) resurse hardware: 3 locații de memorie;

b) operații efectuate: 3 atribuiri.

Varianta 2: Fără memorie suplimentară:
Date de intrare: a, b;
Date de ieșire: a, b;
Variabile: $a, b \in \mathbb{R}$;
citește a, b;
$a \leftarrow a + b$;
$b \leftarrow a - b$;
$a \leftarrow a - b$;
afișează a, b;

Analiza algoritmului:

a) resurse hardware: 2 locații de memorie;

b) operații efectuate:

- 3 atribuiri;
- 3 adunări și scăderi.

Observăm că Varianta 2 utilizează mai puține locații de memorie, dar mai multe operații.

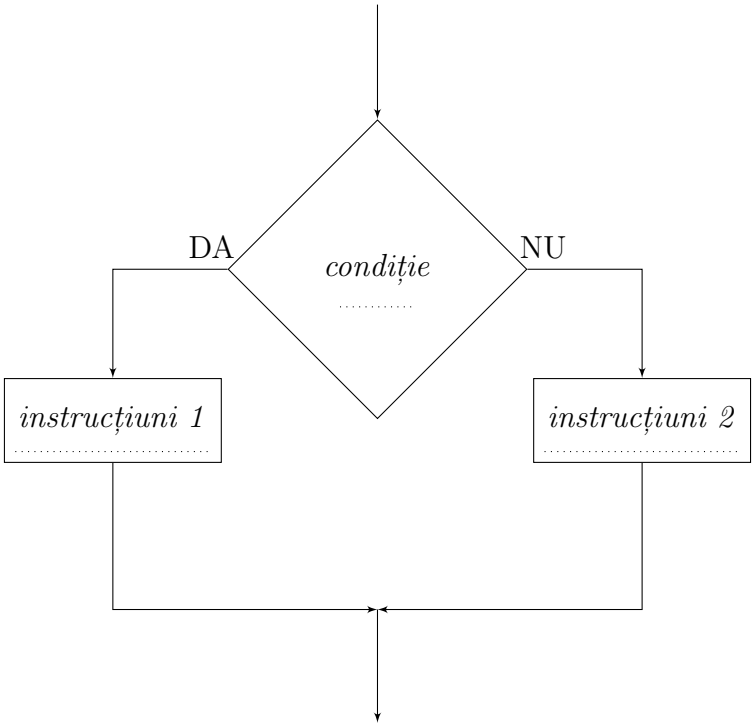
2.5 Instrucțiunea de decizie (selecție)

- *Sintaxa*:

dacă *condiție* **atunci**
| *instrucțiuni 1*
altfel
└ *instrucțiuni 2*

- *Efect*:
  - se evaluează condiția (pentru valorile curente ale variabilelor componente);
  - dacă ea este îndeplinită (este adevărată), atunci se execută *instrucțiunile 1* și se sare peste *instrucțiunile 2*;
  - în caz contrar, adică dacă ea nu este îndeplinită (este falsă), atunci se sare peste *instrucțiunile 1* și se execută *instrucțiunile 2*.

- *Schema logică*:



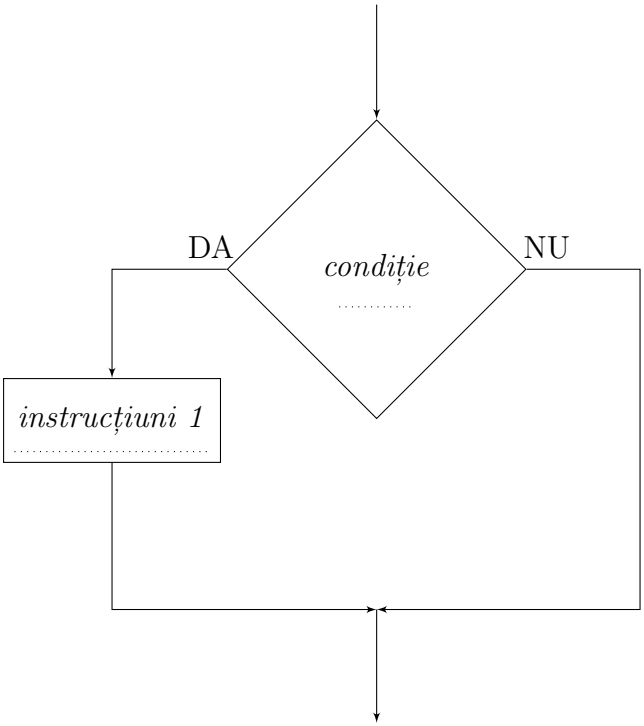
Observația 2.5.1. Ramura **altfel** este **opțională**, adică poate să lipsească. În acest caz avem:

- *Sintaxa*:

dacă *condiție* **atunci**
└ *instrucțiuni 1*

- *Efect*:
  - se evaluează condiția (pentru valorile curente ale variabilelor componente);
  - dacă ea este îndeplinită (este adevărată), atunci se execută *instrucțiunile 1*;
  - în caz contrar, adică dacă ea nu este îndeplinită (este falsă), atunci se sare peste *instrucțiunile 1*.

- *Schema logică*:



Exemplul 2.5.1. Să se calculeze valoarea funcției

$$f : \mathbb{R} \to \mathbb{R}, f(x) = \begin{cases} \sqrt{x^2 - 4}, & \text{dacă } x \geq 2, \\ \dfrac{\sin x + \cos x}{2}, & \text{dacă } x < 2 \end{cases}$$

într-un punct dat $x \in \mathbb{R}$.

Următorul algoritm rezolvă problema dată.

Date de intrare: x;
Date de ieșire: v; // $v = f(x)$
Variabile: $x, v \in \mathbb{R}$;
citește x; // se introduce valoarea lui x
dacă $x \geq 2$ **atunci**
  $v \leftarrow \sqrt{x \times x - 4}$;
altfel
  $v \leftarrow \dfrac{\sin x + \cos x}{2}$;
afișează v; // se afișează valoarea calculată a lui v

Exemplul 2.5.2. Să se calculeze valoarea expresiei

$$E(x,y)=\begin{cases}\dfrac{2x-3y}{x+y+1}, & \text{dacă } x \text{ și } y \text{ sunt pare,}\\ \ln(x^2+2y^2), & \text{dacă } x \text{ și } y \text{ sunt impare,}\\ \sqrt{2x^2+y^2-1}, & \text{dacă } x \text{ este par și } y \text{ este impar,}\\ \dfrac{x^3-4y}{xy-1}, & \text{dacă } x \text{ este impar și } y \text{ este par,}\end{cases}$$

unde $x, y \in \mathbb{Z}$ sunt numere date.

Următorul algoritm rezolvă problema dată.

Date de intrare: x, y;
Date de ieșire: E; // $E = E(x,y)$
Variabile: $x, y \in \mathbb{Z},\ E \in \mathbb{R}$;
citește x, y; // se introduc valorile lui x și y
dacă x *MOD* $2 = 0$ **atunci** // x este par
  dacă y *MOD* $2 = 0$ **atunci** // y este par
    $E \leftarrow \dfrac{2 \times x - 3 \times y}{x + y + 1}$;
  altfel // y este impar
    $E \leftarrow \sqrt{2 \times x \times x + y \times y - 1}$;
altfel // x este impar
  dacă y *MOD* $2 = 0$ **atunci** // y este par
    $E \leftarrow \dfrac{x \times x \times x - 4 \times y}{x \times y - 1}$;
  altfel // y este impar
    $E \leftarrow \ln(x \times x + 2 \times y \times y)$;
afișează E; // se afișează valoarea calculată a lui E

Exemplul 2.5.3. Să se calculeze ultima cifră a numărului a^n, unde a și n sunt două numere naturale nenule date.

Pentru rezolvare utilizăm următoarele două proprietăți din aritmetică.

1. Ultima cifră a numărului a^n este egală cu ultima cifră a numărului c^n, unde c este ultima cifră a lui a;

2. Ultima cifră a numărului c^n este egală cu ultima cifră a numărului c^k, unde

$$k=\begin{cases}4, & \text{dacă } n \text{ se divide cu } 4,\\ \text{restul împărțirii lui } n \text{ la } 4, & \text{în caz contrar.}\end{cases}$$

Obținem următorul algoritm pentru rezolvarea problemei date.

Date de intrare: a, n;
Date de ieșire: u; // $u =$ ultima cifră a numărului a^n
Variabile: $a, n, u, c, k \in \mathbb{Z}$;
citește a, n; // se introduc valorile lui a și n
$c \leftarrow a$ MOD 10; // $c =$ ultima cifră a lui a
$k \leftarrow n$ MOD 4; // $k =$ restul împărțirii lui n la 4
dacă $k = 0$ **atunci** // n se divide cu 4
  $u \leftarrow (c \times c \times c \times c)$ MOD 10; // $u =$ ultima cifră a lui c^4
altfel // n nu se divide cu 4,
  // deci calculăm $u =$ ultima cifră a lui c^k
  dacă $k = 1$ **atunci**
    $u \leftarrow c$ MOD 10;
  altfel
    dacă $k = 2$ **atunci**
      $u \leftarrow (c \times c)$ MOD 10;
    altfel // $k = 3$
      $u \leftarrow (c \times c \times c)$ MOD 10;
afișează u; // se afișează valoarea calculată a lui u

De exemplu, pentru $a = 2017$ și $n = 1098$ efectul fiecărei instrucțiuni a algoritmului anterior este descris în continuare, la comentarii.

citește a, n; // se introduc valorile $a = 2017$ și $n = 1098$
$c \leftarrow a$ MOD 10; // $c = 2017$ MOD $10 = 7$
$k \leftarrow n$ MOD 4; // $k = 1098$ MOD $4 = 2$
dacă $k = 0$ **atunci** // se testează condiția $2 = 0$, FALS
  $u \leftarrow (c \times c \times c \times c)$ MOD 10; // — (nu se execută)
altfel
  dacă $k = 1$ **atunci** // se testează condiția $2 = 1$, FALS
    $u \leftarrow c$ MOD 10; // — (nu se execută)
  altfel
    dacă $k = 2$ **atunci** // se testează condiția $2 = 2$, ADEV.
      $u \leftarrow (c \times c)$ MOD 10; // $u = (7 \times 7)$ MOD $10 = 9$
    altfel
      $u \leftarrow (c \times c \times c)$ MOD 10; // — (nu se execută)
afișează u; // se afișează valoarea lui u, adică 9

Exemplul 2.5.4. Se dau patru numere întregi a, b, c și d. Să se calculeze câte dintre acestea sunt pătrate perfecte.

Pentru rezolvare, folosim proprietatea:

$$x \in \mathbb{Z} \text{ este pătrat perfect } \Leftrightarrow \begin{cases} x \geq 0, \\ \sqrt{x} \in \mathbb{N} \end{cases} \Leftrightarrow \begin{cases} x \geq 0, \\ [\sqrt{x}] = \sqrt{x}. \end{cases}$$

Obținem următorul algoritm pentru rezolvarea problemei date.

Date de intrare: a, b, c, d;
Date de ieșire: nr; `// numărul de pătrate perfecte`
Variabile: $a, b, c, d, nr \in \mathbb{Z},\ r \in \mathbb{R}$;
citește a, b, c, d;
$nr \leftarrow 0$; `// inițializăm variabila` nr `cu 0`
dacă $a \geq 0$ **atunci** `// verificăm dacă` a `este pătrat perfect`
  $r \leftarrow \sqrt{a}$;
  dacă $[r] = r$ **atunci** `// ` a `este pătrat perfect,`
    $nr \leftarrow nr + 1$; `// deci mărim` nr `cu 1;`
    `// spunem că` *incrementăm variabila* nr
dacă $b \geq 0$ **atunci** `// procedăm analog cu` b`,`
  $r \leftarrow \sqrt{b}$;
  dacă $[r] = r$ **atunci**
    $nr \leftarrow nr + 1$;
dacă $c \geq 0$ **atunci** `// cu` c
  $r \leftarrow \sqrt{c}$;
  dacă $[r] = r$ **atunci**
    $nr \leftarrow nr + 1$;
dacă $d \geq 0$ **atunci** `// și cu` d
  $r \leftarrow \sqrt{d}$;
  dacă $[r] = r$ **atunci**
    $nr \leftarrow nr + 1$;
afișează nr;

Exemplul 2.5.5. Se dau trei numere reale a, b și c. Să se verifice dacă există un triunghi având lungimile laturilot egale cu a, b și, respectiv, c. În caz afirmativ, să se precizeze natura triunghiului (isoscel, echilateral, dreptunghic, oarecare).

Pentru rezolvare, utilizăm proprietățile:

$$\begin{aligned}
\text{există triunghi} &\Leftrightarrow \begin{cases} a, b, c > 0, \\ a + b > c,\ a + c > b,\ b + c > a; \end{cases} \\
\text{triunghiul este echilateral} &\Leftrightarrow a = b = c; \\
\text{triunghiul este isoscel} &\Leftrightarrow a = b \neq c \text{ sau } a = c \neq b \text{ sau } b = c \neq a; \\
\text{triunghiul este dreptunghic} &\Leftrightarrow a^2 + b^2 = c^2 \text{ sau } a^2 + c^2 = b^2 \text{ sau } b^2 + c^2 = a^2.
\end{aligned}$$

Pentru memorarea tipului triunghiului (în cazul în care acesta există), utilizăm o variabilă r având semnificația:

$$r = \begin{cases} 0, & \text{dacă triunghiul este oarecare,} \\ 1, & \text{dacă triunghiul este echilateral,} \\ 2, & \text{dacă triunghiul este dreptunghic și isoscel,} \\ 3, & \text{dacă triunghiul este doar dreptunghic,} \\ 4, & \text{dacă triunghiul este doar isoscel.} \end{cases}$$

Obținem următorul algoritm pentru rezolvarea problemei date.
Date de intrare: a, b, c;
Date de ieșire: r;
Variabile: $a, b, c \in \mathbb{R},\ r \in \mathbb{Z}$;
citește a, b, c;
// verificăm dacă există triunghi
dacă $a \leq 0\ sau\ b \leq 0\ sau\ c \leq 0\ sau\ a + b \leq c\ sau\ a + c \leq b\ sau\ b + c \leq a$
atunci
  afișează "nu există triunghi";
altfel // există triunghi
  $r \leftarrow 0$; // inițial, presupunem că este oarecare
  // verificăm dacă este echilateral
  dacă $a = b$ *și* $b = c$ **atunci** // este echilateral
    $r \leftarrow 1$;
  altfel // nu este echilateral
    // verificăm dacă este isoscel
    dacă $a = b\ sau\ a = c\ sau\ b = c$ **atunci** // este isoscel
      $r \leftarrow 4$;
    // verificăm dacă este dreptunghic
    dacă $a^2 + b^2 = c^2\ sau\ a^2 + c^2 = b^2\ sau\ b^2 + c^2 = a^2$ **atunci**
      // este dreptunghic
      // verificăm dacă este și isoscel
      dacă $r = 4$ **atunci** // este și isoscel
        $r \leftarrow 2$;
      altfel // este doar dreptunghic
        $r \leftarrow 3$;
afișează r;

Exemplul 2.5.6. Să se rezolve ecuația

$$\sin x = a,\ x \in [0, 2\pi],$$

unde a este un număr real dat.

Pentru rezolvare, avem următoarele cazuri:

1. Dacă $a \notin [-1, 1]$ ecuația nu are soluție.
2. Dacă $a = 0$ ecuația are soluțiile $x_1 = 0$, $x_2 = \pi$ și $x_3 = 2\pi$.
3. Dacă $a \in (0, 1)$ ecuația are soluțiile $x_1 = \arcsin a$ și $x_2 = \pi - \arcsin a$.
4. Dacă $a = 1$ ecuația are soluția $x = \dfrac{\pi}{2}$.

5. Dacă $a \in (-1, 0)$ ecuația are soluțiile $x_1 = \pi + \arcsin(-a)$ și $x_2 = 2\pi - \arcsin(-a)$.

6. Dacă $a = -1$ ecuația are soluția $x = \dfrac{3\pi}{2}$.

Obținem următorul algoritm pentru rezolvarea problemei date.

Date de intrare: a;
Date de ieșire: x_1, x_2, x_3; `// soluțiile ecuației`
Variabile: $a, x_1, x_2, x_3 \in \mathbb{R}$;
citește a; `// se introduce valoarea lui` a
dacă $|a| > 1$ **atunci** `// cazul` $a \notin [-1, 1]$
  afișează "ecuația nu are soluții";
altfel `// cazul` $a \in [-1, 1]$
  dacă $a = 0$ **atunci** `// cazul` $a = 0$
    $x_1 \leftarrow 0$;
    $x_2 \leftarrow \pi$;
    $x_3 \leftarrow 2\pi$;
    afișează x_1, x_2, x_3;
  altfel `// cazul` $a \in [-1, 1],\ a \neq 0$
    dacă $a > 0$ **atunci** `// cazul` $a \in (0, 1]$
      dacă $a = 1$ **atunci** `// cazul` $a = 1$
        $x_1 \leftarrow \dfrac{\pi}{2}$;
        afișează x_1;
      altfel `// cazul` $a \in (0, 1)$
        $x_1 \leftarrow \arcsin a$;
        $x_2 \leftarrow \pi - \arcsin a$;
        afișează x_1, x_2;
    altfel `// cazul` $a \in [-1, 0)$
      dacă $a = -1$ **atunci** `// cazul` $a = -1$
        $x_1 \leftarrow \dfrac{3\pi}{2}$;
        afișează x_1;
      altfel `// cazul` $a \in (-1, 0)$
        $x_1 \leftarrow \pi + \arcsin(-a)$;
        $x_2 \leftarrow 2\pi - \arcsin(-a)$;
        afișează x_1, x_2;

Exemplul 2.5.7. Să se calculeze nota finală a unui student la examenul de *Programare*, cunoscând notele acestuia la fiecare din *activitățile periodice* evaluate:

- *Activitate laborator*;
- *Temă de casă*;

precum și nota de la *Examen*, dacă studentul îndeplinește *condiția de participare*. Fie

- NL = nota la *Activitatea de laborator*;
- NT = nota la *Tema de casă*;
- NE = nota la *Examen* (dacă este cazul).

Toate aceste note sunt numere întregi din mulțimea $\{0, 1, 2, \ldots, 10\}$.

Condiția de participare la examen este promovarea *activităților periodice obligatorii*:

- *Activitate laborator*;
- *Tema de casă*,

adică notele la aceste activități trebuie sa fie mai mari sau egale cu 5.

Pentru calculul *notei finale*, notată cu NF, se procedează astfel:

- Dacă studentul nu a promovat toate activitățile obligatorii, atunci în locul notei finale se menționează calificativul *ABSENT* și examenul nu este promovat;
- În caz contrar:
  - se calculează *punctajul activităților periodice*, dat de formula
  $$PA = 30\% \times NL + 20\% \times NT;$$
  - Dacă nota de la examen (NE) este mai mică decât 5, atunci nota finală se consideră ca fiind 4 și examenul nu este promovat;
  - În caz contrar:
    - se calculează *punctajul total*, dat de formula
    $$PT = PA + 50\% \times NE;$$
    - Se calculează nota finală, NF, prin *rotunjirea la cel mai apropiat număr întreg* a punctajului total, **cu excepția valorilor mai mari ca 4 și mai mici decât 5, când rotunjirea se face la 4**. Examenul este considerat promovat dacă nota finală este mai mare sau egală cu 5.

Reamintim că orice număr $x \in \mathbb{R}$ este cuprins între două numere întregi consecutive, mai precis

$$[x] \leq x < [x] + 1,$$

unde $[x] \in \mathbb{Z}$ reprezintă *partea întreagă* a lui x, definită prin

$$[x] = \max\{k \mid k \in \mathbb{Z},\ k \leq x\}.$$

Rotunjirea la cel mai apropiat număr întreg a numărului $x \in \mathbb{R}$, notată cu $\text{ROUND}(x)$, poate fi definită astfel:

$$\text{ROUND}(x) = \begin{cases} [x], & \text{dacă } [x] \leq x < [x] + 0.5, \\ [x] + 1, & \text{dacă } [x] + 0.5 \leq x < [x] + 1. \end{cases}$$

Evident, $\text{ROUND}(x) \in \mathbb{Z},\ \forall x \in \mathbb{R}$, și are loc egalitatea

$$\text{ROUND}(x) = [x + 0.5],\ \forall x \in \mathbb{R}.$$

De exemplu, avem:

$$\begin{aligned} &\text{ROUND}(7) = [7 + 0.5] = [7.5] = 7, \\ &\text{ROUND}(7.49) = [7.49 + 0.5] = [7.99] = 7, \\ &\text{ROUND}(7.5) = [7.5 + 0.5] = [8] = 8, \\ &\text{ROUND}(7.8) = [7.8 + 0.5] = [8.3] = 8. \end{aligned}$$

Obținem următorul algoritm pentru calculul notei finale.
Date de intrare: NL, NT, NE;
Date de ieșire: NF;
Variabile: $NL, NT, NE, NF \in \mathbb{Z},\ PA, PT \in \mathbb{R}$;
citește NL, NT;
dacă $NL < 5$ *sau* $NT < 5$ **atunci**
  afișează "ABSENT; NEPROMOVAT";
altfel
  $PA \leftarrow 0.3 \times NL + 0.2 \times NT$;
  citește NE;
  dacă $NE < 5$ **atunci**
    $NF \leftarrow 4$;
    afișează NF,"NEPROMOVAT";
  altfel
    $PT \leftarrow PA + 0.5 \times NE$;
    dacă $PT > 4$ *și* $PT < 5$ **atunci**
      $NF \leftarrow 4$;
    altfel
      $NF \leftarrow [PT + 0.5]$;
    afișează NF;
    dacă $NF \geq 5$ **atunci**
      afișează "PROMOVAT";
    altfel
      afișează "NEPROMOVAT";

2.6 Instrucțiunea repetitivă cu contor (cu număr finit de pași)

- *Sintaxa*:

pentru *contor* = $\overline{\textit{val. inițială},\ \textit{val. finală},\ \textit{pas}}$ **execută**
  instrucțiuni

unde:

  - *contor* este o variabilă de tip întreg;
  - *val. inițială* și *val. finală* sunt expresii de tip întreg;
  - *pas* este o constantă (sau expresie) nenulă de tip întreg.

- *Efect*:

– variabilei *contor* i se atribuie succesiv valorile *progresiei aritmetice*

$$\begin{aligned} v_1 &= \textit{val. inițială}, \\ v_2 &= v_1 + \textit{pas}, \\ v_3 &= v_2 + \textit{pas}, \\ &\dots, \\ v_n &= v_{n-1} + \textit{pas}, \end{aligned}$$

unde v_n este ultima valoare din această progresie ce satisface proprietatea

$$\begin{cases} v_n \leq \textit{val. finală}, & \text{dacă } \textit{pas} > 0 \text{ (\textbf{varianta ascendentă})}, \\ v_n \geq \textit{val. finală}, & \text{dacă } \textit{pas} < 0 \text{ (\textbf{varianta descendentă})}, \end{cases}$$

adică

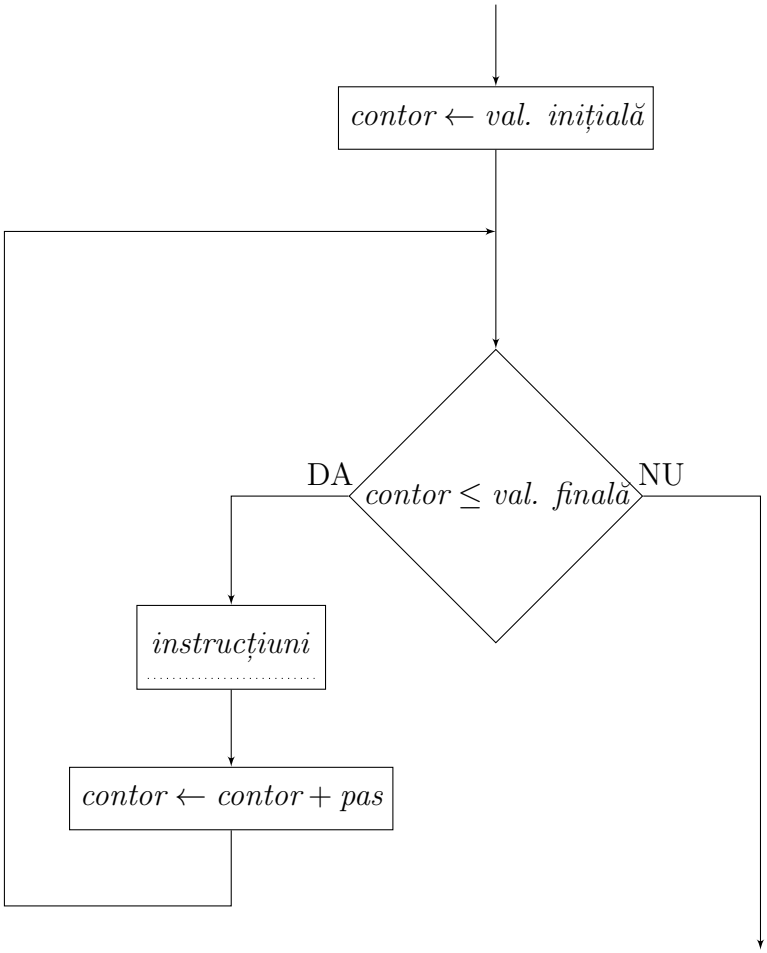
$$\begin{cases} v_n \leq \textit{val. finală} < v_n + \textit{pas}, & \text{dacă } \textit{pas} > 0, \\ v_n \geq \textit{val. finală} > v_n + \textit{pas}, & \text{dacă } \textit{pas} < 0; \end{cases}$$

– după fiecare astfel de atribuire, se execută instrucțiunile *instrucțiuni*;

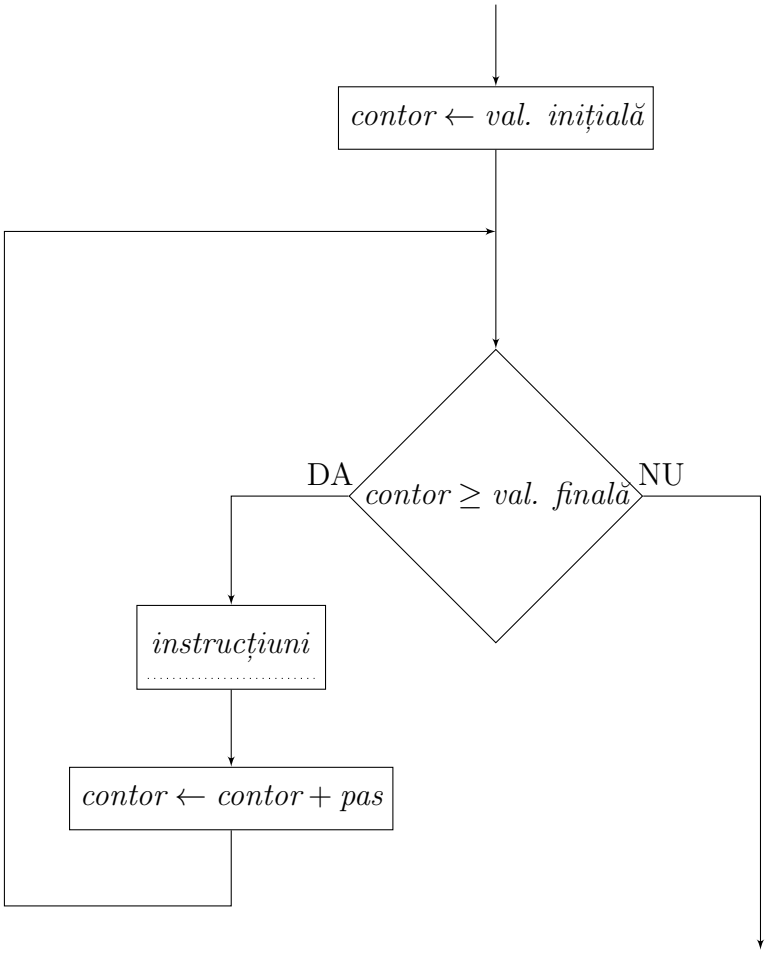
– după aceste atribuiri succesive, variabilei *contor* i se atribuie valoarea $v_{n+1} = v_n + \textit{pas}$, ceea ce conduce la terminarea (încheierea) aplicării (executării) instrucțiunii repetitive **pentru**.

- *Schema logică*:

  – pentru **varianta ascendentă**, adică pentru varianta în care avem $\textit{pas} > 0$:



– pentru **varianta descendentă**, adică pentru varianta în care avem *pas* < 0:



Observația 2.6.1. Valorile $v_1, v_2, \ldots, v_n, v_{n+1}$ $(n \in \mathbb{N})$ atribuite succesiv variabilei *contor* sunt în progresie aritmetică cu rația *pas*, deci

$$v_k = v_1 + (k-1) \cdot \textit{pas} = \textit{val. inițială} + (k-1) \cdot \textit{pas}, \ \forall\, k \in \{1, 2, \ldots, n+1\}.$$

Astfel, ultima valoare a variabilei *contor* pentru care se execută instrucțiunile *instrucțiuni* este

$$v_n = v_1 + (n-1) \cdot \textit{pas} = \textit{val. inițială} + (n-1) \cdot \textit{pas}$$

(sub presupunerea că aceste instrucțiuni se execută cel puțin o dată, adică $n \geq 1$), iar la ieșirea din instrucțiunea repetitivă **pentru** valoarea variabilei *contor* este

$$v_{n+1} = v_n + \textit{pas} = v_1 + n \cdot \textit{pas} = \textit{val. inițială} + n \cdot \textit{pas}.$$

Observația 2.6.2. Considerăm cazul **variantei ascendente**, *pas* > 0.

- Valorile $v_1, v_2, \ldots, v_n$ ($n \in \mathbb{N}$) atribuite succesiv variabilei *contor* și pentru care se execută instrucțiunile *instrucțiuni* sunt în ordine crescătoare. Mai mult, avem

$$\begin{aligned} \textit{val. inițială} = v_1 < v_1 + \textit{pas} = v_2 < v_2 + \textit{pas} = v_3 < \ldots \\ < v_{n-1} + \textit{pas} = v_n \leq \textit{val. finală} < v_n + \textit{pas} = v_{n+1}. \end{aligned}$$

- Dacă *val. inițială* $>$ *val. finală*, atunci instrucțiunile *instrucțiuni* nu se execută nici o dată (nici măcar o singură dată), adică numărul de execuții ale acestora este $n = 0$.

  În acest caz, la ieșirea din instrucțiunea repetitivă **pentru** valoarea variabilei *contor* este

$$v_1 = \textit{val. inițială}.$$

- Dacă *val. inițială* $\leq$ *val. finală*, atunci ultima valoare a variabilei *contor* pentru care se execută instrucțiunile *instrucțiuni*, adică

$$v_n = \textit{val. inițială} + (n-1) \cdot \textit{pas},$$

  verifică inegalitățile

$$v_n \leq \textit{val. finală} < v_n + \textit{pas},$$

  deci

$$n - 1 \leq \frac{\textit{val. finală} - \textit{val. inițială}}{\textit{pas}} < n.$$

  Rezultă că, în acest caz, numărul de execuții ale instrucțiunilor *instrucțiuni* este

$$n = \left[\frac{\textit{val. finală} - \textit{val. inițială}}{\textit{pas}} \right] + 1 \tag{2.6.1}$$

  (deci aceste instrucțiuni se execută cel puțin o dată), ultima valoare a variabilei *contor* pentru care se execută aceste instrucțiuni este

$$v_n = \textit{val. inițială} + \left[\frac{\textit{val. finală} - \textit{val. inițială}}{\textit{pas}} \right] \cdot \textit{pas}, \tag{2.6.2}$$

  iar la ieșirea din instrucțiunea repetitivă **pentru** valoarea variabilei *contor* este

$$v_{n+1} = \textit{val. inițială} + \left(\left[\frac{\textit{val. finală} - \textit{val. inițială}}{\textit{pas}} \right] + 1 \right) \cdot \textit{pas}. \tag{2.6.3}$$

Observația 2.6.3. Considerăm acum cazul **variantei descendente**, *pas* < 0.

- Valorile $v_1, v_2, \ldots, v_n$ ($n \in \mathbb{N}$) atribuite succesiv variabilei *contor* și pentru care se execută instrucțiunile *instrucțiuni* sunt acum în ordine descrescătoare. Mai mult, avem

$$\begin{aligned} \textit{val. inițială} = v_1 > v_1 + \textit{pas} = v_2 > v_2 + \textit{pas} = v_3 > \ldots \\ > v_{n-1} + \textit{pas} = v_n \geq \textit{val. finală} > v_n + \textit{pas} = v_{n+1}. \end{aligned}$$

- Dacă *val. inițială* $<$ *val. finală*, atunci instrucțiunile *instrucțiuni* nu se execută nici o dată, adică numărul de execuții ale acestora este $n = 0$.

  În acest caz, la ieșirea din instrucțiunea repetitivă **pentru** valoarea variabilei *contor* este, din nou,

$$v_1 = \textit{val. inițială}.$$

- Dacă *val. inițială* $\geq$ *val. finală*, atunci ultima valoare a variabilei *contor* pentru care se execută instrucțiunile *instrucțiuni*, adică

$$v_n = \textit{val. inițială} + (n-1) \cdot \textit{pas},$$

  verifică acum inegalitățile

$$v_n \geq \textit{val. finală} > v_n + \textit{pas},$$

  deci, din nou,

$$n - 1 \leq \frac{\textit{val. finală} - \textit{val. inițială}}{\textit{pas}} < n.$$

  Continuând ca în cazul variantei ascendente, rezultă din nou că numărul de execuții ale instrucțiunilor *instrucțiuni* este dat de relația (2.6.1) (deci aceste instrucțiuni se execută cel puțin o dată), ultima valoare a variabilei *contor* pentru care se execută aceste instrucțiuni este dată de relația (2.6.2) și la ieșirea din instrucțiunea repetitivă **pentru** valoarea variabilei *contor* este dată de relația (2.6.3).

Observația 2.6.4. În cazul particular, cel mai des utilizat, în care valoarea constantei *pas* este

$$\textit{pas} = 1,$$

această constantă poate fi omisă din sintaxa instrucțiunii repetitive **pentru**, adică instrucțiunea are forma

pentru *contor* $= \overline{\textit{val. inițială}, \textit{val. finală}}$ **execută** // *pas* $= 1$
  instrucțiuni

În acest caz avem:

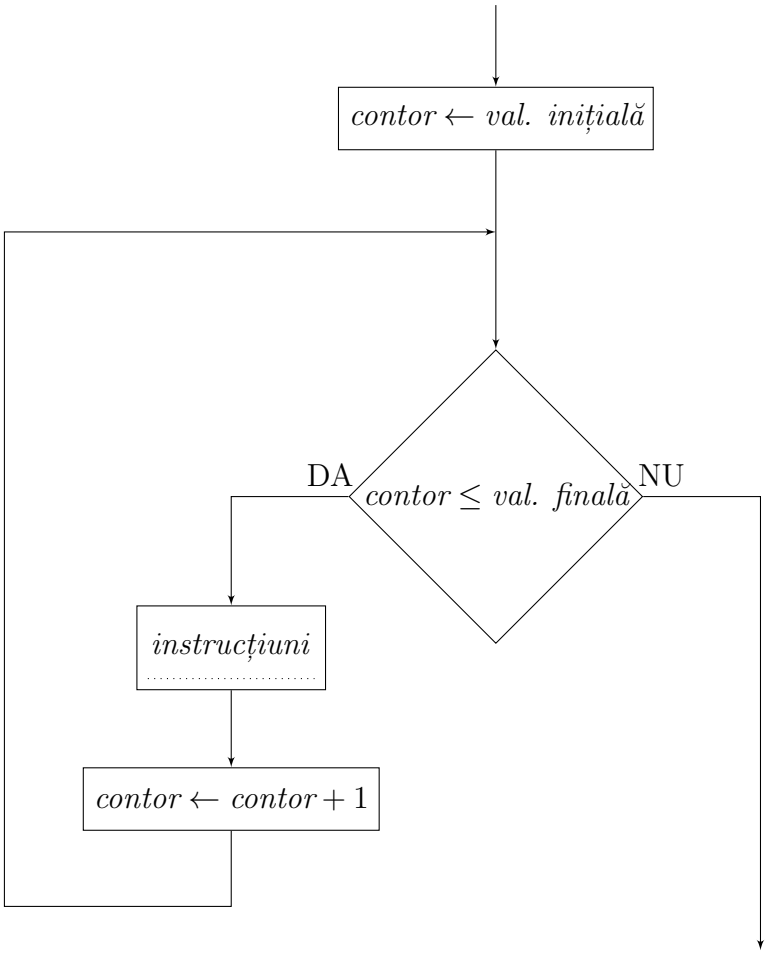
- dacă *val. inițială* > *val. finală*, atunci instrucțiunile *instrucțiuni* nu se execută nici o dată și la ieșirea din instrucțiunea repetitivă **pentru** variabila *contor* are valoarea *val. inițială*;

- dacă *val. inițială* ≤ *val. finală*, atunci instrucțiunile *instrucțiuni* se execută pentru fiecare din valorile succesive

  $$\textit{val. inițială}, \textit{val. inițială} + 1, \textit{val. inițială} + 2, \ldots, \textit{val. finală}$$

  ale variabilei *contor* (deci de *val. finală* − *val. inițială* + 1 ori), iar la ieșirea din instrucțiunea repetitivă **pentru** variabila *contor* are valoarea *val. finală* + 1.

- *Schema logică*:



Observația 2.6.5. Un alt cazul particular des utilizat este cel în care valoarea constantei *pas* este

$$pas = -1,$$

adică instrucțiunea repetitivă **pentru** are forma

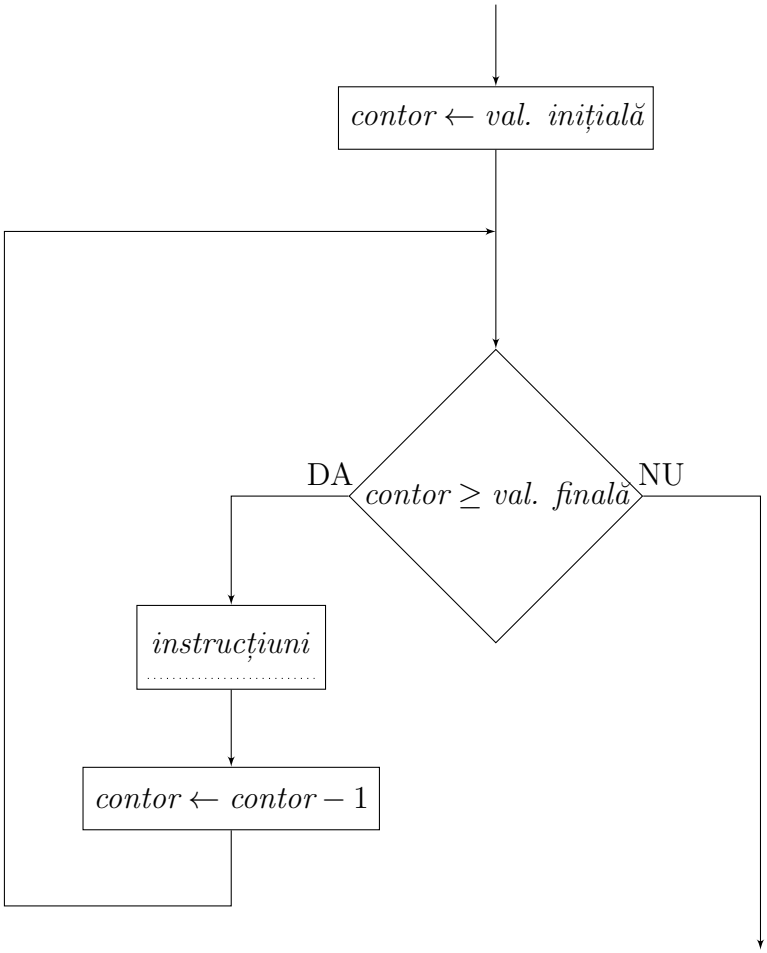
pentru *contor* $=$ $\overline{\textit{val. inițială}, \textit{val. finală}, -1}$ **execută** // $pas = -1$
  instrucțiuni

În acest caz avem:

- dacă *val. inițială* $<$ *val. finală*, atunci instrucțiunile *instrucțiuni* nu se execută nici o dată și la ieșirea din instrucțiunea repetitivă **pentru** variabila *contor* are valoarea *val. inițială*;
- dacă *val. inițială* $\geq$ *val. finală*, atunci instrucțiunile *instrucțiuni* se execută pentru fiecare din valorile succesive

  $$\textit{val. inițială}, \textit{val. inițială} - 1, \textit{val. inițială} - 2, \ldots, \textit{val. finală}$$

  ale variabilei *contor* (deci de *val. inițială* $-$ *val. finală* $+ 1$ ori), iar la ieșirea din instrucțiunea repetitivă **pentru** variabila *contor* are valoarea *val. finală* $- 1$.
- *Schema logică*:



Exemplul 2.6.1. Pentru următorul algoritm, efectul fiecărei instrucțiuni este

scris la comentarii.

Variabile: $i \in \mathbb{Z}$;
pentru $i = \overline{2,\ 10}$ **execută**
  instr 1 `// se execută, succesiv, pentru` $i = 2, 3, 4, \ldots, 10$
  `//` $i = 11$
pentru $i = \overline{2,\ 10,\ 2}$ **execută**
  instr 2 `// se execută, succesiv, pentru` $i = 2, 4, 6, 8, 10$
  `//` $i = 12$
pentru $i = \overline{2,\ 10,\ 3}$ **execută**
  instr 3 `// se execută, succesiv, pentru` $i = 2, 5, 8$
  `//` $i = 11$
pentru $i = \overline{2,\ 10,\ -1}$ **execută**
  instr 4 `// nu se execută nici o dată`
  `//` $i = 2$
pentru $i = \overline{10,\ 2,\ -1}$ **execută**
  instr 5 `// se execută, succesiv, pentru` $i = 10, 9, 8, \ldots, 2$
  `//` $i = 1$
pentru $i = \overline{10,\ 2,\ -2}$ **execută**
  instr 6 `// se execută, succesiv, pentru` $i = 10, 8, 6, 4, 2$
  `//` $i = 0$
pentru $i = \overline{10,\ 2,\ -3}$ **execută**
  instr 7 `// se execută, succesiv, pentru` $i = 10, 7, 4$
  `//` $i = 1$
pentru $i = \overline{10,\ 2}$ **execută**
  instr 8 `// nu se execută nici o dată`
  `//` $i = 10$
pentru $i = \overline{10,\ 2,\ 2}$ **execută**
  instr 9 `// nu se execută nici o dată`
  `//` $i = 10$

Observația 2.6.6. Pentru calculul unei sume de forma

$$S = \sum_{k=p}^{n} f(k) = f(p) + f(p+1) + f(p+2) + \cdots + f(n),$$

unde $p, n \in \mathbb{Z},\ p \leq n$, iar $f : \{p, p+1, p+2, \ldots, n\} \to \mathbb{R}$ este o funcție, se

poate utiliza un algoritm având următoarea formă:

$S \leftarrow 0$; `// inițializăm` S `cu 0 (înainte de a însuma // termenii doriți, valoarea unei sume este 0)`
pentru $k = \overline{p,\, n}$ **execută**
  `// pentru fiecare` $k = p, p+1, \ldots, n$,
  $S \leftarrow S + f(k)$; `// la suma` S `se adună termenul` $f(k)$

Pentru calculul unui produs de forma

$$P = \prod_{k=p}^{n} f(k) = f(p) \cdot f(p+1) \cdot f(p+2) \cdot \ldots \cdot f(n),$$

unde, din nou, $p, n \in \mathbb{Z},\ p \leq n$ și $f : \{p, p+1, p+2, \ldots, n\} \to \mathbb{R}$ este o funcție, se poate utiliza un algoritm având următoarea formă:

$P \leftarrow 1$; `// inițializăm` P `cu 1 (înainte de a înmulți // factorii doriți, valoarea unui produs este 1)`
pentru $k = \overline{p,\, n}$ **execută**
  `// pentru fiecare` $k = p, p+1, \ldots, n$,
  $P \leftarrow P \times f(k)$; `// la produsul` P `se înmulțește // factorul` $f(k)$

Exemplul 2.6.2. Să se calculeze suma

$$S = \frac{2 \sin 1}{3} + \frac{3 \sin 2}{4} + \cdots + \frac{(n+1) \sin n}{n+2},$$

unde $n \in \mathbb{N}^*$ este un număr dat.

Suma poate fi scrisă, prescurtat, sub forma

$$S = \sum_{k=1}^{n} \frac{(k+1) \sin k}{k+2}.$$

Următorul algoritm rezolvă problema dată.

Date de intrare: n;
Date de ieșire: S;
Variabile: $n, k \in \mathbb{Z},\ S \in \mathbb{R}$;
citește n;
$S \leftarrow 0$;
pentru $k = \overline{1,\, n}$ **execută**
  $S \leftarrow S + \dfrac{(k+1) \times \sin k}{k+2}$;
afișează S;

Exemplul 2.6.3. Să se calculeze suma

$$S = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \cdots + (-1)^{n-1} \cdot \frac{1}{n},$$

unde $n \in \mathbb{N}^*$ este un număr dat.

Suma poate fi scrisă, prescurtat, sub forma

$$S = \sum_{k=1}^{n} (-1)^{k-1} \cdot \frac{1}{k}.$$

Pentru fiecare $k = \overline{1, n}$, termenul $\frac{1}{k}$ se adună la sumă când k este impar, respectiv se scade din sumă când k este par.

Următorul algoritm rezolvă problema dată.

Date de intrare: n;
Date de ieșire: S;
Variabile: $n, k \in \mathbb{Z},\ S \in \mathbb{R}$;
citește n;
$S \leftarrow 0$;
pentru $k = \overline{1, n}$ **execută** // $k = 1, 2, 3, \ldots, n$
  dacă k *MOD* $2 = 0$ **atunci** // k este par
    $S \leftarrow S - \frac{1}{k}$; // scădem $\frac{1}{k}$ din S
  altfel // k este impar
    $S \leftarrow S + \frac{1}{k}$; // adunăm $\frac{1}{k}$ la S
afișează S;

Exemplul 2.6.4. Să se calculeze suma rădăcinilor pătrate (radicalilor) ale tuturor numerelor naturale impare mai mici sau egale cu un număr real x dat.

De exemplu, pemtru $x = 10.23$ suma cerută este

$$S = \sqrt{1} + \sqrt{3} + \sqrt{5} + \sqrt{7} + \sqrt{9}.$$

Evident, dacă $x < 1$ atunci nu există numere naturale impare mai mici sau egale cu x. În acest caz se poate considera că suma cerută este suma elementelor mulțimii vide, această sumă fiind, prin convenție, egală cu 0.

Dacă $x \geq 1$, atunci suma cerută este

$$S = \sqrt{1} + \sqrt{3} + \sqrt{5} + \cdots + \sqrt{m},$$

unde m este cel mai mare număr impar mai mic sau egal cu $[x]$.

Următorul algoritm rezolvă problema dată.
Date de intrare: x;
Date de ieșire: S;
Variabile: $k \in \mathbb{Z},\ x, S \in \mathbb{R}$;
citește x;
$S \leftarrow 0$;
dacă $x < 1$ **atunci**
  afișează "$S = 0$; nu există numere cu proprietatea cerută";
altfel
  pentru $k = \overline{1,\ [x],\ 2}$ **execută** `// parcurgem numerele de la 1 // la [x], din 2 în 2`
    $S \leftarrow S + \sqrt{k}$; `// adunarea se execută, succesiv, // pentru` $k = 1, 3, 5, \ldots, m$
  afișează S;

Exemplul 2.6.5. Să se calculeze puterea x^n, unde $x \in \mathbb{R}$ și $n \in \mathbb{N}^*$ sunt numere date.

Puterea x^n poate fi scrisă sub formă de produs

$$x^n = \underbrace{x \cdot x \cdot \ldots \cdot x}_{\text{de } n \text{ ori } x} = \prod_{k=1}^{n} x.$$

Următorul algoritm rezolvă problema dată.
Date de intrare: x, n;
Date de ieșire: P `//` $P = x^n$;
Variabile: $n, k \in \mathbb{Z},\ x, P \in \mathbb{R}$;
citește x, n;
$P \leftarrow 1$;
pentru $k = \overline{1,\ n}$ **execută**
  $P \leftarrow P \times x$;
afișează P;

Exemplul 2.6.6. Fie $n \in \mathbb{N}$. Să se calculeze $n!$ (n *factorial*).

Reamintim că

$$0! = 1 \text{ și } n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n \text{ pentru } n \in \mathbb{N}^*.$$

Evident, $n!$ poate fi scris sub formă de produs

$$n! = \prod_{k=1}^{n} k,$$

egalitate valabilă și pentru $n = 0$ deoarece produsul elementelor mulțimii vide este egal, prin convenție, cu 1.

Următorul algoritm rezolvă problema dată.

Date de intrare: n;
Date de ieșire: P // $P = n!$;
Variabile: $n, P, k \in \mathbb{Z}$;
citește n;
$P \leftarrow 1$;
pentru $k = \overline{1,\ n}$ **execută** // sau $k = \overline{2,\ n}$
  $P \leftarrow P \times k$;
afișează P;

Exemplul 2.6.7. Fie $n, k \in \mathbb{N}$, $k \leq n$. Să se calculeze A_n^k (*aranjamente de n luate câte k*).

Reamintim că

$$A_n^k = \frac{n!}{(n-k)!}.$$

Avem

$$\begin{aligned} A_n^k &= \frac{1 \cdot 2 \cdot \ldots \cdot (n-k) \cdot (n-k+1) \cdot \ldots \cdot n}{1 \cdot 2 \cdot \ldots \cdot (n-k)} \\ &= \underbrace{n(n-1)(n-2)\ldots(n-k+1)}_{k \text{ factori}}. \end{aligned}$$

Următorul algoritm rezolvă problema dată. Deoarece factorii produsului $n(n-1)(n-2)\ldots(n-k+1)$ sunt consecutiv descrescători, utilizăm *varianta descendentă* a instrucțiunii repetitive **pentru**, cu pasul *pas* $= -1$.

Date de intrare: n, k;
Date de ieșire: P // $P = A_n^k$;
Variabile: $n, k, P, i \in \mathbb{Z}$;
citește n, k;
$P \leftarrow 1$;
pentru $i = \overline{n,\ n-k+1,\ -1}$ **execută** // $i = n, n-1, \ldots, n-k+1$
  $P \leftarrow P \times i$;
afișează P;

Exemplul 2.6.8. Fie $n, k \in \mathbb{N}$, $k \leq n$. Să se calculeze C_n^k (*combinări de n luate câte k*).

Reamintim că

$$C_n^k = \frac{n!}{k! \cdot (n-k)!}.$$

Avem

$$\begin{aligned} C_n^k &= \frac{A_n^k}{k!} \\ &= \frac{n(n-1)(n-2)\dots(n-k+1)}{1\cdot 2\cdot 3\cdot \dots \cdot k} \\ &= \frac{n}{1}\cdot\frac{n-1}{2}\cdot\frac{n-2}{3}\cdot\dots\cdot\frac{n-k+1}{k} \\ &= \prod_{i=1}^{k}\frac{n-i+1}{i}. \end{aligned}$$

Următorul algoritm rezolvă problema dată.
Date de intrare: n, k;
Date de ieșire: P // $P = C_n^k$;
Variabile: $n, k, P, i \in \mathbb{Z}$;
citește n, k;
$P \leftarrow 1$;
pentru $i = \overline{1, k}$ **execută**
  $P \leftarrow \dfrac{P \times (n-i+1)}{i}$;
afișează P;

Exemplul 2.6.9. Să se calculeze suma

$$S = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots + \frac{1}{n!},$$

unde $n \in \mathbb{N}^*$ este un număr dat.

Suma poate fi scrisă, prescurtat, sub forma

$$S = \sum_{k=0}^{n}\frac{1}{k!}.$$

Următorul algoritm rezolvă problema dată.
Date de intrare: n;
Date de ieșire: S;
Variabile: $n, k, P, i \in \mathbb{Z},\ S \in \mathbb{R}$;
citește n;
$S \leftarrow 0$;
pentru $k = \overline{0,\ n}$ **execută** `// ` $k = 0, 1, 2, \ldots, n$
`// calculăm` $P = k!$
  $P \leftarrow 1$;
  pentru $i = \overline{2,\ k}$ **execută**
    $P \leftarrow P \times i$;
  $S \leftarrow S + \dfrac{1}{P}$; `// adunăm` $\dfrac{1}{P} = \dfrac{1}{k!}$ `la` S
afișează S;

Exemplul 2.6.10. Să se calculeze suma divizorilor naturali ai unui număr natural nenul n dat.

De exemplu, pentru $n = 36$ suma cerută este

$$S = 1 + 2 + 3 + 4 + 6 + 9 + 12 + 18 + 36 = 91.$$

O metodă de rezolvare a problemei constă în:

- se parcurg toate numerele k de la 1 la n, notat prescurtat prin $k = \overline{1,\ n}$;
- pentru fiecare k se verifică dacă el este divizor al lui n;
- dacă da, atunci k se adună la suma S.

Descrierea în pseudocod a acestui algoritm are următoarea formă.
Date de intrare: n;
Date de ieșire: S;
Variabile: $n, S, k \in \mathbb{Z}$;
citește n;
$S \leftarrow 0$;
pentru $k = \overline{1,\ n}$ **execută**
  dacă n *MOD* $k = 0$ **atunci** `// ` k `divide` n
    $S \leftarrow S + k$;
afișează S;

Putem însă să calculăm suma cerută și fără a parcurge toate numerele de la 1 la n, pornind de la observația că orice divizor al lui n mai mare decât $\sqrt{n}$ are forma $\dfrac{n}{k}$, unde k este un divizor al lui n mai mic decât $\sqrt{n}$. Obținem astfel o metodă mai eficientă de rezolvare, ce constă în:

- se parcurg toate numerele k de la 1 la $[\sqrt{n}]$;
- pentru fiecare k se verifică dacă el este divizor al lui n;
- dacă da, atunci se adună la suma S atât divizorul k cât și divizorul $\frac{n}{k}$;
- dacă n este pătrat perfect, adică $[\sqrt{n}] = \sqrt{n}$, atunci scădem divizorul $\sqrt{n}$ din suma S (deoarece a fost adunat de două ori, ca $\sqrt{n}$ și ca $\frac{n}{\sqrt{n}}$).

Descrierea în pseudocod a acestui algoritm are următoarea formă.

Date de intrare: n;
Date de ieșire: S;
Variabile: $n, S, k \in \mathbb{Z},\ r \in \mathbb{R}$;
citește n;
$r \leftarrow \sqrt{n}$;
$S \leftarrow 0$;
pentru $k = \overline{1,\ [r]}$ **execută**
  dacă $n\ MOD\ k = 0$ **atunci**
    $S \leftarrow S + k + \frac{n}{k}$;
dacă $[r] = r$ **atunci** `// n este pătrat perfect`
  $S \leftarrow S - [r]$;
afișează S;

Exemplul 2.6.11. Să se calculeze media (aritmetică a) divizorilor naturali ai unui număr natural nenul n dat.

De exemplu, pentru $n = 36$ media cerută este

$$M = \frac{1+2+3+4+6+9+12+18+36}{9} = \frac{91}{9} = 10.(1)\ .$$

Pentru rezolvare, adaptăm algoritmii din exemplul anterior, calculând atât suma S a divizorilor cât și numărul lor, notat cu nr, iar media cerută va fi

$$M = \frac{S}{nr}.$$

O primă metodă de rezolvare a problemei constă în:

- se parcurg toate numerele k de la 1 la n, notat prescurtat prin $k = \overline{1,\ n}$;
- pentru fiecare k se verifică dacă el este divizor al lui n;
- dacă da, atunci k se adună la suma S iar numărul de divizori nr se mărește cu 1 (se *incrementează*);

- după încheierea parcurgerii numerelor k, calculăm media $M = \frac{S}{nr}$.

Descrierea în pseudocod a acestui algoritm are următoarea formă.

Date de intrare: n;
Date de ieșire: M;
Variabile: $n, S, nr, k \in \mathbb{Z},\ M \in \mathbb{R}$;
citește n;
$S \leftarrow 0$;
$nr \leftarrow 0$;
pentru $k = \overline{1,\ n}$ **execută**
  dacă n *MOD* $k = 0$ **atunci** `// k divide n`
    $S \leftarrow S + k$;
    $nr \leftarrow nr + 1$;
$M \leftarrow \frac{S}{nr}$;
afișează M;

O metodă mai eficientă de rezolvare, fără a parcurge toate numerele de la 1 la n, constă în:

- se parcurg toate numerele k de la 1 la $[\sqrt{n}]$;
- pentru fiecare k se verifică dacă el este divizor al lui n;
- dacă da, atunci se adună la suma S atât divizorul k cât și divizorul $\frac{n}{k}$, iar numărul de divizori nr se mărește cu 2;
- dacă n este pătrat perfect, adică $[\sqrt{n}] = \sqrt{n}$, atunci scădem divizorul $\sqrt{n}$ din suma S iar numărul de divizori nr se micșorează cu 1 (se *decrementează*) (deoarece divizorul $\sqrt{n}$ a fost adunat și numărat de două ori, ca $\sqrt{n}$ și ca $\frac{n}{\sqrt{n}}$);
- după încheierea parcurgerii numerelor k, calculăm media $M = \frac{S}{nr}$.

Descrierea în pseudocod a acestui algoritm are următoarea formă.

Date de intrare: n;
Date de ieșire: M;
Variabile: $n, S, k \in \mathbb{Z},\ r \in \mathbb{R}$;
citește n;
$r \leftarrow \sqrt{n}$;
$S \leftarrow 0$;
$nr \leftarrow 0$;
pentru $k = \overline{1,\ [r]}$ **execută**
  dacă $n\ MOD\ k = 0$ **atunci**
    $S \leftarrow S + k + \dfrac{n}{k}$;
    $nr \leftarrow nr + 2$;
dacă $[r] = r$ **atunci** `// n este pătrat perfect`
  $S \leftarrow S - [r]$;
  $nr \leftarrow nr - 1$;
$M \leftarrow \dfrac{S}{nr}$;
afișează M;

Exemplul 2.6.12. Să se verifice dacă un număr natural n dat este sau nu un număr prim.

Reamintim că un număr prim este un număr natural care are exact doi divizori: pe 1 și pe el însuși.

Pentru rezolvare vom defini și utiliza (*apela*) o funcție *PRIM* având semnificația

$$PRIM(x) = \begin{cases} 1, & \text{dacă } x \text{ este număr prim,} \\ 0, & \text{în caz contrar,} \end{cases} \quad \forall\, x \in \mathbb{N}.$$

Evident, numerele 0 și 1 nu sunt prime, iar numerele 2 și 3 sunt prime.

Un număr $x > 3$ nu este prim dacă și numai dacă se divide cu 2 (este par) sau cu un număr impar din mulțimea $\{3, \ldots, [\sqrt{x}\,]\}$.

Obținem următorul algoritm pentru rezolvarea problemei date.

Date de intrare: n;
Variabile: $n \in \mathbb{Z}$;
Funcția $PRIM(x) \in \{0,1\}$ **:** // $x \in \mathbb{Z}$
  Variabile: $k \in \mathbb{Z}$;
  dacă $x < 2$ **atunci**
    returnează 0; // 0 și 1 nu sunt prime
  // urmează cazul $x \geq 2$
  dacă $x < 4$ **atunci**
    returnează 1; // 2 și 3 sunt prime
  // urmează cazul $x \geq 4$
  dacă x *MOD* $2 = 0$ **atunci** // x este par,
    returnează 0; // deci nu este prim
  // urmează cazul x impar, $x \geq 5$
  pentru $k = \overline{3,\ [\sqrt{x}],\ 2}$ **execută** // k parcurge numerele
    // impare de la 3 la $[\sqrt{x}]$ (din 2 în 2)
    dacă x *MOD* $k = 0$ **atunci** // x se divide cu k,
      returnează 0; // deci nu este prim
  // în toate celelalte cazuri posibile
  returnează 1; // x este prim
// definirea funcției $PRIM$ este încheiată;
// continuăm algoritmul, în care vom apela funcția $PRIM$
citește n;
dacă $PRIM(n) = 1$ **atunci**
  afișează n," este prim";
altfel
  afișează n,"nu este prim";

Exemplul 2.6.13. Se consideră un vector $v = (v_1, v_2, \ldots, v_n)$ cu elemente întregi, $n \geq 1$. Să se calculeze suma elementelor impare.

De exemplu, pentru vectorul $v = (-51, 10, 6, 11, -42, 0, 11, 22, 7)$ suma cerută este

$$S = -51 + 11 + 11 + 7 = -22.$$

Următorul algoritm rezolvă problema dată.

Date de intrare: $n, v = (v_1, v_2, \ldots, v_n)$;
Date de ieșire: S;
Variabile: $n, S, nr, i \in \mathbb{Z},\ v = (v_1, v_2, \ldots, v_n) \in \mathbb{Z}^n$;
citește n; `// citim numărul de elemente`
pentru $i = \overline{1, n}$ **execută** `// citim elementele vectorului`
  citește v_i;
$S \leftarrow 0$;
$nr \leftarrow 0$;
pentru $i = \overline{1, n}$ **execută** `// parcurgem elementele vectorului`
  dacă v_i MOD $2 = 1$ **atunci** `// v_i este impar,`
    $S \leftarrow S + v_i$; `// deci îl adunăm la S`
    $nr \leftarrow nr + 1$;
dacă $nr = 0$ **atunci**
  afișează "Vectorul nu conține elemente impare.";
altfel
  afișează S;

Exemplul 2.6.14. Se consideră un vector $a = (a_1, a_2, \ldots, a_n)$ cu elemente reale, $n \geq 1$. Să se calculeze mediile aritmetică și geometrică a elementelor pozitive.

Reamintim că *media aritmetică* a numerelor reale $x_1, x_2, \ldots, x_k$ este

$$m_a = \frac{x_1 + x_2 + \cdots + x_k}{k},$$

iar *media geometrică* a numerelor reale $x_1, x_2, \ldots, x_k \geq 0$ este

$$m_g = \sqrt[k]{x_1 \cdot x_2 \cdot \ldots \cdot x_k}.$$

De exemplu, pentru vectorul $a = (-51, 10, -76, 5, 20, -11, -42, 10, -2)$ mediile cerute sunt

$$m_a = \frac{10 + 5 + 20 + 10}{4} = \frac{45}{4} = 11.25,$$
$$m_g = \sqrt[4]{10 \cdot 5 \cdot 20 \cdot 10} = \sqrt[4]{10000} = 10.$$

Pentru calculul radicalului de ordin k vom utiliza formula

$$\sqrt[k]{x} = x^{\frac{1}{k}} = e^{\ln(x^{\frac{1}{k}})} = e^{\frac{1}{k} \cdot \ln x} = e^{\frac{\ln x}{k}},\ \forall\, x > 0.$$

Următorul algoritm rezolvă problema dată.

Date de intrare: $n, a = (a_1, a_2, \ldots, a_n)$;
Date de ieșire: MA, MG; `// media aritmetică, respectiv // media geometrică`
Variabile: $n, nr, i \in \mathbb{Z},\ MA, MG, S, P \in \mathbb{R},\ a = (a_1, a_2, \ldots, a_n) \in \mathbb{R}^n$;
citește n; `// citim numărul de elemente`
pentru $i = \overline{1, n}$ **execută** `// citim elementele vectorului`
  citește a_i;
$nr \leftarrow 0$; `// nr = numărul de elemente pozitive`
$S \leftarrow 0$; `// S = suma elementelor pozitive`
$P \leftarrow 1$; `// P = produsul elementelor pozitive`
pentru $i = \overline{1, n}$ **execută** `// parcurgem elementele vectorului`
  dacă $a_i > 0$ **atunci** `// a_i este pozitiv, deci`
    $S \leftarrow S + a_i$; `// îl adunăm la S,`
    $P \leftarrow P \times a_i$; `// îl înmulțim la P`
    $nr \leftarrow nr + 1$; `// și incrementăm nr`
dacă $nr = 0$ **atunci**
  afișează "Vectorul nu conține elemente pozitive.";
altfel
  $MA \leftarrow \dfrac{S}{nr}$; `// calculăm media aritmetică` $MA = \dfrac{S}{nr}$
  $MG \leftarrow e^{\frac{\ln P}{nr}}$; `// calculăm media geometrică` $MG = \sqrt[nr]{P} = e^{\frac{\ln P}{nr}}$
  afișează MA, MG; `// afișăm mediile`

2.7 Instrucțiunea repetitivă cu test inițial

- *Sintaxa*:

cât timp *condiție* **repetă**
  instrucțiuni

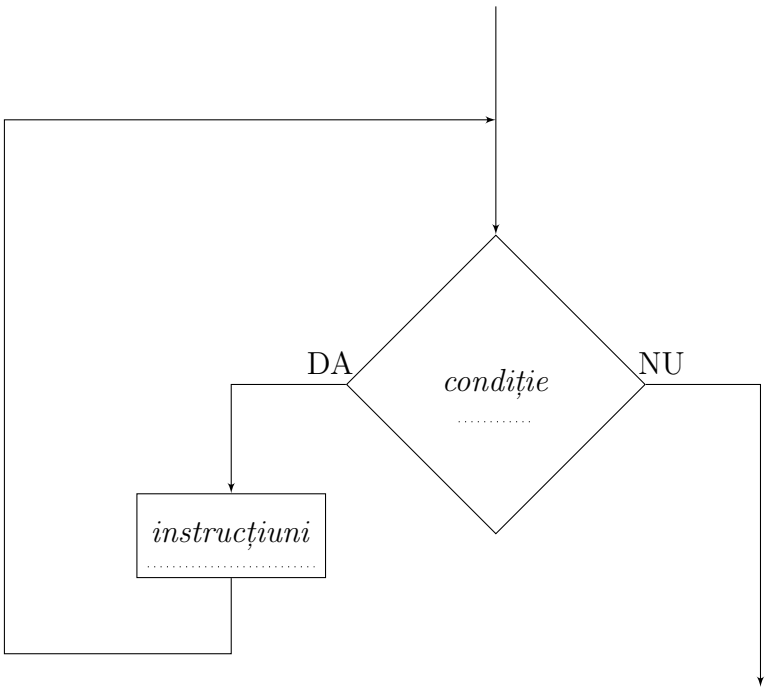
- *Efect*:

Pasul 1. Se evaluează condiția (pentru valorile curente ale variabilelor componente);

Pasul 2. Dacă ea este îndeplinită (este adevărată), atunci se execută instrucțiunile *instrucțiuni* și se revine la Pasul 1 (adică se evaluează din nou condiția, dacă ea este adevărată se execută instrucțiunile *instrucțiuni* și se evaluează din nou condiția, ș.a.m.d., cât timp condiția este adevărată, adică până când ea devine falsă);

Pasul 3. În caz contrar, adică dacă ea nu este îndeplinită (este falsă), atunci se sare peste instrucțiunile *instrucțiuni* (adică acestea nu se execută) și se încheie aplicarea instrucțiunii repetitive cu test inițial.

- *Schema logică*:



Observația 2.7.1. Condiția trebuie să devină falsă la un moment dat, deoarece în caz contrar executarea instrucțiunii repetitive cu test inițial nu s-ar încheia niciodată (spunem că algoritmul intră într-un ciclu infinit, sau că *algoritmul ciclează la infinit*), deci nu s-ar respecta caracteristica de finitudine a algoritmului.

Observația 2.7.2. Instrucțiunile *instrucțiuni* pot să nu fie executate nici o dată, și anume atunci când condiția este falsă de la prima verificare.

Exemplul 2.7.1. Să se calculeze punctajul total obținut de un elev la un test compus din patru subiecte a câte 7 puncte fiecare, cunoscând punctaje $p1, p2, p3$ și $p4$ acumulate la cele patru subiecte. Înainte de calculul punctajului total, să se *valideze datele introduse*, adică să se verifice că fiecare dintre punctajele $p1, p2, p3$ și $p4$ este mai mare sau egal cu 0 și mai mic sau egal cu 7.

Următorul algoritm rezolvă problema dată.
Date de intrare: $p1, p2, p3, p4$;
Date de ieșire: PT; `// punctajul total`
Variabile: $p1, p2, p3, p4, PT \in \mathbb{R}$;
citește $p1$; `// citim punctajul de la subiectul 1`
`// verificăm dacă acest punctaj este valid`
cât timp $p1 < 0$ *sau* $p1 > 7$ **repetă** `// punctajul nu este valid`
  afișează "Valoare incorectă! Introduceți valoarea corectă.";
  citește $p1$; `// citim din nou punctajul de la subiectul 1`
`// punctajul de la subiectul 1 este valid`
`// procedăm analog pentru punctajele` $p2, p3$ `și` $p4$
citește $p2$;
cât timp $p2 < 0$ *sau* $p2 > 7$ **repetă**
  afișează "Valoare incorectă! Introduceți valoarea corectă.";
  citește $p2$;
citește $p3$;
cât timp $p3 < 0$ *sau* $p3 > 7$ **repetă**
  afișează "Valoare incorectă! Introduceți valoarea corectă.";
  citește $p3$;
citește $p4$;
cât timp $p4 < 0$ *sau* $p4 > 7$ **repetă**
  afișează "Valoare incorectă! Introduceți valoarea corectă.";
  citește $p4$;
$PT \leftarrow p1 + p2 + p3 + p4$; `// calculăm punctajul total`
afișează PT; `// afișăm punctajul total`

Exemplul 2.7.2. Să se calculeze cel mai mic număr prim p mai mare sau egal cu un număr real x dat.

Existența unui astfel de număr p este asigurată de faptul că mulțimea numerelor prime este infinită.

De exemplu, pentru $x = 320.25$ avem $p = 331$ (numerele 321, 322, 323, ..., 330 nu sunt prime).

Evident, pentru $x \leq 2$ avem $p = 2$.

Pentru $x > 2$, p este primul număr prim din șirul numerelor impare mai mari sau egale cu x, adică din șirul $m, m+2, m+4, m+6, \ldots$, unde m este cel mai mic număr impar mai mare sau egal cu x. Notând $m = 2i+1,\ i \in \mathbb{Z}$, inegalitatea $m \geq x$ devine $i \geq \dfrac{x-1}{2}$, deci $i = \left\lceil \dfrac{x-1}{2} \right\rceil$ și astfel

$$m = 2 \cdot \left\lceil \frac{x-1}{2} \right\rceil + 1,$$

unde, pentru orice număr $a \in \mathbb{R}$, $\lceil a \rceil$ reprezintă *partea întreagă superioară* a lui a, definită prin

$$\lceil a \rceil = \min\{k \mid k \in \mathbb{Z}, k \geq a\}$$

(deci $\lceil a \rceil \in \mathbb{Z}$ și $\lceil a \rceil - 1 < a \leq \lceil a \rceil$). De exemplu, avem:

$$\lceil 7 \rceil = 7,\ \lceil 7.4 \rceil = \lceil 7.6 \rceil = 8,\ \lceil -7.6 \rceil = \lceil -7.4 \rceil = \lceil -7 \rceil = -7.$$

Utilizând funcția $PRIM(x)$ definită în algoritmul din Exemplul 2.6.12, obținem următorul algoritm pentru rezolvarea problemei date.

Date de intrare: x;
Date de ieșire: p;
Variabile: $x \in \mathbb{R}$, $p \in \mathbb{Z}$;
Funcția $PRIM(x) \in \{0, 1\}$ **:** // $x \in \mathbb{Z}$
  ⋮ // ca în Exemplul 2.6.12
// continuăm algoritmul, în care vom apela funcția $PRIM$
citește x;
dacă $x \leq 2$ **atunci**
  $p \leftarrow 2$;
altfel
  $p \leftarrow 2 \times \left\lceil \frac{x-1}{2} \right\rceil + 1$; // cel mai mic număr impar mai mare
                                                  // sau egal cu x
  cât timp $PRIM(p) = 0$ **repetă** // p nu este prim
    $p \leftarrow p + 2$; // trecem la următorul număr impar
                                                  // p este prim
afișează p;

2.8 Instrucțiunea repetitivă cu test final

- *Sintaxa*:

repetă
  instrucțiuni
cât timp *condiție*

- *Efect*:

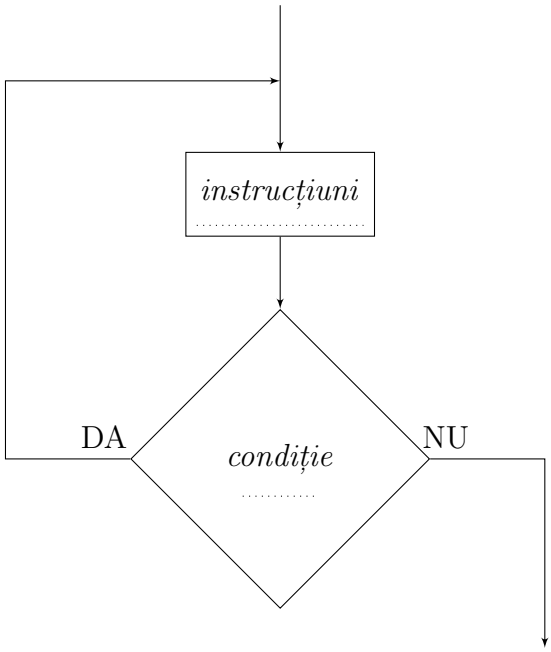
Pasul 1. Se execută instrucțiunile *instrucțiuni*;

Pasul 2. Se evaluează condiția (pentru valorile curente ale variabilelor componente);

Pasul 3. Dacă ea este îndeplinită (este adevărată), atunci se revine la Pasul 1 (adică se execută din nou instrucțiunile *instrucțiuni* apoi se evaluează condiția, ș.a.m.d., cât timp condiția este adevărată, adică până când ea devine falsă);

Pasul 4. În caz contrar, adică dacă ea nu este îndeplinită (este falsă), atunci aplicarea instrucțiunii repetitive cu test final se încheie.

- *Schema logică*:



Observația 2.8.1. Ca și în cazul instrucțiunii repetitive cu test inițial, condiția trebuie să devină falsă la un moment dat, deoarece în caz contrar executarea instrucțiunii repetitive cu test final nu s-ar încheia niciodată (spunem din nou că algoritmul intră într-un ciclu infinit, sau că *algoritmul ciclează la infinit*), deci nu s-ar respecta caracteristica de finitudine a algoritmului.

Observația 2.8.2. Spre deosebire de cazul instrucțiunii repetitive cu test inițial, acum instrucțiunile *instrucțiuni* sunt executate cel puțin o dată, și anume înainte de prima verificare a condiției (chiar și atunci când condiția este falsă de la prima verificare).

Observația 2.8.3. Conform Observațiilor 2.7.2 și 2.8.2, dacă instrucțiunile *instrucțiuni* ce trebuie executate în mod repetat pot să nu fie executate nici o dată, atunci trebuie utilizată instrucțiunea repetitivă cu test inițial, iar

dacă aceste instrucțiuni se execută cel puțin o dată, atunci este de preferat utilizarea instrucțiunii repetitive cu test final.

Exemplul 2.8.1. Să se calculeze suma cifrelor unui număr natural n dat.

De exemplu, pentru $n = 57047$ suma este $S = 5 + 7 + 0 + 4 + 7 = 23$.

Pentru rezolvare, extindem procedeul din Exemplul 2.4.4, calculând succesiv cifrele lui n, de la dreapta la stânga (de la ultima cifră până la prima).

Mai precis, pornind cu numărul dat, executăm repetat următoarele calcule:

- calculăm ultima cifră c a numărului curent m (ca rest al împărțirii lui m prin 10),
- o adunăm la suma S,
- actualizăm numărul curent m, eliminându-i ultima cifră (prin împărțire la 10),

cât timp numărul curent m mai are cifre necalculate, adică este mai mare ca zero (cu alte cuvinte, până când numărul curent devine egal cu zero, deoarece s-a obținut prin împărțirea la 10 a numărului format doar din prima cifră a lui n).

De exemplu, pentru $n = 57047$ valorile succesive ale numărului curent m și ale cifrei c sunt:

	$m = 57047$ (valoarea inițială);
$c = 57047$ MOD $10 = 7$,	$m = 57047$ DIV $10 = 5704$;
$c = 5704$ MOD $10 = 4$,	$m = 5704$ DIV $10 = 570$;
$c = 570$ MOD $10 = 0$,	$m = 570$ DIV $10 = 57$;
$c = 57$ MOD $10 = 7$,	$m = 57$ DIV $10 = 5$;
$c = 5$ MOD $10 = 5$,	$m = 5$ DIV $10 = 0$.

Obținem următorul algoritm pentru rezolvarea problemei date.

Date de intrare: n;
Date de ieșire: S;
Variabile: $n, S, m, c \in \mathbb{Z}$;
citește n;
$S \leftarrow 0$;
$m \leftarrow n$; // $m =$ `numărul curent`
repetă // `calculăm repetat:`
  $c \leftarrow m$ MOD 10; // $c =$ `ultima cifră a lui` m`,`
  $S \leftarrow S + c$; // `o adunăm la` S`,`
  $m \leftarrow m$ DIV 10; // `eliminăm ultima cifră din` m`,`
cât timp $m > 0$; // `cât timp există cifre necalculate`
// `toate cifrele lui` n `au fost calculate`
afișează S;

Exemplul 2.8.2. Să se calculeze *răsturnatul* unui număr natural n dat.

De exemplu, pentru $n = 57047$ răsturnatul este $r = 74075$.

Dacă un număr de cel puțin două cifre are ultima cifră egală cu zero, atunci se consideră că el nu are răsturnat.

Pentru rezolvare, utilizând procedeul din exemplul anterior, determinăm succesiv cifrele lui n, de la dreapta la stânga (de la ultima cifră până la prima) și construim/actualizăm simultan răsturnatul format cu cifrele determinate.

Mai precis, pornind cu $m = n$ (numărul dat) și $r = 0$, executăm repetat următoarele calcule:

- calculăm ultima cifră c a numărului curent m (ca rest al împărțirii lui m prin 10),
- actualizăm răsturmatul r format cu cifrele determinate, adăugându-i la final (drept ultimă cifră) cifra c (prin înmulțirea cu 10 și adunarea lui c),
- actualizăm numărul curent m, eliminându-i ultima cifră (prin împărțire la 10),

cât timp numărul curent m mai are cifre necalculate, adică este mai mare ca zero.

De exemplu, pentru $n = 57047$ valorile succesive ale numărului curent m,

ale cifrei c și ale răsturnatului r sunt:

$$r = 0,\ m = 57047;$$
$$c = 57047 \text{ MOD } 10 = 7,\ r = 0 \cdot 10 + 7 = 7,\ m = 57047 \text{ DIV } 10 = 5704;$$
$$c = 5704 \text{ MOD } 10 = 4,\ r = 7 \cdot 10 + 4 = 74,\ m = 5704 \text{ DIV } 10 = 570;$$
$$c = 570 \text{ MOD } 10 = 0,\ r = 74 \cdot 10 + 0 = 740,\ m = 570 \text{ DIV } 10 = 57;$$
$$c = 57 \text{ MOD } 10 = 7,\ r = 740 \cdot 10 + 7 = 7407,\ m = 57 \text{ DIV } 10 = 5;$$
$$c = 5 \text{ MOD } 10 = 5,\ r = 7407 \cdot 10 + 5 = 74075,\ m = 5 \text{ DIV } 10 = 0.$$

Obținem următorul algoritm pentru rezolvarea problemei date.
Date de intrare: n;
Date de ieșire: r;
Variabile: $n, r, m, c \in \mathbb{Z}$;
citește n;
dacă $n \geq 10$ *și* n *MOD* $10 = 0$ **atunci**
  afișează " nu există răsturnat";
altfel
  $m \leftarrow n$; // $m =$ `numărul curent`
  $r \leftarrow 0$;
  repetă // `calculăm repetat:`
    $c \leftarrow m$ MOD 10; // $c =$ `ultima cifră a lui` m`,`
    $r \leftarrow 10 \times r + c$; // `actualizăm răsturnatul` r`,`
    $m \leftarrow m$ DIV 10; // `eliminăm ultima cifră din` m`,`
  cât timp $m > 0$; // `cât timp există cifre necalculate`
  // `toate cifrele lui` n `au fost calculate`
  afișează r;

Tema 3

Limbajul C/C++

(sursa bibliografică: [2])

3.1 Prezentarea limbajului C/C++

Limbajul C este un limbaj de nivel înalt, de *programare structurată*, adecvat pentru *programare sistem* (scriere de sisteme de operare, compilatoare, editoare de texte, etc.). El a fost creat la începutul anilor 1970 de Ken Thompson și Dennis Ritchie pentru scrierea nucleului sistemului de operare UNIX.
Limbajul C++ este o extindere a limbajului C ce oferă posibilitatea *Programării Orientate pe Obiecte* (POO) (*obiect* = o structură de date împreună cu metode de operare cu aceste date). El a fost dezvoltat de Bjarne Stroustrup la începutul anilor 1980.
Sintaxa limbajului C stă la baza și a altor limbaje de programare foarte utilizate, precum C#, Java, JavaScript.

Alfabetul limbajului C++

Orice caracter este reprezentat în calculator în codul ASCII (*American Standard Code for Information Interchange*), printr-un număr natural unic, cuprins între 0 şi 255 .
Caracterele limbajului C++ sunt:

- litere:
  - literele mari ale alfabetului englez: A, B, ..., X, Y, Z (cu codurile 65,...,90)
  - literele mici ale alfabetului englez: a, b, ..., x, y, z (cu codurile 97,...,122)
- cifre: cifrele bazei zece: 0, 1, 2, ... ,9 (cu codurile 48,...,57)
- simboluri speciale: semne de punctuaţie şi semne speciale : , . : ? ‘ ( ) [ ] { } < > ! | \ / ~ # & ^ + - * % _

Vocabularul limbajului C++

Unităţile lexicale (*cuvintele*) ale limbajului C++ sunt:

- identificatori
- cuvinte cheie
- separatori:
  - spațiu
  - TAB
  - trecere la linie nouă
  - comentarii
- constante
- operatori

La scrierea lor se utilizează setul de caractere al codului ASCII și se respectă regulile precise date de sintaxa limbajului.

Identificatori

Un *identificator* reprezintă o succesiune de litere (litera mică este tratată ca distinctă de litera mare), cifre, liniuţe de subliniere (_), primul caracter din secvenţă fiind obligatoriu o literă sau o liniuţă de subliniere.
Identificatorii sunt *nume simbolice* date de programator constantelor, variabilelor, tipurilor de date, funcţiilor etc, pentru a descrie datele de prelucrat (de exemplu nume de variabile) şi procesele de prelucrare (de exemplu nume de funcţii). În general numai primele 32 de caractere se consideră semnificative în C++.
Ca identificatori se preferă folosirea unor nume sau simboluri care să sugereze semnificaţia mărimilor pe care le desemnează, contribuind la creşterea clarităţii programului.

Cuvinte cheie

Cuvintele cheie (keywords) sunt cuvinte rezervate pentru limbaj în sine, au înţeles predefinit şi nu pot avea altă utilizare. Aceste cuvinte se scriu cu litere mici.
ANSI (American National Standards Institute) C are 32 de cuvinte cheie:

```
auto      const     double    float     int       short
struct    unsigned  break     continue  else      for
long      signed    switch    void      case      default
enum      goto      register  sizeof    typedef   volatile
char      do        extern    if        return    static
union     while
```

Comentarii

În redactarea programelor se folosesc o serie de texte care dau explicaţii cu privire la program, la părţile sale, la variabilele utilizate etc. Aceste texte explicative se adresează utilizatorilor, se numesc comentarii şi sunt ignorate de compilator.
În limbajul C un comentariu începe prin `/*` şi se termină prin `*/`.
În C++ un comentariu care încape pe un rând poate fi scris după `//`.

3.2 Tipuri de date, constante, variabile

Limbajul C lucrează cu date (valori) care pot fi stocate în:

- constante
- variabile.

Constantele stochează valori nemodificabile pe parcursul execuţiei programului.
Variabilele sunt mărimi care îşi pot modifica valoarea în timpul execuţiei programului.
Fiecare dată (constantă sau variabilă) are:

- un identificator (nume)
- un tip
- o declarare.

Tipul unei date determină:

- spaţiul de memorie ocupat;
- modul de reprezentare internă;
- domeniul de valori;

- timpul de viaţă asociat datei;
- operatorii ce pot fi utilizaţi şi restricţii în folosirea acestora.

Tipurile de date utilizate de limbajul C se clasifică astfel:

- *tipuri fundamentale (scalare, predefinite, simple, de bază):*
  - caracter
  - întregi
  - reale
  - tip void (vid, fără tip)
- *tipuri derivate:*
  - tablouri
  - șiruri de caractere (stringuri)
  - pointeri
  - structuri
  - uniuni
  - enumerări
  - definite de programator

Tipuri de date standard

Datele reprezintă informaţii care fac obiectul prelucrărilor. Fiecare dată este memorată într-un anumit format. Pe de altă parte, interpretarea valorilor memorate se face diferit, în funcţie de semnificaţia datelor respective. Prin urmare, este important atât modul de memorare a datelor (formatul fizic de reprezentare) prin care se stabilește domeniul valorilor datelor, cât şi semnificaţia lor prin care se stabilesc operaţiile care se pot efectua cu aceste date. Aceste caracteristici ale unei date sunt precizate prin tipul său.

Tipurile de date standard (predefinite) ale limbajului C sunt:

Specificator	Abreviaţia	Lungime în biţi	Domeniu de valori
signed char	char	8	Caracter reprezentat prin cod ASCII sau întreg binar din intervalul –128 ... 127.
unsigned char		8	Caracter reprezentat prin cod ASCII sau întreg binar fără semn din intervalul 0 ... 255.
signed int	int	dependentă de calculator 16 sau 32	Întreg binar reprezentat în cod complementar faţă de 2
short int	short	16	Întreg binar reprezentat în cod complementar faţă de 2, din intervalul -32768 ... 32767.
long int	long	32	Întreg binar reprezentat în cod complementar faţă de 2 din intervalul -2147483648 ... 2147483647
unsigned int	unsigned	dependentă	Întreg binar fără semn

		de calculator 16 sau 32	
unsigned short int	unsigned short	16	Întreg binar fără semn din intervalul 0 ... 65535.
unsigned long int	unsigned long	32	Întreg binar fără semn din intervalul 0 ... 4294967295
float		32	Număr reprezentat în virgulă mobilă 1bit pentru semn, 7b exponent, 24b mantisa, precizie 7 zecimale. Domeniu [3.4E-38, 3.4E38]
long float	double	64	Număr reprezentat în virgulă mobilă 1 bit ptr. semn, 11b exponent, 52b mantisa, precizie 15 zecimale. Domeniu [1.7E-308, 1.7E308]
long double		80	Număr reprezentat în virgulă mobilă precizie 19 zecimale. Domeniu [3.4E-4932, 1.1E4932]

Pe lângă aceste tipuri de date, limbajul C mai dispune de tipul *void*. Tipul *void* indică absenţa oricărei valori.

Pointerii se utilizează pentru a face referire la date cunoscute prin adresele lor. Un *pointer* este o variabilă care are ca valori adrese.
Tipul pointer are formatul:

```
<tip>* <nume>;
```

ceea ce înseamnă că <*nume*> este un pointer către o zonă de memorie ce conţine o dată de tipul <*tip*>.

Constante

O constantă este o valoare fixă care apare literalmente în codul sursă al unui program. Tipul și valoarea constantei sunt determinate de modul în care constanta este scrisă. Constantele pot fi de mai multe tipuri: întreg, flotant (real), caracter, șir de caractere. Aceste constante sunt folosite, de exemplu, pentru a iniţializarea variabilelor.

Constante întregi

O constantă întreagă este un număr întreg reprezentat în cod complementar faţă de 2 pe 16 biţi sau pe 32 biţi dacă nu încape pe 16 biţi. Exemple: 7, -3, 0.
În cazul în care dorim ca o constantă întreagă din intervalul -32768 ... +32767 să fie reprezentată pe 32 biţi (implicit astfel de constante se reprezintă pe 16 biţi), vom termina constanta respectivă prin L sau l, adică îi impunem tipul long. Exemplu:`10L`.
Dacă, la o constantă întreagă, adăugăm sufixul U sau u, atunci forţăm tipul constantei la unsigned int sau unsigned long. Dacă adăugăm sufixul UL sau ul sau Ul sau uL constanta va fi de tipul unsigned long.

O constantă întreagă, precedată de un zero nesemnificativ se consideră scrisă în sistemul de numeraţie cu baza 8.
O constantă întreagă care începe cu `0X` sau `0x` se consideră scrisă în sistemul de numeraţie cu baza 16 (cifrele hexazecimale sunt `0...9`, `a...f` sau `A...F`). În rest se consideră că baza de numeraţie este 10.
Exemple:

Constanta	Tip	Constanta	Tip
1234	int	123456789L	long
02322	int /* octal */	1234U	unsigned int
0x4D2	Int /* hexazecimal */	123456789UL	unsigned long int

Constante reale (flotante)

Atunci când încercăm să reprezentăm în memoria calculatorului un număr real, căutăm de fapt cel mai apropiat număr real reprezentabil în calculator şi aproximăm numărul iniţial cu acesta din urmă. Ca rezultat, putem efectua calcule complexe cu o precizie rezonabilă.
Constantele reale sunt reprezentate în virgulă mobilă prin notaţia clasică cu mantisă şi exponent.
Sintaxa:
±partea întreagă . partea fracţionară {E|e} ±exponentul

Pot lipsi fie partea întreagă, fie partea fracţionară, dar nu ambele.
Pot lipsi punctul zecimal cu partea fracţionară sau litera E cu exponentul dar nu ambele. Semnul + este opţional pentru numerele nenegative.

Exemple:

```
3.1415921    -12.          .34       -.125
4.3E20                     /* pentru numărul 4.3*10^20 */
-.2e+15                    /* pentru numărul –0.2*10^15 */
2e-5                       /* pentru numărul 2*10^-5 */
```

În mod implicit, o constantă reală este reprezentată intern în format double. Tipul constantei poate fi influenţat prin adăugarea unui sufix de f (sau F) sau l(sau L). Sufixul f (saui F) forţează constanta la tipul float, sufixul l (sau L) forţează constanta la tipul long double.

Constanta	tip
12.34	double
12.3e-4	double
12.34F	float
12.34L	long double

Constante caracter

O constantă caracter reprezintă un caracter şi are ca valoare codul ASCII al caracterului respectiv.
O constantă *caracter grafic* se poate scrie incluzând caracterul respectiv între caractere apostrof (').

Exemple:

- **literele mari au codurile ASCII în intervalul [65,90]**

'A' —> 65, ... , 'Z' —> 90.

- **literele mici au codurile ASCII în intervalul [97,122]**

'a' —> 97, ... ,'z' —> 122

- **cifrele au codurile ASCII în intervalul [48,57]**

'0' —> 48, ... ,'9' —> 57

- **constanta '*' are valoarea 77.**

Caracterele negrafice cu excepţia caracterului DEL (care are codul 127), au coduri ASCII mai mici decât 32. O parte dintre aceste caractere formează categoria *caracterelor de control* şi au notaţii speciale de tipul `\caracter`.
De exemplu codul ASCII de valoare zero defineşte caracterul NULL. Acesta este un caracter impropriu şi spre deosebire de alte caractere el nu poate fi generat de la tastatură şi nu are efect nici la ieşire. În C este folosit ca terminator pentru şiruri de caractere şi are notaţia \0.

Setul de caractere de control:

Valoare cod ASCII	Reprezentare	Rol
0	\0	Caracterul NULL (zero binar)
7	\a	Alarmă (bell)
8	\b	Spaţiu înapoi (backspace); BS
9	\t	Tabulator orizontal; TAB
10	\n	Salt la linie nouă (new line)
11	\v	Tabulator vertical
12	\f	Salt de pagină la imprimantă (formfeed); FF
13	\r	Deplasarea cursorului în coloana 1 pe aceeaşi linie; CR

O constantă caracter cu notaţie specială, se va scrie incluzând notaţia între caractere apostrof. Exemple: '\n', '\t', '\r'
Constanta *apostrof* se reprezintă prin `'\''`.
Constanta *backslash* se reprezintă prin `'\\'`.
Construcţia `'\ddd'`, unde `d` este o cifră octală, reprezintă caracterul al cărui cod ASCII are valoarea egală cu numărul octal ddd. În particular caracterul impropriu NULL se poate reprezenta prin constanta caracter `'\0'`.
Caracterul DEL al cărui cod ASCII are valoarea 127 se reprezintă prin `'\177'`.
Caracterul spaţiu al cărui cod ASCII are valoarea 32 se poate reprezenta prin `' '` sau `'\40'`.
Construcţia `'\xdd'`, unde d reprezintă o cifră hexazecimală reprezintă caracterul al cărui cod ASCII are valoarea egală cu numărul zecimal dd.
Exemplu: `'\x20'` reprezintă caracterul spaţiu.
Dacă `\` este urmat de un alt caracter decât cele arătate, atunci `\` este ignorat de compilator.
Caracterele *spaţiu* (cod ASCII 32), *tab* (cod ASCII 9) şi *linie nouă* (cod ASCII 10) formează categoria de separatori "*spaţii albe*". În afară de locurile unde sunt necesare spaţiile albe pentru separarea identificatorilor, a cuvintelor cheie etc. aceste spaţii albe sunt ignorate de compilator şi pot fi folosite oriunde în program.

Constante şir de caractere

O constantă *şir de caractere* este o succesiune de zero sau mai multe caractere delimitate prin ghilimele ("). Ghilimelele nu fac parte din şirul de caractere. Dacă dorim să folosim caractere negrafice în compunerea unui şir de caractere, atunci putem folosi convenţia de utilizare a caracterului \. Dacă dorim să reprezentăm chiar caracterul ghilimele, atunci vom scrie \", de asemenea pentru backslash scriem \\.
Exemple:

```
"123"
"1\"2"          -reprezintă succesiunea 1"2
"a\\b"          -reprezintă succesiunea a\b
"c:\\tc\\bg"    -reprezintă succesiunea c:\tc\bgi
```

Un şir de caractere poate fi continuat de pe un rând pe altul, dacă înainte de a acţiona tasta <enter> se va tasta \.
Constanta şir de caractere se reprezintă printr-o succesiune de octeţi în care se păstrează codurile ASCII ale caracterelor şirului, iar ultimul octet conţine totdeauna caracterul NULL pentru a marca sfârşitul şirului. De aici rezultă că, de exemplu, `'A'` şi `"A"` sunt construcţii diferite. Prima reprezintă o constantă caracter care se păstrează pe un singur octet în memorie. A doua, reprezintă un şir de caractere şi se păstrează pe doi octeţi, primul octet conţine valoarea codului ASCII al lui A, iar cel de-al doilea conţine caracterul NULL, adică valoarea 0.

Variabile

Prin variabilă înţelegem o zonă temporară de stocare a datelor a cărei valoare se poate schimba în timpul execuţiei programului. Unei variabile i se asociază un nume (*identificator*) prin intermediul căruia putem avea acces la valoarea ei şi un tip de date care stabileşte valorile pe care le poate lua variabila. Corespondenţa între numele şi tipul unei variabile se realizează printr-o construcţie specială numită *declaraţie*. Toate variabilele utilizate într-un program trebuie declarate înaintea utilizării lor.
O **declaraţie de variabilă** are următoarea sintaxă:

< tip > < lista de variabile >

unde lista conţine unul sau mai multe nume de variabile despărţite prin virgule.

Exemple:

```
int i,j,X;
unsigned long k;
float a,b;
char c;
```

Sunt situaţii în care variabilele trebuie să fie grupate din punct de vedere logic. **Tablourile** reprezintă grupuri unidimensionale sau multidimensionale de variabile de acelaşi tip. Declaraţia unui tablou conţine tipul comun al elementelor sale, numele tabloului şi numărul de elemente pentru fiecare dimensiune incluse între paranteze drepte.

```
<tip>  <lista de elemente>;
```

Elementele se separă prin virgule.
Un element din lista de tip tablou are formatul:

```
nume[dim1][dim2]...[dimn]
```

unde $dim_1, dim_2, ..., dim_n$ sunt expresii constante care au valori întregi.

Exemple:
```
int v[10];
float a[100][3];
```

La elementele unui tablou ne referim prin variabile cu indici. O astfel de variabilă se compune din numele tabloului urmat de unul sau mai mulţi indici, fiecare indice fiind inclus între paranteze drepte. Indicii au limita inferioară zero.

Exemple: Tablourile `v` şi `a` declarate mai sus se compun din variabilele
```
v[0], v[1], ... , v[9]
```
respectiv,
```
a[0][0],   a[0][1],   a[0][2],
a[1][0],   a[1][1],   a[1][2],
                ...
a[99][0]   a[99][1],   a[99][2].
```

Tablourile unidimensionale de tip caracter se utilizează pentru a păstra şiruri de caractere.

Exemplu:
```
char tab[4];
```
tab poate păstra un şir de maxim 3 caractere, al patrulea octet fiind necesar pentru caracterul NULL - marcatorul sfârşitului şirului.
Numele unui tablou are ca valoare adresa primului său element.

3.3 Structura unui program C/C++

Prin *program* înţelegem un text ce specifică acţiuni ce vor fi executate de un procesor. Limajul C este un limbaj procedural ceea ce înseamnă că structura programelor scrise în C se bazează pe subprograme.

Un *subprogram* este o secvenţă de declaraţii şi instrucţiuni care formează o structură unitară, ce rezolvă o problemă de complexitate redusă, putând fi innclus într-un program sau stocat în biblioteci, compilat separat şi utilizat ori de câte ori este nevoie.

Acţiunile sunt descrise cu ajutorul instrucţiunilor.

In limbajul C subprogramele sunt realizate sub formă de *funcţii*. Un program C se compune din una sau mai multe funcţii. Fiecare funcţie are un nume.

Orice program scris în limbajul C, indiferent de complexitate, trebuie să conţină o funcţie, numită funcţie principală, al cărui nume este *main*. Această funcţie preia controlul de la sistemul de operare în momentul în care programul este lansat în execuţie şi îl redă la terminarea execuţiei.

Execuţia unui program C înseamnă execuţia instrucţiunilor din funcţia `main`.

Structura unei funcţii este următoarea:
```
<tip> <nume>( <lista parametrilor formali>)    //antet
  {
   <declaraţii şi instrucţiuni>      //corpul funcţiei
  }
```

Structura generală a unui program C este următoarea:

```
<directive preprocesor>

<declaratii globale>

<tip> <functie1> (<listă parametri>)
  {
    <declaraţii locale si instrucţiuni>
  }
  ...

<tip> <functien> (<listă parametri>)
  {
    <declaraţii locale si instrucţiuni>
  }

<tip> main (<listă parametri>)
  {
    <declaraţii locale si instrucţiuni>
  }
```

Realizarea unui program C/C++ parcurge următoarele etape:

- ***editarea* fişierului sursă, care constă în scrierea programului folosind regulile de sintaxă ale editorului de texte corespunzător;**
- ***compilarea* fişierului sursă, un program specializat numit compilator, transformă instrucţiunile programului sursă în instrucţiuni maşină. Dacă nu detectează erori sintactice, el va genera un fişier numit fişier obiect, care are numele fişierului sursă şi extensia .obj.**
- ***editarea legăturilor,* un program specializat numit linkeditor, asamblează mai multe module obiect şi generează programul executabil sub forma unui fişier cu extensia .exe. Pot să apară erori generate de incompatibilitatea modulelor obiect asamblate.**
- ***lansarea în execuţie* – programul se află sub formă executabilă şi poate fi lansat în execuţie. În această etapă pot să apară erori fie datorită datelor eronate introduse în calculator, fie conceperii greşite a programului.**

În limbajul C există două categorii de funcţii:

- **funcţii care produc (returnează) un rezultat direct ce poate fi utilizat în diverse expresii. Tipul acestui rezultat se defineşte prin <tip> din antetul funcţiei. Dacă <tip> este absent, se presupune că funcţia returnează o valoare de tip int.**
- **funcţii care nu produc un rezultat direct. Pentru aceste funcţii se va folosi cuvântul cheie void în calitate de tip. El semnifică lipsa valorii returnate la revenirea din funcţie.**

O funcţie poate avea zero sau mai mulţi parametri separaţi prin virgule.

Dacă o funcţie are lista parametrilor formali vidă, antetul său se reduce la:

```
<tip> <nume>()
```

Absenţa parametrilor formali poate fi indicată explicit folosind cuvântul cheie `void`. Astfel, antetul de mai sus poate fi scris şi sub forma:

```
<tip> <nume>(void)
```

Exemplu:

```
float putere(float x, long n)
  {
```

```
    ...
}
```

3.4 Instrucțiunile de bază ale limbajului C/C++

Instrucțiune	Pseudocod	Limbajul C/C++
De citire	**citește** var1, var2, var3	**cin**>>var1>>var2>>var3;
De scriere	**afișează** expr1, expr2, expr3	**cout**<<expr1<<expr2<<expr3;
Compusă	instr1 instr2 instr3	{ instr1; instr2; instr3; }
De atribuire	v←expresie	v=expresie;
De decizie	**dacă** condiție **atunci** instr1 **altfel** instr2	**if** (condiție) instr1 **else** instr2
	dacă condiție **atunci** instr1	**if** (condiție) instr1
Repetitivă cu contor	**pentru** contor=val i, val f, pas **execută** instr	**for**(contor=val i; contor<=val f; contor+=pas) instr // pt. pas>0
		for(contor=val i; contor>=val f; contor-=pas) instr // pt. pas<0
	pentru contor=val i, val f **execută** instr	**for**(contor=val i; contor<=val f; contor++) instr
	pentru contor=val i, val f, -1 **execută** instr	**for**(contor=val i; contor>=val f; contor--) instr
Repetitivă cu test inițial	**cât timp** condiție **repetă** instr	**while** (condiție) instr
Repetitivă cu test final	**repetă** instr **cât timp** condiție	**do** instr **while** (condiție);
De revenire dintr-o funcție	**returnează** expresie	**return** expresie;

3.5 Exemple de programe în limbajul C++

Programele C++ vor fi scrise în **Code::Blocks** – o platformă open-source care permite dezvoltarea aplicațiilor C++.

Exemplul 1: Calculul sumei $S = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \ldots + (-1)^{n-1} \cdot \frac{1}{n}$, unde $n \in \mathbf{N}^*$ este dat.

Programul C++, corespunzător algoritmului din ***Exemplul 2.6.3***:

```
#include <iostream>
#include <math.h>
using namespace std;
 // in Code::Blocks, biblioteca standard C++ este declarata in
 // spatiul de nume (namespace) std
```

```
int main()
{   int n,k;
    float S;
    cout<<"Dati n="; cin>>n;
    S=0;
    for(k=1;k<=n;k++)
        if (k%2==0) S=S-1./k; // k%2 =restul impartirii lui k la 2
        else S=S+1./k;
    cout<<"S="<<S;
    return 0;
}
```

Exemplul 2: Calculul sumei $S = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots + \frac{1}{n!}$, unde $n \in \mathbf{N}^*$ este dat.

Programul C++, corespunzător algoritmului din ***Exemplul 2.6.9***:

```
#include<iostream>
#include<conio.h>
#include<math.h>
using namespace std;
int main(void)
{ int n,k,i;
  double S,P;
  cout<<"Dati n="; cin>>n;
  S=0;
  for(k=0;k<=n;k++)
  {  P=1;
     for(i=2;i<=k;i++)
        P=P*i;
     S=S+1/P;
  }
  cout<<"Suma este S="<<S;
  return 0;
}
```

O variantă îmbunătățită, cu un singur ciclu *for*, este următoarea:

```
#include<iostream>
#include<conio.h>
#include<math.h>
using namespace std;
int main(void)
{ int n,k,i;
  double S,T;  // T=1/k!
  cout<<"Dati n="; cin>>n;
  S=2; T=1;
  for(k=2;k<=n;k++)
  {  T=T/k;
     S=S+T;
  }
  cout<<"Suma este S="<<S;
  return 0;
}
```

Exemplul 3: Calculul sumei elementelor impare dintr-un vector v = (v_1, v_2, . . . , v_n) cu elemente întregi ($n \geq 1$).
Programul C++, corespunzător algoritmului din ***Exemplul 2.6.13***:

```
#include<iostream>
#include<conio.h>
#include<math.h>
#define DIM 20
using namespace std;
int main(void)
{ int n,S,nr,i,v[DIM];
  cout<<"Dati numarul de elemente n="; cin>>n;
  cout<<"Dati elementele (separate prin spatii):"<<endl;
  for(i=1;i<=n;i++)
     cin>>v[i];
  S=0; nr=0;
  for(i=1;i<=n;i++)
    if(v[i]%2!=0)
    {  nr=nr+1;
       S=S+v[i];
    }
  if (nr==0) cout<<"Vectorul nu contine elemente impare";
  else cout<<"Suma este S="<<S;
  return 0;
}
```

Exemplul 4: Calculul mediilor aritmetică și geometrică a elementelor pozitive dintr-un vector a = (a_1, a_2, . . . , a_n) cu elemente reale ($n \geq 1$).
Programul C++, corespunzător algoritmului din ***Exemplul 2.6.14***:

```
#include<iostream>
#include<conio.h>
#include<math.h>
#define DIM 20
using namespace std;
int main(void)
{ int n,nr,i;
  double MA,MG,S,P,a[DIM];
  cout<<"Dati numarul de elemente n="; cin>>n;
  cout<<"Dati elementele (separate prin spatii):"<<endl;
  for(i=1;i<=n;i++)
     cin>>a[i];
  nr=0; S=0; P=1;
  for(i=1;i<=n;i++)
    if(a[i]>0)
    {  S=S+a[i];
       P=P*a[i];
       nr=nr+1;
    }
  if (nr==0) cout<<"Vectorul nu contine elemente pozitive";
  else
  { MA=S/nr;
    MG=exp(log(P)/nr); // sau MG=pow(P,1./nr);
    cout<<"Mediile sunt MA="<<MA<<"; MG="<<MG;
  }
  return 0;
}
```

Exemplul 5: Calculul celui mai mic număr prim *p* mai mare sau egal cu un număr real *x* dat.
Programul C++, corespunzător algoritmului din ***Exemplul 2.7.2***:

```
#include<iostream>
#include<conio.h>
#include<math.h>
```

```
using namespace std;

int PRIM(long x) // functie ce are rezultatul 1 daca x este prim,
                 // respectiv 0 daca x nu este prim
{ int k;
  if (x<2) return 0;
  if (x<4) return 1;
  if(x%2==0) return 0;
  for(k=3;k<=floor(sqrt(x));k+=2) // sqrt(x)=radical din x;
                                  // floor(x)=partea intreaga a lui x
     if (x%k==0) return 0;      // x%k = restul impartirii lui x la k
  return 1;
}

int main(void)
{ double x;
  long p;
  cout<<"Dati x="; cin>>x;
  if (x<=2)
     p=2;
  else
  {  p=2*ceil((x-1)/2)+1;//ceil(x)=partea intreaga superioara a lui x
     while(PRIM(p)==0)
        p=p+2;
  }
  cout<<"Primul numar prim mai mare sau egal cu "<<x<<" este p="<<p;
  return 0;
}
```

3.6. Preprocesare

Un program C poate suporta anumite prelucrări înainte de compilare. O astfel de prelucrare se numeşte preprocesare. Ea se realizează printr-un program special numit preprocesor. Preprocesorul este apelat automat înainte de a începe compilarea.

Prin intermediul preprocesorului se pot realiza:

- **includeri de fişiere standard şi utilizator;**
- **definirea de macrodefiniţii;**
- **compilare condiţionată.**

Includeri de fişiere

Fişierele se includ cu ajutorul construcţiei #include folosindu-se formatele:

```
#include<specificator_de_fişier>
#include "specificator_de_fişier"
```

Preprocesorul localizează fişierul şi înlocuie construcţia include cu textul fişierului localizat. În felul acesta compilatorul C nu va mai întâlni linia `#include`, ci textul fişierului inclus de preprocesor.

Prima variantă se foloseşte pentru încorporarea fişierelor standard ce se găsesc în bibliotecile ataşate mediului de programare. A doua variantă se foloseşte uzual pentru încorporarea fişierelor create de utilizator; dacă nu este specificată calea atunci fişierul este căutat în directorul curent şi în bibliotecile ataşate mediului de programare.

Includerile de fişiere se fac, de obicei, la începutul fişierului sursă. Textul unui fişier inclus poate să conţină construcţia `#include` în vederea includerii altor fişiere.

Exemple:

```
#include<iostream>
#include<math.h>
#include"geo.cpp"
```

Macrodefiniţii

O altă construcţie tratată de preprocesor este construcţia `define` cu formatul:

```
#define <nume> <succesiune de caractere>
```

Folosind această construcţie, preprocesorul substituie `<nume>` cu `<succesiune de caractere>` peste tot în textul sursă care urmează, exceptând cazul în care `<nume>` apare într-un şir de caractere sau într-un comentariu. Dacă succesiunea de caractere nu încape pe un rând ea poate fi continuată terminând rândul cu "\".

Se recomandă ca `<nume>` să se scrie cu litere mari; <succesiune de caractere> poate conţine alte macrodefiniţii care trebuie să fie în prealabil definite.

O macrodefiniţie este definită din punctul construcţiei `#define` şi până la sfârşitul fişierului sursă respectiv sau până la redefinirea ei sau până la anihilarea ei prin intermediul construcţiei: `#undef <nume>`

Exemple:

```
#define PI5 3.14159
#define DIM 100
#define A 123
#define B A+120
...
x=3*B       // se substituie prin x=3*123+120
#define A 123
#define B (A+120)
...
x=3*B       // se substituie prin x=3*(123+120)
```

Macrodefiniţii cu argumente

Directiva `#define` poate fi folosită şi în sintaxa:

```
#define <nume>(<listă de parametri>) <corp macrodefinţie>
```

între <nume> şi "(" nu există spaţii.

Exemplu:

```
#define MAX(a,b) ((a)>(b)?(a):(b))
...
x=max(k+5,m)
```

Compilare condiţionată

Compilarea condiţionată se realizează folosind construcţiile:

1.

```
#if  expresieConstantă   //dacă expresie este diferită de zero
  text
#endif
```

2.

```
#if  expresieConstantă
   text1
#else
   text2
#endif
```

3.

```
#ifdef  identificator  //dacă identificator a apărut într-o directivă #define
   text
#endif
```

4.

```
#ifdef  identificator  // dacă identificator a apărut într-o directivă  #define
   text1
#else
   text2
#endif
```

5.

```
#ifndef  identificator  //dacă identificator nu a apărut într-o directivă #define
   text
#endif
```

6.

```
#ifndef  identificator  //dacă identificator nu a apărut într-o directivă  #define
   text1
#else
   text2
#endif
```

Pentru toate directivele `if` liniile care urmează până la o directivă `#endif` sau `#else` sunt supuse preprocesării dacă condiţia testată este satisfăcută şi sunt ignorate dacă condiţia nu este satisfăcută. Liniile dintre `#else` şi `#endif` sunt supuse preprocesării dacă condiţia testată de directiva #if nu este satisfăcută.

Exemplu:

```
#ifndef tipData
   #define tipData long
#endif
tipData x;
```

3.7 Operatori şi expresii

Operatorii sunt simboluri care specifică operaţiile ce se aplică unor variabile sau constante numite *operanzi*.
O *expresie* este o construcţie aritmetică sau algebrică care defineşte un calcul prin aplicarea unor operatori asupra unor termeni care pot fi: constante, variabile, funcţii.
Expresiile se evaluează pe baza unui set de reguli care precizează prioritatea şi modul de asociere a operatorilor precum şi conversiile aplicate operanzilor:

- **prioritatea determină ordinea de efectuare a operaţiilor într-o expresie cu diverşi operatori.**
- **modul de asociere indică ordinea de efectuare a operaţiilor într-o secvenţă de operaţii care au aceeaşi prioritate.**

În tabelul de mai jos se indică operatorii C++ în ordinea descrescătoare a priorităţii lor. Operatorii din aceeaşi categorie au aceeaşi prioritate. Operatorii de aceeaşi prioritate sunt prelucrati în ordinea de la stânga la dreapta sau la dreapta la stânga în direcţia indicată de sageată.

Categoria de operatori	Operatori	Prio-ritate	Mod de asociere
Primari: Apel de funcţie Indice de tablou Operator rezoluţie Referinţă la membru de structură Referinţă indirectă la membru structură	 ( ) [ ] :: . ->	15	→
Unari: Stabilirea tipului Dimensiune în octeţi Alocare memorie Dezalocare memorie Adresă Conţinut adresă Semn Negaţie Incrementare, decrementare	 (tip) sizeof new delete & * + - ! ~ ++ --	14	←
Dereferenţierea pointerilor spre membrii claselor	.* ->*	13	→
Multiplicativi	* / %	12	→
Aditivi	+ -	11	→
Deplasare	<< >>	10	→
Relaţionali	< <= > >=	9	→
Egalitate	= = !=	8	→
ŞI la nivel de bit	&	7	→
SAU EXCLUSIV la nivel de bit	^	6	→
SAU la nivel de bit	|	5	→
ŞI logic	&&	4	→
SAU logic	||	3	→
Condiţional	?:	2	←
Atribuire	= += -= *= /= %= &= |= ^= <<= >>=	1	←
Operatorul virgulă	,	0	→

Operatori aritmetici

Operatorii + şi – unari se aplică unui singur operand şi se folosesc la stabilrea semnului operandului: pozitiv sau negativ.
Operatorul * reprezintă operatorul de înmulţire al operanzilor la care se aplică.
Operatorul / reprezintă operatorul de împărţire. Dacă ambii operanzi sunt întregi (char, int, unsigned, long), se realizează o împărţire întreagă, adică ne furnizează câtul împărţirii.
Operatorul % are ca rezultat restul împărţirii dintre doi operanzi întregi.
Operatorii binari + şi – reprezintă operaţiile obişnuite de adunare şi scădere.

Exemple:
int a, b;
float x, y;
Dacă a=3 şi b=7 atunci b/a are valoarea 2 iar b%a are valoarea 1.
Dacă x=9 şi y=2 atunci x/y are valoarea 4.5.
Expresia x*–y are sens, aici – este operatorul unar.

Operatori de incrementare / decrementare

Sunt operatori unari. Operandul asupra căruia se aplică trebuie să fie o variabilă întreagă sau flotantă. Operatorul de incrementare se notează prin ++, şi măreşte valoarea operandului cu 1, iar cel de decrementare se notează cu --, şi micşorează valoarea operandului cu 1.
Operatorii pot fi folosiţi prefixaţi:

```
++operand
--operand
```

sau postfixaţi:

```
operand++
operand--
```

În cazul în care sunt folosiţi postfixaţi, ei produc ca rezultat valoarea operandului şi apoi incrementează/decrementează operandul. Când se folosesc prefixaţi se incrementează/decrementează operandul după care produc ca rezultat valoarea incrementată/decrementată.

Exemple:

Expresie	Efect
j=i++	j=i; i=i+1;
y=--x	x=x-1; y=x;
x=v[3]--	x=v[3]; v[3]=v[3]-1;
x=++v[++j]	j=j+1; v[j]=v[j]+1; x=v[j];
y=++i-j	i=i+1; y=i-j;
y=i++-j	y=i-j; i=i+1;
y=(i-j)++	construcţie eronată

Operatori relaţionali

Operatorii relaţionali sunt `<, <=, >, >=, ==, !=.`
Rezultatul aplicării unui operator relaţional este 1 sau 0 după cum operanzii se află în relaţia definită de operatorul respectiv sau nu.
De exemplu, dacă a=5 şi b=6 atunci expresia a<=b are valoarea 1, iar expresia a+1>b are valoarea 0.
Operatorul == (egal) furnizează 1 dacă operanzii sunt egali şi zero în caz contrar.
Operatorul != (*diferit*) furnizează 1 dacă operanzii nu sunt egali şi zero în caz contrar.

Exemple:
Dacă x=2 şi y =-1 atunci expresia x==y are valoarea 0, expresia x!=y are valoarea 1, expresia x+y==1 are valoarea 1.

Operatori logici

! – *negaţie logică*,operator unar.
&& – *ŞI logic*.
|| – *SAU logic*.

În limbajul C nu există valori logice speciale. Valoarea fals este reprezentată prin 0. Orice valoare diferită de 0 reprezintă valoarea adevărat. Operatorii logici admit operanzi de orice tip scalar. Rezultatul evaluării unei expresii logice este de tip întreg: zero pentu fals şi 1 pentru adevărat.
Dacă la evaluarea unei expresii logice se ajunge într-un punct în care se cunoaşte valoarea întregii expresii, atunci restul expresiei nu se mai evaluează.

Exemple:
Dacă a şi b sunt ambii diferiţi de zero expresia a&&b are valoarea 1, altfel 0.
Dacă a este negativ, expresia !(a<0) are valoarea 0, altfel 1.
Expresia !a&&b||a&&!b realizează SAU EXCLUSIV. Dacă a=0 şi b=1, deoarece !a&&b are valoarea 1 şi deci rezultatul întregii expresii este 1, subexpresia a&&!b nu se mai evaluează.

Operatori logici pe biţi

Constituie una din extensiile limbajului C spre limbajele de asamblare. Se aplică operanzilor de tip întreg, execuţia făcîndu-se bit cu bit, cu ajutorul celor patru operaţii logice şi două operaţii de deplasare la stînga şi la dreapta.
Se utilizează următoarele simboluri:
& pentru ŞI
| pentru SAU
^ pentru SAU EXCLUSIV
~ pentru NEGARE(complement faţă de unu, schimbă fiecare bit 1 al operandului în 0 şi fiecare bit 0 în 1)
<< pentru deplasare stânga
>> pentru deplasare dreapta
Cu excepţia operatorului negare care este unar, ceilalţi sunt operatori binari.

Operaţiile logice pentru o pereche oarecare de biţi x şi y se prezintă astfel:

x	y	x&y	x | y	x^ y	~ x
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Exemple: Numerele x=5, y=36 se reprezintă în binar astfel:
x=00000101, y=00100100

```
        00000101                  00000101
        00100100                  00100100
x&y= ------------      ,   x | y = ------------
        00000100                  00100101

        00000101
        00100100                   00000101
 x^y= ------------      ,    ~ x= ------------
        00100001                   11111010
```

Operatorul & se poate utiliza la anulări de biţi, de exemplu:
`a & 0x00ff` are ca valoare, valoarea octetului mai puţin semnificativ al valorii variabilei a (primii 8 biţi sunt înlocuiţi cu 0, iar următorii 8 biţi coincid cu cei ai lui a).

Operatorul | se poate utiliza la la setări de biţi, de exemplu:
`a | 0x00ff` primii 8 biţi ai rezultatului coincid cu cei ai lui a, iar următorii 8 sunt 1.

Operatorul <<, operator binar, realizează *deplasarea la stânga* a valorii primului operand cu un număr de poziţii binare egal cu valoarea celui de-al doilea operand al său, biţii liberi din dreapta se completează cu zero. Această operaţie este echivalentă cu o înmulţire cu puteri ale lui 2.

Exemplu.
x=7 în baza 2 se scrie 0000 0000 0000 0111
x <<2 va produce 0000 0000 0001 1100 deplasarea cu o poziţie spre stânga şi adăugarea unui zero. Valoarea sa este 28 adica $7*2^2$

Operatorul >>, operator binar, realizează *deplasarea la dreapta* a valorii primului operand cu un număr de poziţii binare egal cu valoarea celui de-al doilea operand al său. Această operaţie este echivalentă cu o împărţire cu puteri ale lui 2.
In cazul deplasarii spre dreapta, biţii liberi din stânga se completează automat cu zero numai dacă numărul este nenegativ. Dacă numărul este negativ, din necesitatea de a conserva semnul (reprezentat în bitul cel mai semnificativ cu 1), biţii liberi din stînga se completează cu 1.

Exemple:
19>>3 are ca rezultat numărul binar 0000 0000 0000 0010 egal cu 2 adică $19/2^3$.
x=-9 se reprezintă în binar în cod complementar faţă de 2. Acesta se obţine adunând la complementul faţă de unu al numărului, valoarea 1.
Deci – 9 se obţine ca ~ 9+1.
9 se reprezintă prin: `0000 0000 0000 1001`

~ 9 se repretintă prin: `1111 1111 1111 0110`
~ 9 +1 se reprezintă prin: `1111 1111 1111 0111`
atunci – 9 se reprezintă prin: `1111 1111 1111 0111`
x>>2 va produce: `1111 1111 1111 1101`

Operatori de atribuire

Operatorul = se utilizează în construcţii de forma `v = expresie`.
Această construcţie se numeşte expresie de atribuire şi este un caz particular de expresie. Tipul ei coincide cu tipul lui v, iar valoarea ei este chiar valoarea atribuită lui v. Rezultă că o expresie de forma:

```
v1 = (v = expresie)
```

este legală.
Deoarece operatorii de atribuire se evaluează de la dreapta la stânga expresia de mai sus se poate scrie fără paranteze.
În general, putem realiza atribuiri multiple de forma

```
vn=vn-1=...=v1=expresie.
```

În cazul unei expresii de atribuire, dacă expresia din dreapta semnului egal are un tip diferit de cel al variabilei v, atunci întâi se converteşte valoarea ei spre tipul variabilei v şi pe urmă se realizează atribuirea.
Pentru operaţia de atribuire putem folosi operatorii de atribuire combinată:

```
op=
```

Unde prin op se înţelege unul din operatorii binari aritmetici sau logici pe biţi, adică: *, /, %, +, -, &, |, ^, <<, >>.
Expresia:

```
v op= expresie
```

este echivalentă cu

```
v = v op (expresie)
```

Se pot realiza maximum zece combinaţii:

```
+= ,-=, *=, /=, %=, & =, |=, ^=, <<=, >>=
```

Exemple:
Expresia: `x=x+5` este echivalentă cu `x+=5`.
Expresia `x=x^y` este echivalentă cu `x^=y.`
Expresia `x=x<<y` este echivalentă cu `x<<=y`.
Expresia: `x/=y+3` este echivalentă cu `x=x/(y+3)`.
Expresia: `tab[i*j+1]=tab[i*j+1]*z` este echivalentă cu `tab[i*j+1]*=z`.
Expresia `C=C*n/k,` cu `C, n` şi `k` de tip întreg, nu produce acelaşi rezultat cu `C*=n/k,` de exemplu pentru C=20, n=5, k=2 prima expresie furnizează rezultatul 50, iar a doua 40 (`C*=n/k` este echivalentă cu `C=C*(n/k)`).

Operatorul de conversie explicită

Dacă dorim să forţăm tipul unui operand sau al unei expresii putem folosi o construcţie de forma `(tip)operand`. Prin aceasta, valoarea operandului se converteşte spre tipul indicat în paranteze.
În C++, dacă tip este format dintr-un singur cuvânt, se poate folosi şi construcţia: `tip(operand)`

Exemple:

```
int x,y;
double z;
x=10; y=4;
 z=x/y;   /* z primeşte valoarea 2.0 deoarece împărţirea, făcându-se între operanzi de tip întreg,
      este o împărţire întreagă*/
z=(double)x/(double)y;  // rezultatul va fi z=2.5.
```

Construcţia: `(float)(x+y)` este echivalentă în C++ cu `float(x+y).`
Construcţia: `unsigned char(x)` este eronată deoarece tipul conversiei este format din două cuvinte.

Operatori condiţionali

Operatorii condiţionali se utilizează în evaluări de expresii care prezintă alternative. Ei sunt „`?:`”.
O astfel de expresie are formatul:

```
exp1 ? exp2 : exp3
```

şi are următoarea interpretare: dacă `exp1` este diferită de zero, atunci valoarea şi tipul expresiei condiţionale sunt date de valoarea şi tipul expresiei `exp2` altfel de valoarea şi tipul lui `exp3`.

Exemplu:

```
y?x/y:x*x
```

Maximul dintre două numere se poate determina astfel:

```
max=a>b?a:b;
```

Operatorul virgulă

Există cazuri în care este util să grupăm mai multe expresii într-una singură, expresii care să se evalueze succesiv. În acest scop se foloseşte operatorul virgulă care separă secvenţa de expresii, acestea grupându-se într-o singură expresie.
Expresia:

$exp_1, exp_2, ... , exp_n$

va avea ca valoare şi tip valoarea şi tipul lui exp_n, deci cu a ultimei expresii.

Exemple:

```
k=(x=10, y=2*i-5, z=3*j, x+y+z);
```

Se execută pe rând cele trei atribuiri, apoi se efectuează suma x+y+z care se atribuie lui k.

```
++i, --j
```

i se măreşte cu o unitate, j se micşorează cu o unitate, valoarea şi tipul întregii expresii coincid cu valoarea şi tipul lui j.

Operatorul dimensiune (sizeof)

Lungimea în octeţi a unei date se poate afla cu o construcţie de forma

```
 sizeof(data)
```

unde data este numele unei variabile, al unui tablou, al unui tip etc.

Exemple:

```
int i;
float x;
char c;
double d;
int tab[10];

sizeof(i)             -furnizează valoarea 2
sizeof(x)             -furnizează valoarea 4
sizeof(float)         -furnizează valoarea 4
sizeof(c)             -furnizează valoarea 1
sizeof(tab)           -furnizează valoarea 20
sizeof(tab[i])        -furnizează valoarea 2
```

Operatori paranteză

Operatorul paranteză rotundă „( )” se utilizează pentru a impune o altă ordine în efectuarea operaţiilor. O expresie inclusă între paranteze rotunde formează un operand.
Parantezele rotunde se utilizează şi la apelul funcţiilor.
Parantezele pătrate „[ ]” includ expresii care reprezintă indici. Ele formează operatorul de indexare.

Operatori de adresă

Operatorul & returnează adresa unei variabile. Astfel, dacă x este o variabilă, &x va fi adresa variabilei x.
Operatorul * returnează valoarea de la o anumită adresă. Astfel, dacă p este pointer la tipul char, *p va fi caracterul referit de p.

Regula conversiilor implicite

Dacă operanzii unui operator binar nu sunt de acelaşi tip, sunt necesare conversii care se execută automat astfel:

1. Fiecare operand de tip char, unsigned char sau short se converteşte spre tipul int şi orice operand de tip float se converteşte spre tipul double.
2. Dacă unul dintre operanzi este de tip long double, atunci şi celălalt se converteşte la tipul long double şi rezultatul va fi de tip long double.
3. Dacă unul dintre operanzi este de tip double, atunci şi celălalt se converteşte la tipul double şi rezultatul va fi de tip double.
4. Dacă unul dintre operanzi este de tip unsigned long, atunci şi celălalt se converteşte la tipul unsigned long şi rezultatul va fi de tip unsigned long.
5. Dacă unul dintre operanzi este de tip long, atunci şi celălalt se converteşte la tipul long şi rezultatul va fi de tip long.
6. Dacă unul dintre operanzi este de tip unsigned, atunci şi celălalt se converteşte la tipul unsigned şi rezultatul va fi de tip unsigned.

3.8. Operaţii de intrare-ieşire

Operaţiile de intrare/ieşire asigură instrumentul necesar pentru comunicarea dintre utilizator şi calculator. LIMBAJUL C/C++ asigură posibilitatea de a folosi orice echipamente periferice: consola (tastatura şi ecranul), imprimanta, unităţi de magnetice diverse etc.

Pentru uniformizarea modului de lucru cu dispozitivele de intrare/ieşire, se introduce un nivel intermediar între program şi echipamentul periferic folosit. Forma intermediară a informaţiilor corespunde unui echipament "logic" şi se numeşte "flux" sau "şir" de informaţii (stream, în limba engleză) şi nu depinde de echipamentul periferic folosit. Un flux de informaţii constă dintr-o succesiune ordonată de octeţi şi poate să fie privit ca un tablou unidimensional de caractere de lungime neprecizată. Citirea sau scrierea la un echipament periferic constă în citirea datelor dintr-un flux, sau scrierea datelor într-un flux.

Operaţii de intrare / ieşire, utilizând consola, în C++

Limbajul C++ nu dispune de instrucţiuni specifice pentru operaţiile de intrare/ieşire. Acestea se efectuează cu ajutorul unor funcţii de bibliotecă ce folosesc conceptele generale ale limbajului C++: programare orientată pe obiecte, cu ierarhii de clase, moşteniri multiple, supraîncărcarea operatorilor.
Fişierul `iostream.h`, care trebuie inclus în orice program pentru a apela operaţiile de I/E specifice C++, conţine cele mai importante funcţii de lucru cu tastatura şi ecranul.
De asemenea în acest fişier sunt definite fluxurile:
`cin` folosit pentru intrare, dispoziţiv implicit tastatura (*console input*);
`cout` folosit pentru ieşire, dispoziţiv implicit ecranul (*console output*);
`cerr` folosit pentru afişarea erorilor, dispoziţiv implicit ecranul;
`clog` folosit pentru afişarea erorilor, dispozitiv implicit ecranul; `clog` reprezintă versiunea cu tampon a lui `cerr`.
Deoarece conceptele programării orientate pe obiecte nu au fost prezentate, vom explica numai modul de utilizare a funcţiilor de I/E.
Vom accepta că în fişierul `iostream.h` sunt definite noi tipuri de date printre care `istream` şi `ostream`. Obiectele de tipul `istream` sunt dispozitive logice de intrare (`cin` este un astfel de obiect).
Obiectele de tipul `ostream` reprezintă dispozitive logice de ieşire (`cout, cerr, clog` sunt astfel de obiecte).
Se prevăd două nivele de interfaţă între programator şi dipozitivele logice de intrare/ieşire, prin două seturi de funcţii:
a) funcţii pentru operaţii de I/E la nivel înalt
b) funcţii pentru operaţii la nivel de caracter.

Funcţii pentru operaţii de I/E la nivel înalt

Pentru operaţiile de I/E la nivel înalt, au fost supraîncărcaţi operatorii "`>>`" pentru fluxul *cin* şi "`<<`" pentru fluxul *cout, cerr, clog*.
Operatorul "`>>`", redefinit, se numeşte *operator de extracţie* sau *extractor*. Această denumire provine de la faptul că la citire se extrag date dintr-un stream.
Operatorul "`<<`", redefinit, se numeşte *operator de inserţie* sau *insertor*. Această denumire provine de la faptul că un operator de ieşire inserează informaţia în stream.
Ambii operatori posedă proprietatea de asociativitate la stânga şi returnează o referinţă la streamul asociat ca prim operand (cin pentru ">>", respectiv cout, cerr sau clog pentru "<<"), deci ei pot fi înlănţuiţi.

Exemplu:

```
    float v,t,d;
    cout<<”Distanta parcursa(Km)=”;  cin>>d;
    cout<<”Numar de ore =”;          cin>>t;
    v=d/t;
    cout<<”Viteza medie este de ”<<v<<” km/h”;
       // înlănţuirea operatorului <<
```

Nu se acceptă înlănţuiri mixte ("<<" cu ">>").

Faptul ca operatorii returnează o referinţă către stream, permite să se poată testa starea streamului ca în exemplul următor:

```
    int n;
    if(cin>>n) cout<<"citire cu succes";
    else  cout<<"esec la citire";  // s-a tastat un şir nenumeric
```

Tipurile de date predefinite acceptate implicit de un stream de intrare sunt: char, short, int, long toate cu sau fără unsigned, float double şi şir de caractere.
A extrage o valoare numerică dintr-un şir înseamnă:
• ignorarea tuturor caracterelor de tip spaţiu;
• un test dacă primul caracter nespaţiu este cifră, semn sau punct zecimal (pentru tipurile flotante). În caz că această condiţie este îndeplinită se va continua cu următorul pas, altfel se va semnaliza eroare şi nu se vor mai putea efectua operaţii de intrare până ce nu se tratează respectiva eroare;
• se extrag toate caracterele consecutive, până la întâlnirea unuia invalid pentru respectivul tip de dată;
• în cele din urmă, se va efectua conversia şirului extras la tipul de dată respectiv.
În cazul tipului de dată char, o secvenţă de genul:

```
char c;
cin>>c;
```

atribuie variabilei c, primul caracter nespaţiu aflat în streamul de intrare.
Dacă se doreşte extragerea unui şir de caractere, se vor citi toate caracterele întâlnite consecutiv în streamul de intrare, începând cu primul caracter nespaţiu din stream, până se va ajunge la un caracter spaţiu. Stringului astfel obţinut i se va adăuga la sfârşit caracterul NULL (\0).
Să presupunem că, la o operaţie de citire, primul caracter nespaţiu din streamul de intrare va fi invalid pentru tipul de dată citit. În acest caz valoarea variabilei destinaţie va rămâne neschimbată. În plus se va seta un indicator, indicatorul "fail" (eşec) al streamului de intrare, fapt ce va duce la imposibilitatea de a mai citi din acest stream până la resetarea indicatorului.
Indicatorul de eroare se anulează cu ajutorul metodei (funcţiei) `clear().`
Sintaxa de apel pentru streamul `cin` este:

```
cin.clear();
```

Metoda fail(), returnează o valoare non-zero (true) dacă streamul de citire se află în “stare de eroare” şi 0 dacă citirea s-a efectuat cu succes.
Sintaxa de apel pentru streamul `cin` este:

```
cin.fail()
```

Metoda `eof()`, returnează o valoare 1 (true) dacă s-a atins sfârşitul streamului din care citim şi 0 în caz contrar.

Sintaxa de apel pentru streamul `cin` este:

```
cin.eof()
```

Tipurile de date predefinite pe care le acceptă implicit un stream de ieşire sunt: `char, short, int, long` cu sau fără `unsigned, float, double, long double`, pointeri la aceste tipuri şi tipul `void*`.

Pointerul la caracter `(char*)` este utilizat la tipărirea şirurilor de caractere iar `void*` la cea a variabilelor de tip pointer (afişarea se face în hexa).

Exemple:

1.

```
char c;
long n;
float x;
double t;
cin>>c; cout<<"c="<<c<<'\n'; // cout<<'\n' are ca efect saltul la linie nouă.
cin>>n; cout<<"n="<<n<<'\n';
cin>>x; cout<<"x="<<x<<'\n';
cin>>t; cout<<"t="<<t<<'\n';
```

2.

```
char s[20];
cin>>s; cout<<s;
```

Deoarece numele unui tablou are ca valoare adresa primului său element, înseamnă că, în exemplul de mai sus, s este de fapt un pointer (constant) către caractere.

3.

```
char* a="xyzw";
cout<<"sirul a="<<a<<" adresa sirului a="<<(void*)a;
```

Manipulatori de intrare / ieşire

Pentru situaţia în care nu dorim ca citirea dintr-un stream sau scrierea într-un stream să se facă în formatele implicite din C++, sunt prevăzute şi posibilităţi de a controla formatele de intrare şi de ieşire prin comenzi incluse în construcţiile de intrare/ieşire.
Aceste comenzi se numesc manipulatori de I/E.
Pentru a putea folosi manipulatorii care au parametri (de exemplu setw()) trebuie să includem fişierul iomanip.h. Acest lucru nu este necesar dacă utilizăm manipulatori fără parametri. Manipulatorii pot apare în lanţul de operaţii de I/E.

Lista manipulatorilor de I/E este următoarea:

Manipulator	Scop	Intrare/ ieşire
dec	formatează datele numerice în zecimal valabil până la resetare	E
oct	formatează datele numerice în octal, valabil până la resetare	E
hex	formatează datele numerice în hexazecimal, valabil până la resetare	E
setbase(int baza)	stabileşte baza de numeraţie la *baza* (8,10,16), valabil până la resetare	E
setw(int w)	stabileşte la w numărul de poziţii pe care se va afişa următoarea dată de scris.	E
setprecision(int p)	stabileşte numărul de cifre aflat după punctul zecimal. comanda este valabilă	E

Manipulator	Scop	Intrare/ ieşire
	până la reapelare.	
setfill(int ch)	stabileşte caracterul de umplere, implicit spaţiu, valabil până la resetare	E
setiosflags(long f)	activează indicatorii de format specificaţi în variabila f.	I/E
resetiosflags(long f)	dezactivează indicatorii de format specificaţi în variabila f.	I/E
flush	eliberează un stream	E
endl	scrie un caracter "newline" şi eliberează streamul	E
ends	Scrie un caracter null	E
ws	ignoră caracterele de tip spaţiu	I

Exemplu:

```
#include<iostream>
#include<iomanip>
int main()
  { cout<<hex<<100<<endl;
    cout<<dec <<setfill('*')<<setw(5)<<100<<endl;
    return 0;
  }
```

Indicatori de format

Fiecare stream din C++ are asociat un număr de indicatori de format flags. Ei sunt codificaţi într-o variabilă de tip *long*.
Următoarele constante definesc aceşti indicatori de format:

Constanta	Valoare	Rol
ios::skipws	0x0001	Ignoră caracterele de tip spaţiu de la intrare
ios::left	0x0002	aliniere la stânga în ieşire
ios::right	0x0004	aliniere la dreapta în ieşire
ios::internal	0x0008	semn aliniat ex: –55 —> – 55
ios::dec	0x0010	conversie în baza 10
ios::oct	0x0020	conversie în baza 8
ios::hex	0x0040	conversie în hexazecimal
ios::showbase	0x0080	se va tipări şi baza (0 - octal, 0x -hexa)
ios::showpoint	0x0100	afişează şi zerourile nesemnificative de după punctul zecimal
ios::uppercase	0x0200	Pentru "literele" din baza 16 se vor utiliza majuscule
ios::showpos	0x0400	întregii pozitivi sunt prefixati de "+"
ios::scientific	0x0800	Pentru date flotante se foloseşte notaţia stiinţifică (1.234e2)
ios::fixed	0x1000	utilizează notaţia normală (123.45)
ios::unitbuf	0x2000	streamul se goleşte (afişează) după fiecare inserare
ios::stdio	0x4000	Stream-urile predefinite de ieşire se vor goli după fiecare inserare

Prin aplicarea operatorului de tip or "|" putem poziţiona mai mulţi indicatori.

De exemplu:
```
cout<<setiosflags(ios::left|ios::showpoint);
```
realizează alinierea la stânga şi afişarea zerourilor nesemnificative de după punctul zecimal.

Pentru stream-urile standard, avem următoarele valori implicite:
pentru `cin: 0x0001` pentru `cout: 0x2001`
pentru `cerr: 0x2001` pentru `clog: 0x0001`
Pentru a afla stările curente ale indicatorilor, se foloseşte funcţia `flags()`.
Funcţia se apelează utilizând formatul: `<stream>.flags()` şi returnează o valoare de tip `long` ce conţine, codificat, indicatorii de stare ai streamului asociat.

Exemplu: Indicatorii streamului cin se pot afişa astfel:
```
cout<<hex<<cin.flags();
```

3.9 Instrucţiuni C/C++

Instrucţiunea de atribuire

Se obţine scriind punct şi virgulă după o expresie de atribuire sau după o expresie în care se aplică la o variabilă unul din operatorii de incrementare sau decrementare. Deci o instrucţiune de atribuire are una din următoarele formate:

```
<expresie de atribuire>;
<variabila>++;
++<variabila>;
<variabila>--;
--<variabila>;
```

Instrucţiunea compusă (blocul)

Instrucţiunea compusă este o succesiune de instrucţiuni incluse între acolade, succesiune care eventual poate conţine declaraţii. Sintaxa:

```
{
  <declaraţii şi instrucţiuni>
}
```

Dacă sunt prezente, declaraţiile definesc variabile care sunt valabile numai în instrucţiunea compusă respectivă. După paranteza închisă a unei instrucţiuni compuse nu se pune ";"

Structura unei funcţii poate fi considerată ca fiind:

```
<antetul funcţiei>
   <instrucţiune compusă>
```

Instrucţiunea *if*

Instrucţiunea *if* implementează structura alternativă.
Ea are unul din formatele:

Formatul 1:

```
if(<expresie>)
   <instrucţiune1>
else
   <instrucţiune2>
```

Dacă <*expresie*> este diferită de zero se execută <*instrucţiune1*>, altfel se execută <*instrucţiune2*>.

Formatul 2:

```
if(<expresie>) <instrucţiune>
```

Dacă <*expresie*> este diferită de zero se execută <*instrucţiune*>, altfel instrucţiunea *if* nu are niciun efect.

O selecţie multiplă se poate programa cu mai multe instrucţiuni *if – else* în cascadă.
Sintaxa:

```
if ( <expresie1> ) <instrucţiune1> ;
else if ( <expresie2> ) <instrucţiune2> ;
       . . .
     else if ( <expresien> ) <instrucţiunen> ;
          else <instrucţiunen+1> ;
```

Trebuie ţinut seama de faptul că *else* este asociat celui mai apropiat *if*.
Pentru a determina un alt mod de asociere se pot utiliza delimitatorii de bloc.

```
if ( <expresie1> )
  { if ( <expresie2 > ) <instrucţiune1 >  }
else <instrucţiune2 >
```

În acest mod *else* se asociază primului *if* şi nu celui de–al doilea care este mai apropiat.

Exemplu de utilizare a instrucţiunii *if*:

Programul următor calculează valoarea funcţiei

$$f(x)=\begin{cases}4x^2+2x-1, & \text{dacă } x<0\\ 50, & \text{dacă } x=0\\ 2x^2+8x+1, & \text{dacă } x>0\end{cases}$$

într-un punct x introdus de la tastatură.

```
#include<iostream>
#include<conio.h>
using namespace std;
int main()
```

```
  { float x,f;
    cout<<"x="; cin>>x;  // Citirea lui x

    // Evaluarea functiei:
    if(x<0) f=4*x*x+2*x-1;
    else if(x==0)  f=50;
         else      f=2*x*x+8*x+1;
    cout<<"f("<<x<<")="<<f<<endl;      // Afisarea rezultatului
    return 0;
}
```

Instrucţiunea *while*

Instrucţiunea `while` implementează structura repetitivă cu test iniţial şi are sintaxa:

```
while(<expresie>)
    <instrucţiune>
```

Se execută <instrucţiune> cât timp <*expresie*> este adevărată.

Exemple de utilizare a instrucţiunii *while*:

1) Fie x un vector cu n elemente reale ce se introduc de la tastatură. Următorul program inversează ordinea elementelor în vector.

```
#include<iostream>
using namespace std;

int main()
 {float x[100],aux;
  int n,i,j;

  // Citirea vectorului:
  cout<<"n=";  cin>>n;
  i=0;
  while(i<n)
    { cout<<"x"<<i<<"="; cin>>x[i];
      i++;
    }

  // Inversarea elementelor:
  i=0; j=n-1;
  while(i<j)
    {aux=x[i];  x[i]=x[j];  x[j]=aux;
     i++;   j--;
    }

  // Afisarea vectorului:
  cout<<" vectorul inversat este:";
  i=0;
  while(i<n)
    { cout<<x[i]<<" ";
      i++;
    }
  return 0;
 }
```

2) Să se calculeze valoarea numărului π *utilizând formula lui Madhava din Sangamagrama (anul ≈1400)* $\pi = \sqrt{12}\sum_{k=0}^{\infty}\frac{\left(-\frac{1}{3}\right)^k}{2k+1}$

```
#include<iostream>
#include<iomanip>
#include <math.h>
using namespace std;

long double Pi()
 { long double eps=1.0e-20;
   long double pi=1.0, pia=0,t=1;
      /*  pi=valoarea la pasul curent, k;
          pia=valoarea de la pasul anterior;
          t=(-1/3)^k ;
       */
   unsigned long k=1;
   while(fabsl(pi-pia)>eps)
     { pia=pi;
       t*=-1.0/3.0;
       pi+=t/(2*k+1);
       k++;
     }
   // cout<<k<<endl;
   return sqrt(12)*pi;
 }

 int main()
  {
    cout<<endl<<setw(22)<<setprecision(20)<<Pi()<<endl;
    return 0;
  }
```

Instrucţiunea *for*

Instrucţiunea *for*, ca şi instrucţiunea *while* se utilizează pentru implementarea structurii repetitive cu test iniţial. Uzual, instrucţiunea *for* se foloseşte pentru implementarea ciclului cu contor.
Formatul ei este:

```
for( <exp1>; <exp2>; <exp3> )
   <instrucţiune>
```

unde:

<*exp₁*> se numeşte expresie de iniţializare şi se evaluează o singură dată, înaintea primei iteraţii, realizând iniţializarea ciclului. Uzual, ea atribuie o valoare iniţială variabilei de control a ciclului. Valoarea sau expresia de iniţializare poate lipsi in situaţia în care iniţializarea ciclului este făcută în afara sa, sau poate conţine mai multe expresii de iniţializare separate prin operatorul virgulă.

<*exp₂*> se numeşte expresie de testare şi se execută înaintea fiecărei iteraţii, reprezentând condiţia de continuare a ciclului. Ciclul se termină când această expresie

devine falsă. Dacă <*exp₂*> lipseşte, se consideră că expresia de test este adevărată tot timpul, iar ciclul se execută fără întrerupere.

<*exp₃*> specifică reiniţializările ce se efectuează după fiecare iteraţie; <instrucţiune> formează corpul ciclului, care se execută repetat.

Expresiile <exp1>,<exp2>,<exp3> pot fi şi vide. Totuşi caracterele ";" vor fi totdeauna prezente.

Instrucţiunea *for* este echivalentă cu:

```
<exp₁>;
while(<exp₂>)
  { <instrucţiune>
    <exp₃>;
  }
```

Reciproc, orice instrucţiune while:

```
while(<exp>) <instrucţiune>
```

este echivalentă cu

```
for( ; <exp> ; ) <instrucţiune>
```

Instrucţiunea:

```
for(;;) <instrucţiune>
```

defineşte un ciclu "infinit" din care se iese prin alte mijloace decât cele obişnuite.

Exemple de utilizare a instrucţiunii *for*:

Programul următor calculează suma $S=\sum_{k=0}^{n}\frac{(-a)^k}{k!}x^k$

```
#include<iostream>
using namespace std;

int main()
{ long double T,S;
  float a,x;
  int n,k;
  cout<<"n="; cin>>n;
  cout<<"a="; cin>>a;
  cout<<"x="; cin>>x;
  T=1;
  S=1;
  for(k=1;k<=n;k++)
   { T*=-a*x/k;
     S+=T;
   }
  cout<<"S="<<S<<endl;
  return 0;
}
```

2) Fie x_1, x_2,…, x_n ,numere întregi ce se introduc de la tastatură. Să se determine suma numerelor pozitive şi suma pătratelor numerelor negative.

```
#include<iostream>
using namespace std;

int main()
{ int n,x,S1,S2,i;
  S1=0;
  S2=0;
  cout<<"n="; cin>>n;
  for(i=1;i<=n;i++)
    { cout<<"x"<<i<<"="; cin>>x;
      if(x>=0)S1+=x;
      else S2+=x*x;
    }
  cout<<"Suma numerelor pozitive="<<S1<<endl;
  cout<<"Suma patratelor numerelor negative="<<S2<<endl;
  return 0;
}
```

3) Fie a_0, a_1,…, a_{n-1} şi x_0, x_1,…, x_{n-1}, numere reale ce se introduc de la tastatură în vectorii a şi x. Programul următor calculează suma $S = \sum_{i=0}^{n-1} a_i \cdot x_i$

```
#include<iostream>
using namespace std;

int main()
 {float a[100],x[100],S;
  int n,i;
  cout<<"n=";  cin>>n;
  for(i=0;i<n;i++)  { cout<<"a"<<i<<"="; cin>>a[i]; }
  for(i=0;i<n;i++)  { cout<<"x"<<i<<"="; cin>>x[i]; }

  S=0;
  for(i=0;i<n;i++)
       S=S+a[i]*x[i];
  cout<<"S="<<S;
  return 0;
 }
```

Putem avea mai multe cicluri *for* consecutive, ca de exemplu:

```
p=1;
for(i=1; i<=m; i++)
  for(j=1; j<=n; j++)
    for(k=1; k<=p; k++)
      p*=i+j+k;
```

Instrucţiunea *do while*

Această instrucţiune are formatul:

```
do
  <instrucţiune>
while(<expresie>);
```

şi implementează structura repetitivă cu test final.

Efect: Execută în mod repetat <*instrucţiune*> (simplă sau compusă) cât timp <*expresie*> este adevărată (diferită de zero).

Exemplu de utilizare a instrucţiunii *do while*:

Programul următor calculează suma unor produse de perechi de numere introduse de la tastatură cât timp suma rezultată este mai mica decât 1000.

```
#include<iostream>
using namespace std;

int main()
 {long x,y,S;
  S=0;
  do
   { cout<<"x,y=";
     cin>>x>>y;      //se vor  tasta două numere întregi separate prin spaţiu
     S+=x*y;
   }
  while (S<1000);
  cout<<"S="<<S<<endl;
  return 0;
 }
```

Instrucţiunea *break*

Formatul acestei instrucţiuni este:

```
break;
```

Ea produce ieşirea forţată din instrucţiunile repetitive `while, do while` şi `for` sau dintr-o instrucţiune `switch`. Instrucţiunea `break` permite ieşirea dintr-un singur ciclu, nu şi din eventualele cicluri care ar conţine ciclul în care s-a executat instrucţiunea `break`.
Un exemplu de utilizare frecventă, il constituie ieşirea dintr-un ciclu infinit de forma:

```
for(;;)
 { ...
   if(...) break;
   ...
 }
```

Instrucţiunea `break` provoacă eroare dacă apare în afara instrucţiunilor `while, for, do while şi switch`.

Exemple de utilizare a instrucţiunii *break:*

1) Următorul program implementează jocul „Ghiceşte numărul!”.

```
#include<iostream>
#include<conio.h>
using namespace std;

int main()
```

```
{ int a=576, b;
  cout<<"Ghiciti un numar de la 1 la 1000: ";
  cin>>b;
  while(1) // ciclu infinit
  { if(b==a)
    {  cout<<"Ai ghicit nr!";
       break;
    }
    if(b>a)  cout<<"Numar prea mare"<<endl;
    else     cout<<"Numar prea mic"<<endl;
    cout<<"Incercati alt numar:";  cin>>b;
  }
  return 0;
}
```

2) Următorul program afişează poziţia pe care se află un număr x în vectorul neordonat (a_0, a_1,..., a_{n-1}).

```
#include<iostream>
using namespace std;

int main()
 {long a[100],x,poz;
  int n,i;
   // Citirea vectorului:
  cout<<"n="; cin>>n;
  cout<<“Tastati elementele vectorului:”;
  for(i=0;i<n;i++) cin>>a[i];
   // Citirea lui x:
  cout<<"x="; cin>>x;
   // Determinarea pozitiei lui x in vector::
  poz=-1;
  for(i=0;i<n;i++)
    if(x==a[i]){poz=i; break;}
  if(poz>=0)
        cout<<x<<"se afla in vector pe pozitia "<<poz<<endl;
  else  cout<<x<<"nu se afla in vector "<<endl;
  return 0;
 }
```

3) Următorul program verifică dacă un vector format din n elemente numere reale, este ordonat crescător.

In program se foloseşte variabila semafor pentru a indica dacă vectorul este ordonat crescător (semafor==1) sau nu (semafor==0).

```
#include<iostream>
using namespace std;


int main()
 {float a[100];
  int n,i;
  cout<<"n=";  cin>>n;
  cout<<"Tastati "<<n<<" numere separate prin spatiu:"<<endl;
  for(i=0;i<n;i++)cin>>a[i]; //citirea elementelor vectorului
```

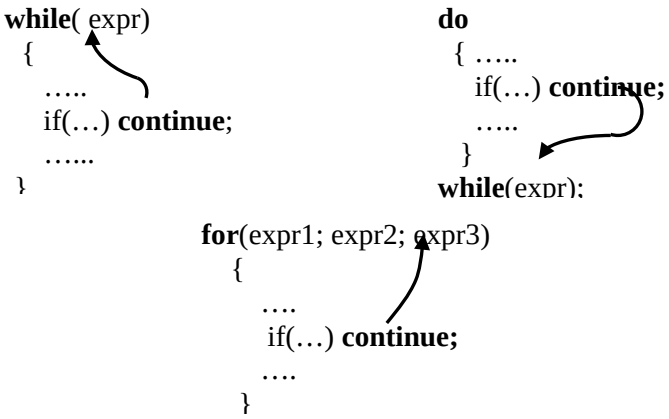
```
  int semafor;
  semafor=1;
  for(i=1;i<n;i++)
     if(a[i]<a[i-1])
     {  semafor=0;
        break; //vectorul nu este ordonat crescator
     }
  if(semafor==1)
     cout<<"vectorul este ordonat crescator"<<endl;
  else
     cout<<"vectorul nu este ordonat crescator"<<endl;
  return 0;
 }
```

Instrucţiunea *continue*

Are formatul:

```
continue;
```

Se utilizează în corpul unui ciclu şi are următorul efect:
• în ciclurile *while* şi *do while* ea realizează saltul la evaluarea expresiei care decide asupra continuării ciclului;
• în ciclul *for* ea realizează saltul la pasul de reiniţializare.



Astfel, programul

```
#include<iostream>
using namespace std;

int main()
{  int i;
   for (i=0;i<5;i++)
    {  if (i==3) continue;
       cout<<"i="<<i<<endl;
    }
   cout<<"valoarea lui i la iesirea din ciclul for este "<<i;
   return 0;
}
```

va afișa urmatoarele rezultate

i = 0

i = 1
i = 2
i = 4
valoarea lui i la iesirea din ciclul for este 5

Exemplu de utilizare a instrucţiunilor continue şi break:

Se introduc de la tastatură numere întregi (pozitive şi negative) cît timp suma numerelor pozitive este mai mică decât 1000. Programul afişează pătratul numerelor pozitive.

```
#include<iostream>
#include<conio.h>
using namespace std;

int main()
 { long S,x;
   S=0;
   for(;;)   // ciclu infinit
    { cout<<"x=";  cin>>x;
      if(x<=0) continue; /* salt la partea de reinitializare a instructiunii for
                            (in cazul de fata   expresia de  reinitializare este vida)*/
      S+=x;
      if(S>=1000) break;       // iesire din for
      cout<<"x*x="<<x*x<<endl;
    }
   return 0;
  }
```

Instrucţiunea *switch*

Implementează structura de selecţie multiplă.
Sintaxa acestei instrucţiuni este:

```
switch(<expresie>)
  {  case <c1>: <sir1>
     case <c2>: <sir2>
     ...
     case <cn>: <sirn>
     default  : <sir>
  }
```

unde
<*c1*>,<*c2*>,...,<*cn*> sunt expresii constante de tip întreg,
<*expresie*> este o expresie de tip întreg (orice tip întreg),
<*sir1*>,...,<*sirn*> sunt şiruri de instrucţiuni (un astfel de şir poate fi şi vid).

Efect:
1) Se evaluează <*expresie*>.
2) Se compară valoarea expresiei <*expresie*>, succesiv, cu valorile $<c_1>, <c_2>, \ldots, <c_n>$.
3) Dacă valoarea <*expresie*> coincide cu $<c_k>$, se execută secvenţa de instrucţiuni $<sir_k>, <sir_{k+1}>, \ldots, <sir_n>, <sir>$.
Dacă în această secvenţă se întâlneşte instrucţiunea `break`, atunci aceasta are ca efect ieşirea din instrucţiunea `switch`.

4) În cazul în care valoarea expresiei nu coincide cu niciuna din constantele `<c1>,<c2>,...,<cn>` se execută secvenţa de instrucţiuni definită de *<sir>*. Alternativa `default` nu este obligatorie, în lipsa ei, dacă valoarea *<expresie>* nu coincide cu niciuna din constantele `<c1>,<c2>,...,<cn>` instrucţiunea `switch` nu are niciun efect.

Exemplu de utilizare a instrucţiunii `switch`:

Programul următor calculează funcţia

$$f(x,k)=\begin{cases}\sin x, & k=1\\ \cos x, & k=2\\ \operatorname{tg} x, & k=3\\ \operatorname{ctg} x, & k=4\\ \sqrt{5+2\cdot\sin x+3\cdot\cos x}, & k\notin\{1,2,3,4\}\end{cases}$$

unde x este dat în grade.

```
#include<iostream>
#include<math.h>
#include<conio.h>
using namespace std;

int main()
 { int k;
   float x,f;
   float r; // pentru conversia lui x in radiani
   cout<<"k="; cin>>k;
   cout<<"x="; cin>>x;
   r=x*M_PI/180; //sau r=x*3.141592/180;
   switch(k)
     { case 1: f=sin(r);
               break;
       case 2: f=cos(r);
               break;
       case 3: f=tan(r);
               break;
       case 4: f=1/tan(r);
               break;
       default: f=sqrt(5+2*sin(r)+3*cos(r));
      }
    cout<<"f("<<k<<","<<x<<")="<<f<<endl;
    return 0;
  }
```

Instrucţiunea vidă

Pentru instrucţiunea vidă nu există cuvânt rezervat, prezenţa ei este marcată prin caracterul punct şi virgulă şi nu are niciun efect asupra variabilelor, starea acestora rămânând neschimbată; Instrucţiunea vidă este necesară în anumite situaţii de programare.

De exemplu, poate fi utilă pentru un ciclu fără instrucţiuni în corpul său:

```
i=0;
while(x[i++]=y[i]); /* copiaza elementele vectorului y in vectorul x pana la
                       intalnirea primului element zero din y */
```

Instrucţiunea *goto*

Prin etichetă înţelegem un nume urmat de două puncte (:)

<*nume*>:

Etichetele sunt locale funcţiei şi prefixează instrucţiuni. Instrucţiunea goto are formatul

```
goto <nume>;
```

Ea realizează saltul la instrucţiunea prefixată de <*nume*>:

Se recomandă folosirea instrucţiunii goto când dorim să ieşim dintr-un ciclu inclus în mai multe cicluri.

Apelul unei funcţii

O funcţie de forma:

```
void <nume funcţie>(<lista parametrilor formali>)
     { ... }
```

care nu produce o valoare directă, se apelează printr-o instrucţiune de apel cu următorul format:

```
<nume funcţie>(<lista parametrilor efectivi>);
```

O funcţie de forma

```
<tip returnat> <nume funcţie>(<lista parametrilor formali>)
     { ... }
```

unde `<tip returnat>` este diferit de `void` (prin urmare returnează valori directe), poate fi apelată fie printr-o instrucţiune de apel, când nu dorim să utilizăm valoarea returnată, fie sub forma unui operand al unei expresii când utilizăm valoarea returnată.

În exemplul următor funcţia `int getch(void)` (returnează codul ASCII al caracterului citit de la tastatură), este apelată în ambele variante.

```
#include<iostream>
#include<conio.h>
using namespace std;

int main()
 {char c;
  cout<<"tastati un caracter";
  c=getch();     //citeste caracterul tastat si il memoreaza in c  (folosim valoarea returnata)
  cout<<c<<endl;

  getch();       // citeste caracterul tastat fara memorare  (nu folosim valoarea returnata)
  return 0;
 }
```

Instrucţiunea return

Revenirea dintr-o funcţie se poate face în două moduri:
• la întâlnirea instrucţiunii `return`;
• după execuţia ultimei sale instrucţiuni, adică a instrucţiunii ce precede acolada închisă ce termină corpul funcţiei respective. În această situaţie funcţia nu returnează nicio valoare.
Instrucţiunea `return` are două formate:

```
return;
```

caz în care funcţia nu returnează un rezultat direct(tipul funcţiei este void), sau

```
return <expresie>
```

caz în care funcţia returnează valoarea expresiei *<expresie>* (convertită, dacă este cazul, la tipul funcţiei).

Exemplu:

În următorul program funcţia `int prim(long n)` *returnează 1 dacă argumentul este număr prim şi 0 în caz contrar.*

```
#include<iostream>
#include<math.h>
using namespace std;

int prim(long n) // verifica daca n ≥1 este prim
 { if(n==1) return 0; //1 nu este prim
   if(n==2||n==3) return 1; // 2 si 3 sunt nr. prime
   if(n%2==0) return 0; // n nu este prim pentru ca se divide cu 2
   long r=sqrt(n);
   for(long d=3; d<=r; d+=2)
     if(n%d==0) return 0; // n nu este prim pentru ca se divide cu d impar >=3
   return 1;
 }

int main()
  { long n;
    cout<<"n="; cin>>n;
    if(prim(n))cout<<n<<" este numar prim "<<endl;
    else       cout<<n<<" nu este numar prim "<<endl;
    return 0;
  }
```

3.10 Sfera de influenţă a variabilelor

În funcţie de locul de declarare, variabilele pot fi: *globale* când sunt declarate în afara funcţiilor şi *locale* când sunt declarate in interiorul funcţiilor.

Variabilele globale formează nivelul extern. Aceste variabile sunt accesibile din orice funcţie a fişierului sursă care urmează declaraţiei. Variabilele globale se alocă la compilare şi rămân în memoria calculatorului pe tot parcursul executării programului, de aceea se mai numesc şi permanente.
Variabilele locale formează nivelul intern si este format din declaraţiile conţinute în interiorul blocurilor formează (prin bloc înţelegem o instrucţiune compusă sau corpul unei funcţii, adică o succesiune de instrucţiuni delimitate de acolade). Aceste variabile pot fi folosite (sunt vizibile) doar în blocul în care au fost declarate sau într-un bloc

subordonat acestuia. La nivel inferior putem declara o variabilă având aceeaşi denumire cu a uneia declarate la nivel superior. In acest caz noua declaraţie va fi valabilă la acest nivel şi la nivelele inferioare (subordonate), iar în nivelele superioare rămâne valabilă declaraţia iniţială.

Variabilele locale pot fi declarate statice (prin utilizarea cuvântului cheie „*static*”), urmând să fie alocate la compilare şisă ramână în memoria calculatorului pe tot parcursul executării programului.
Variabilele locale nestatice sunt create şi li se alocă spaţiu în memoria calculatorului numai în momentul în care se execută blocul de program în care este declarată variabila. La încheierea execuţiei blocului respectiv, variabila dispare şi spaţiul de memorie va fi alocat altor blocuri. Dacă se revine ulterior în blocul iniţial, variabila va fi realocată şi poate să primească altă adresă. Variabilele de acest fel se numesc variabile cu alocare automată a adresei (variabile *automatice*). Alocarea variabilelor din clasa "*auto*" se face pe stiva sistemului.
Variabilele locale, al căror domeniu de valabilitate se limitează la un bloc sunt în mod implicit variabile automatice, chiar dacă nu se menţionează în mod explicit acest lucru. De asemenea, parametrii formali sunt variabile din clasa "auto" şi deci se alocă pe stiva sistemului. Dacă dorim ca o variabilă locală să nu fie alocată pe stivă, deci să nu fie "auto", o declarăm obişnuit, dar declaraţia va fi precedată de cuvântul "static":

```
static  <tip>  <lista de nume>;
```

Exemplu:

```
float f()
  { int k;
    static int a[5];
   // ...
  }
```

Variabila simplă `k`, precum şi variabila tablou `a` sunt cunoscute şi pot fi referite în interiorul blocului în care au fost declarate. Se mai spune că ele sunt locale. Variabila tablou `a`, descrisă prin cuvântul cheie static, îşi păstrează aceeaşi adresă pe toată durata programului, adresă pe care o primeşte la începutul executării programului. Variabilei `k` i se alocă spaţiu pe stivă de fiecare dată când se execută funcţia `f()`, la adrese care pot fi diferite.

Declaraţia informativă "extern"

Pentru ca o variabilă globală să poată fi folosită de funcţii situate în alt fişier sursă, sau în cadrul aceluiaşi fişier în funcţii anterioare declarării variabilei, trebuie ca acea variabilă să fie descrisă printr-o declaraţie "*extern*" în funcţiile respective sau în afara oricărei funcţii ale noului fişier. Acum fişierele pot fi compilate separat şi linkeditate împreună.

Exemplu:
Fişierul F1.CPP

```
...
int x,y;
...
```

Fişierul F2.CPP

```
...
```

```
extern int x,y;
int suma(){return x+y;}
```

Fişierele se pot compila separat.
Pentru obţinerea executabilului vom include în fişierul F2.CPP, fişierul F1.CPP cu ajutorul directivei #include"F1.CPP" sau vom crea un *proiect*. Pentru crearea proiectului vom selecta, din meniul turbo C, `Open project`. Alegem pentru proiect, de exemplu, numele PR1. Includem fişierele F1.CPP şi F2.CPP utilizând tasta funcţională <*Insert*>. Compilarea proiectului se face cu `Build all` din `Compile`. Ştergerea unui fişier din proiect se face prin poziţionarea pe fişierul respectiv cu ajutorul săgeţilor şi apoi acţionarea tastei `<Delete>`.

Exemplificare:

Program pentru ordonarea crescătoare a unui vector de numere întregi:

```
#include<iostream>
using namespace std;

long x[100];
int n;      // variabile globale

void citeste()    // funcţie pentru citirea vectorului
 { int i; //variabilă locală
   cout<<"n=";   cin>>n;
   cout<< "Tastati "<<n<<" numere intregi\n:";
   for(i=0;i<n;i++)
     { cout<<"x["<<i+1<<"]="; cin>>x[i];}
 }

void sorteaza()    // funcţie pentru ordonarea vectorului
 { int i,j; //variabile locale
   for(i=0;i<n-1;i++)
    for(j=i+1;j<n;j++)
     if(x[i]>x[j])
      { long aux; //variabilă locală
        aux=x[i]; x[i]=x[j]; x[j]=aux;
      }
 }

void afiseaza()   // funcţie pentru afişarea vectorului
  { int i;
    for(i=0;i<n;i++)cout<<x[i]<<" ";
  }

int main()      // funcţia principală
  {
     citeste();
     cout<<"X=";
     afiseaza();  cout<<endl;
     sorteaza();
     cout<<"Vectorul sortat: "<<endl;
     afiseaza(); cout<<endl;
     return 0;
   }
```

3.11 Iniţializarea variabilelor

Iniţializarea variabilelor simple

O variabilă simplă se poate iniţializa printr-o declaraţie de forma:

```
<tip> <nume> = <expresie>;
```

sau

```
static <tip> <nume> = <expresie>;
```

dacă variabila este statică.

În cazul variabilelor globale şi statice, expresia utilizată trebuie să fie o expresie constantă, care să poată fi evaluată de compilator la întâlnirea ei. Aceasta, deoarece variabilele globale şi statice se iniţializează prin valori definite la compilare. Variabilele automatice se iniţializează la execuţie, de fiecare dată când se activează funcţia în care sunt declarate. Din această cauză, expresia utilizată la iniţializare nu mai este necesar să fie o expresie constantă. Variabilele automatice care nu sunt iniţializate vor conţine valori întâmplătoare şi, deoarece ele apar şi dispar odată cu blocul în care au fost declarate, nu îşi păstrează valorile de la o execuţie la alta a blocului din care aparţin. Spre deosebire de variabilele automatice, variabilele globale şi statice primesc în mod automat valoarea zero la începutul executării programului, dacă nu sunt iniţializate în mod explicit. Variabilele statice sunt iniţializate numai o singură dată, la începutul executării programului, şi îşi păstrează valorile de la o execuţie la alta a blocului în care au fost declarate.

Exemplul 1:

```
#include<iostream>
using namespace std;

void incrementare()
  { int i=1;
    static int k=1;
    i++;
    k++;
    cout<<"i="<<i<<", k="<<k<<endl;
  }

int main()
  { incrementare();
    incrementare();
    incrementare();
    return 0;
  }
```

Rezultatul execuţiei va fi:

```
i=2, k=2
i=2, k=3
i=2, k=4
```

Variabilele interne statice oferă posibilitatea păstrării în permanenţă a unor informaţii ce aparţin funcţiei. O astfel de variabilă ar putea fi folosită pentru a memora de câte ori a fost apelată o funcţie.

Exemplul 2:

```
#include <iostream>
using namespace std;
```

```
void f1()
  { static int k;  // implicit iniţializată cu 0
    k++;
    cout<<"f1 apelul"<<k<<endl;
  }

void f2()
  { static int k;  /* Variabilă iniţializată implicit cu 0.   Este variabilă internă funcţiei f2(),
                      cu alocare statică la o adresă diferită de adresa variabilei k, internă funcţiei f1.*/
    k++;
    cout<<"f2 apelul"<<k<<endl;
  }

int main()
  { f1(); f1(); f2(); f1(); f2();
    return 0;
  }
```

Rezultatul execuţiei este:

```
f1 apelul 1
f1 apelul 2
f2 apelul 1
f1 apelul 3
f2 apelul 2
```

Iniţializarea variabilelor tablou

Un tablou unidimensional se poate iniţializa folosind formatul:

```
<tip> <nume> [<dim>]={<e1>,<e2>,...,<en>};
```

sau

```
static <tip> <nume> [<dim>]={<e1>,<e2>,...,<en>};
```

Numărul expresiilor <e_i> de iniţializare poate fi mai mic decât al numărului elementelor tabloului. Elementele neiniţializate au valoarea iniţială 0.
În cazul în care se iniţializează fiecare element al tabloului, numărul <*dim*> al elementelor acestuia nu mai este obligatoriu în declaraţia tabloului respectiv. Deci putem scrie:

```
<tip> <nume> []={<e1>,<e2>,...,<en>}
```

Numărul elementelor tabloului fiind considerat egal cu numărul expresiilor.

Pentru un tablou bidimensional vom folosi următorul format:

```
<tip><nume>[<dim1>][<dim2>]={{<e11>,<e12>,...,<e1 m1>},
                            {<e21>,<e21>,...,<e2 m2>},
                                      ...
                            {<en1>,<en1>,...,<en mn>}
                           };
```

sau

```
static <tip> <nume> [<dim1>][<dim2>]={{<e11>,<e12>,...,<e1 m1>},
                                      {<e21>,<e21>,...,<e2 m2>},
                                                ...
                                      {<en1>,<en1>,...,<en mn>}
                                     };
```

Numerele m_1, m_2,..., m_n, pot fi mai mici decât <*dim*$_2$> în oricare din acoladele corespunzătoare ale tabloului, de asemenea *n* poate fi mai mic decât <*dim*$_1$>. În aceste situaţii restul elementelor tabloului vor fi iniţializate cu 0. Dacă n este egal cu <*dim*$_1$>, atunci <*dim*$_1$> poate fi omis, dar <*dim*$_2$> este obligatoriu.

Într-o modalitate asemănătoare se pot iniţializa şi tablouri cu mai multe dimensiuni.

Exemplu

```
int  a[2][3]={ {1,2,3},
               {4,5,6}
             };
```

Un tablou multidimensional se poate iniţializa şi astfel:

```
int t[2][3]={1,2,3,4,5,6};
```

sau

```
int t[][3]={1,2,3,4,5,6};
```

Tablourile de tip caracter pot fi iniţializate astfel:

```
char <nume>[<dim>]=<sir de caractere>;
```

sau

```
static char <nume>[<dim>]=<sir de caractere>;
```

Compilatorul adaugă automat caracterul NULL (\0) după ultimul caracter al şirului utilizat în iniţializare. Numărul <*dim*> poate fi omis.

Deci declaraţia:

```
char t[4]={'a','b','c','\0'};
```

este echivalentă cu:

```
char t[4]="abc";
```

şi cu:

```
char t[]="abc";
```

3.12 Transferul parametrilor la apelul funcţiilor

La apelul unei funcţii, fiecărui parametru formal îi corespunde un parametru efectiv. In C++ sunt implementate două metode de transmitere a parametrilor la funcţii:

prin valoare – când o eventuală modificare a parametrului în funcţie nu afectează valoarea parametrului efectiv în funcţia apelantă. În cazul apelului prin valoare, se transferă funcţiei apelate valoarea parametrului efectiv care poate fi o constantă, o variabilă sau o expresie.

prin referinţă – în care variabila transmisă funcţiei ca parametru efectiv este afectată de eventualele modificări aduse în funcţie. Pentru aceasta funcţia apelată trebuie să dispună de adresa parametrului efectiv pentru ca să-l poată modifica.

Sunt trei modalităţi de a realiza transferul prin referinţă:

- prin utilizarea ca parametru a numelui unui tablou ;
- prin utilizarea ca parametru a unei variabile de tip pointer ;
- prin utilizarea ca parametru a unei variabile de tip referinţă.

Numele unui tablou are ca valoare chiar adresa primului său element, în consecinţă, dacă un parametru formal este numele unui tablou atunci la apel se va transmite funcţiei adresa de început a tabloului ce se utilizează efectiv, şi prin urmare îi pot fi modificate elementele.

Deoarece compilatorul nu foloseşte dimensiunea tabloului transmis ca parametru, ci doar adresa lui de început, pentru parametrul formal de tip tablou putem folosi sintaxa de declarare următoare:

```
<tip> <nume tablou>[]
```

3.13 Probleme rezolvate

1) Operaţii cu matrice

Citirea si afişarea matricelor, suma şi produsul a două matrice

```
#include<iostream>
#include<iomanip>
using namespace std;

void citire(int a[10][10],int m,int n)
  { int i,j;
    for(i=0;i<m;i++)
      { cout<<"Linia "<<i+1<<":";
       // tastaţi n numere separate prin spaţiu
        for(j=0;j<n;j++) cin>>a[i][j];
      }
  }

void suma(int a[10][10],int b[10][10],int c[10][10],
          int m,int n)
  { int i,j;
    for(i=0;i<m;i++)
      for(j=0;j<n;j++)
        c[i][j]=a[i][j]+b[i][j];
  }

void produs(int a[10][10],int b[10][10],int c[10][10],
            int m,int n,int p)
    // a cu m linii si n coloane,  b cu n linii si p coloane,  c cu m linii si p coloane
  { int i,j,k,s;
    for(i=0;i<m;i++)
      for(j=0;j<p;j++)
       { s=0;
         for(k=0;k<n;k++)
           s+=a[i][k]*b[k][j];
         c[i][j]=s;
       }
  }

void afisare(int a[10][10],int m,int n)
  { int i,j;
    for(i=0;i<m;i++)
      { for(j=0;j<n;j++)
         cout<<setw(5)<<a[i][j];
        cout<<endl;
      }
  }

int main()
  {
    int x[10][10],y[10][10],z[10][10];
    int n;
    cout<<"n="; cin>>n;   // n linii, n coloane
    cout<<"Matricea X:"<<endl;
    citire(x,n,n);
```

```
    cout<<"Matricea Y:"<<endl;
    citire(y,n,n);
    suma(x,y,z,n,n);
    cout<<"Z=X+Y:"<<endl;
    afisare(z,n,n);
    produs(x,y,z,n,n,n);
    cout<<"Z=X*Y:"<<endl;
    afisare(z,n,n);
    return 0;
  }
```

2) Parcurgere în spirală

*Fie a o matrice cu m linii şi n coloane. Să se construiască vectorul b cu m*n elemente obţinute prin parcurgerea matricei în spirală, din colţul din stânga-sus către dreapta, până în centrul matricei.*

```
#include<iostream>
#include<iomanip> //pentru setw
using namespace std;

void citireMatrice(int a[10][10],int m, int n)
 { int i,j;
   for(i=0;i<m;i++)
    { cout<<"linia "<<i<<":";
      for(j=0;j<n;j++)
       cin>>a[i][j];
    }
 }
void afisareMatrice(int a[10][10],int m,int n)
 { int i,j;
   for(i=0;i<m;i++)
    { for(j=0;j<n;j++)
       cout<<setw(6)<<a[i][j];
     cout<<endl;
    }
 }

void parcurgereInSpirala(int a[10][10], int m, int n,
                         int b[], int mn )
 { int k,p,q,r,i,j;
   k=0;
   p=0; q=m-1; r=n-1; //p,q,r definesc conturul de parcurs (p,p)->(p,q)->(q,r)->(r,p-1)
   while(k<m*n)
    { for(j=p;j<=r;j++)b[k++]=a[p][j];    //parcurgere stanga->dreapta
      for(i=p+1;i<=q;i++)b[k++]=a[i][r];  //sus ->jos (col. din dreapta)
      for(j=r-1;j>=p;j--)b[k++]=a[q][j];  // dreapta->stanga(linia de jos)
      for(i=q-1;i>=p+1;i--)b[k++]=a[i][p]; // jos->sus (col din stanga)
      p++;q--;r--; //urmatorul contur
    }
 }

void afisareVector(int b[],int n)
 { for(int i=0;i<n;i++)
    cout<<b[i]<<" "; }

int main()
 {int a[10][10],m,n;
  int b[100],mn;
```

```
  cout<<"nr. linii="; cin>>m;
  cout<<"nr. coloane="; cin>>n;
  mn=m*n;
  citireMatrice(a,m,n);
  afisareMatrice(a,m,n);
  parcurgereInSpirala(a, m, n, b, mn);
  cout<<"Vectorul obtinut prin parcurgerea matricei in spirala:"<<endl;
  afisareVector(b,mn);
  return 0;
 }
```

3) Problema celor 4 triunghiuri

O matrice pătratică este împărţită de cele două diagonale în patru triunghiuri. Să se determine suma elementelor din cele patru triunghiuri. Elementele de pe diagonale fac parte din triunghiurile respective.

```
#include<iostream>
using namespace std;

long sumaTrSus(long a[20][20],int n)
  { long S=0;
    int p,q,k;
    p=0;q=n-1;
    while(p<=q)
     { for(k=p;k<=q;k++)
         S+=a[p][k];
       p++;q--;
     }
    return S;
  }

long sumaTrJos(long a[20][20],int n)
  { long S=0;
    int p,q,k;
    p=0;q=n-1;
    while(p<=q)
     { for(k=p;k<=q;k++)
         S+=a[q][k];
       p++;q--;
     }
    return S;
  }

long sumaTrStanga(long a[20][20],int n)
  { long S=0;
    int p,q,k;
    p=0;q=n-1;
    while(p<=q)
     { for(k=p;k<=q;k++)
         S+=a[k][p];
       p++;q--;
     }
    return S;
  }
```

```
long sumaTrDreapta(long a[20][20],int n)
  { long S=0;
    int p,q,k;
    p=0;q=n-1;
    while(p<=q)
     { for(k=p;k<=q;k++)
         S+=a[k][q];
       p++;q--;
     }
    return S;
  }

int main()
  {long a[20][20]={{1,2,3,4,5},
                   {1,2,3,4,5},
                   {1,2,3,4,5},
                   {1,2,3,4,5},
                   {1,2,3,4,5}};
   cout<<sumaTrSus(a,5)<<endl;
   cout<<sumaTrStanga(a,5)<<endl;
   cout<<sumaTrJos(a,5)<<endl;
   cout<<sumaTrDreapta(a,5)<<endl;
   return 0;
  }
```

4) Căutare binară

Algoritmul de căutare binară este un algoritm de căutare folosit pentru a găsi un element într-un vector ordonat. Fie a un vector ordonat crescător şi x un element ce se caută în vectorul a. Valoarea x este comparată cu valoarea elementului din mijlocul vectorului a. Dacă cele două valori sunt egale, algoritmul se termină. Dacă valoarea lui x este mai mică decât acea valoare, căutarea se efectuează, prin acelaşi procedeu, pentru elementele de la începutul vectorului până la mijloc, iar dacă este mai mare, căutarea se efectuează de la mijlocul vectorului până la sfârşitul său. Întrucât la fiecare pas cardinalul mulţimii de elemente în care se efectuează căutarea se înjumătăţeşte, algoritmul are complexitate logaritmică.

```
#include<iostream>
#include<stdlib.h>
using namespace std;

int caut(long a[],int n,long x)
 { int i=0,j=n-1,m;
   while(i<=j)
     { m=(i+j)/2;
       if(a[m]==x) return m;// x se gaseste pe pozitia m
       if(x<a[m])j=m-1;
       else i=m+1;
     }
   return -1; // x nu se gaseste in vectorul a
 }

int main()
{ long a[100], x;
  int n, poz;
  cout<<"n=";
  cin>>n;
```

```
  cout<<"Tastati "<<n<<" elemente in ordine crescatoare, "
      <<"separate prin spatiu:"<<endl;
  cin>>a[0];
  for(int i=1;i<n;i++)
  { cin>>a[i];
    if(a[i]<a[i-1])
    { cout<<"vectorul nu este ordonat crescator"<<endl;
      return 0;
    }
  }

  while(1)
  { cout<<"Tastati valoarea lui x "
       <<" (sau CTRL+Z pentru sfarsit):";
    cin>>x;
    if(cin.eof()) break;

    poz=caut(a,n,x);
    if(poz>=0)
      cout<<x<<" se gaseste in vectorul a pe pozitia "
          <<poz+1<<endl;
    else
     cout<<x<<" nu se gaseste in vectorul a"<<endl;
   }
   return 0;
 }
```

5) Exemple de funcţii de lucru cu şiruri de caractere:

Determinarea numărului de caractere dintr-un şir de caractere:

```
int lungime(char s[])
    { int k=0;
      while(s[k]) k++;
      return k;
    }
```

Copierea unui şir de caractere:

```
void copiaza(char sursa[], char dest[])
    { int i=0;
      while(dest[i++]=sursa[i]);
    }
```

Observaţie: Biblioteca <*string.h*> conţine funcţii pentru determinarea lungimii şi pentru copierea şirurilor(*strlen* respectiv *strcpy*).

Inversarea caracterelor unui şir de caractere:

```
void inversare(char s[])
  { int i,j;
    char c;
    for(i=0,j=lungime(s)-1; i<j; i++,j--)
      { c=s[i]; s[i]=s[j]; s[j]=c; }
  }
```

Deterrminarea primei poziţii de unde începe un subşir într-un şir de caractere.

```
#include<iostream>
using namespace std;

int lungime(char s[])
    { int k=0;
      while(s[k]) k++;
      return k;
    }

int cauta(char sir[], char subsir[])
  { int k,j,lgSubsir,pozMax;
    lgSubsir=lungime(subsir);
    pozMax=lungime(sir)-lgSubsir;
    for(k=0; k<= pozMax; k++)
      { for ( j=0; j<lgSubsir && subsir[j]==sir[k+j]; j++ );
          if(j==lgSubsir) return k;
      }
    return -1;
  }

int main()
  { char a[25],b[25];
    int r;
    cout<<"Sirul a=";  cin>>a;
    cout<<"Sirul b=";  cin>>b;
    cout<<"Sirul a are "<<lungime(a)<<" caractere."<<endl;
    cout<<"Sirul b are "<<lungime(b)<<" caractere."<<endl;
    r=cauta(a,b);
    if (r>=0)
      cout<<"Subsirul b apare in a incepand de la poz. "<<r+1<<endl;
    else cout<<"Subsirul b NU apare in a";
    return 0;
  }
```

6) Se introduce un text de la tastatură. Să se afişeze frecvenţa literelor mari şi a literelor mici din textul introdus.

```
#include  <iostream>
#include  <string.h> //pentru strlen
#include  <ctype.h> // pentru islower si isupper
using namespace std;

int main()
{ int i, n, a[26], b[26] ;
  char sir[1000],c ;
  cout<<"Introduceti sirul:";
  cin>>sir;
  n=strlen(sir);
  for(i=0;i<26;i++)
    {a[i]=0; b[i]=0;}

  for(i=0;i<n;i++)
   {c=sir[i];
    if(islower(c))a[c-'a']++;
    if(isupper(c))b[c-'A']++;
   } //'a'=97, ...,'z'=122; 'A'=65, ...,'Z'=90
```

```
  for(i=0;i<26;i++)
   { if(a[i]!=0) cout<<"Litera "<<(char)(i+97)<<" apare de "<<a[i]
                     <<" ori \n";
     if(b[i]!=0) cout<<"Litera "<<(char)(i+65)<<" apare de "<<b[i]
                     <<" ori \n";
   }
  return 0;
 }
```

7) Operaţii cu numere naturale mari: suma, diferenţa aritmetică, compararea şi produsul a două numere mari; împărţirea unui număr mare la un număr întreg. Algoritmi clasici.

Vom considera numerele reprezentate în baza 10. Cifrele numerelor mari, completate cu zerouri nesmnificative, vor fi memorate în vectori cu acelaşi număr de elemente. Dimensiunea comună a vectorilor este aleasă astfel încât să fie suficientă şi pentru memorarea numerelor rezultate in urma operaţiilor aplicate lor.

```
#include <iostream>
#include<string.h>
using namespace std;

int dim=100;    // dimensiunea comuna a vectorilor(nr maxim de cifre); valoare implicita 100

int suma( int u[],int v[],int w[]) //w=u+v
 {int t=0,c;  // t - cifra de transport
  for(int i=dim-1; i>=0;i--)
   { c=u[i]+v[i]+t;
     if(c>=10) { w[i]=c-10; t=1;} else {w[i]=c; t=0;}
   }
  return 1-t;  // 1 adunare cu succes, 0 - depasire
 }

int diferenta(int u[],int v[], int w[]) // w=u-v, unde u>v
 {int t=0,c;
  for(int i=dim-1;i>=0;i--)
   { c=u[i]-v[i]+t;
     if(c<0){w[i]=10+c; t=-1;}
     else {w[i]=c; t=0;}
   }
  return 1+t;
 }

int comparare(int u[], int v[]) //1 daca u>v, 0 daca u=v, -1 daca u<v
 {
  for (int i=0;i<dim;i++)
   {  if(u[i]<v[i])return -1;
      if (u[i]>v[i]) return 1;
   }
  return 0;
 }


/* Produsul a doua numere mari se calculeaza inmultind fiecare cifra a inmultitorului cu fiecare cifra a deinmultitului
si adunind rezultatul la ordinul de marime corespunzător. */
```

```
int produs(int u[],int v[],int w[]) //w=u·v
{int i,j,t=0,s;
  for (i=0;i<dim;i++) w[i]=0;
  for( j=dim-1;j>=0;j--)
   for(i=dim-1;i>=0;i--)
    { int k=i+j+1-dim;
      s=u[i]*v[j]+t;
      if(k>=0)
       { w[k]+=s;
         t=w[k]/10;
         w[k]%=10;
             }
      else if(s!=0)return 0;// esec! dimensiune prea mica
    }
  return t==0?1:0;   //1 succes; 0 esec
 }

void impartire(int u[],long q,int w[],long &r) //w=u·q+r
 { int i;
   long p;
   r=0;
   for(i=0;i<dim;i++)
    {p=r*10+u[i];
     w[i]=p/q;
     r=p%q;
    }
 }

void afisare( int u[])
 { int i=0;
   while(i<dim-1 && u[i]==0)i++;  // salt peste cifrele 0 nesemnificative
   while(i<dim){ cout<<u[i]; i++;}
}

int citire(int v[])
 { char s[255];
   cin>>s;
   int m=strlen(s);
   if(m>dim) return 0;  //esec; dimensiune insuficienta
   int i,j;
   for(i=m-1,j=dim-1; i>=0; i--,j--)
    {v[j]=s[i]-'0';
     if(v[j]<0||v[j]>9) return 0;  //esec; caracter nenumeric
    }
  for( ;j>=0;j--)v[j]=0;
  return 1;  //citire cu succes
}

int main()
{ int u[100], v[100],s[100],d[100],p[100],r[100];
 dim=100;
  cout<<"u=";
  citire(u);
  cout<<"v=";
  citire(v);

  suma(u,v,s);
  afisare(u); cout<<"+"; afisare(v);
  cout<<"="; afisare(s); cout<<endl;
```

```
  if(comparare(u,v)>0)diferenta(u,v,d);
  else diferenta(v,u,d);
  cout<<"|";afisare(u); cout<<"-"; afisare(v);
  cout<<"| ="; afisare(d); cout<<endl;

  int ok=produs(u,v,p);
  afisare(u); cout<<"*"; afisare(v);
  cout<<"="; afisare(p);
  if(!ok)cout<<"esec! dim prea  mica"; cout<<endl;

  long q,rest;
  cout<<"q="; cin>>q;
  impartire(u,q,r,rest);
  afisare(u); cout<<":"; cout<<q<<"="; afisare(r);
  cout<<" rest "<<rest<<endl;
  return 0;
}
```

8) Fiind date n numere întregi a_1, a_2,..., a_n nu în mod necesar diferite, există totdeauna o submulţime a acestei mulţimi de numere, cu proprietatea că suma elementelor sale este divizibilă prin n. Următorul program determină o astfel de submulţime.

Soluţie. Fie

$$s_k = a_1 + a_2 + \ldots + a_k,\ k = 1,\ldots,n$$

Dacă $(\exists)s_k$ a.î. $n \mid s_k$ problema este rezolvată, altfel resturile sunt nenule $(\forall)$ $k \in \{1,2,\ldots,n\}$ şi aparţin mulţimii $\{1,\ldots,n-1\}$ $\Rightarrow$ $(\exists)$ două sume cu resturi egale. Fie s_k şi s_i, $i<k$, două asemenea sume $\Rightarrow$ $n \mid s_k - s_i$.
Dacă s_k `mod n = r`, vom memora indicele `k` în variabila `rest[r]`. În situaţia în care în `rest[r]` este deja memorat un indice nu ne rămâne decât să afişăm soluţia.

```
#include<iostream>
using namespace std;

int a[101],rest[101],n;
void citeste()
  { cout<<"n="; cin>>n;
    cout<<"Tastati "<<n<<" numere:";
    int i;
    for(i=1;i<=n;i++){ cin>>a[i];rest[i]=-1;}
    rest[0]=0;
  }

void solutie()
  { int s,k,i,r;
    s=0;
    for(k=1;k<=n;k++)
     {s+=a[k];
      r=s%n;
      if(rest[r]==-1) rest[r]=k;
      else { cout<<"Solutie:";
             for(i=rest[r]+1; i<=k; i++)
                cout<<a[i]<<" ";
             cout<<endl;
             return;
```

```
          }
      }
  }

int main()
  { citeste();
    solutie();
    return 0;
  }
```

9) Prelucrarea matricelor

Fie o matrice $A=(A_{ij})_{i=1,m,\ j=1,n}$, deci A are m linii și n coloane și are forma:

$$A=\begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2n} \\ \cdots & & & \\ A_{m1} & A_{m2} & \cdots & A_{mn} \end{pmatrix}$$

Linia i a matricei A are elementele: $A_{i1}\ A_{i2}\ \ldots\ A_{i,n-1}\ A_{in}$

Coloana j a matricei A are elementele: $A_{1j}\ A_{2j}\ \ldots\ A_{m-1,j}\ A_{mj}$

Dacă $A=(A_{ij})_{i,j=1,n}$ este o matrice pătratică, deci A are n linii și n coloane și are forma:

$$A=\begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2n} \\ \cdots & & & \\ A_{n1} & A_{n2} & \cdots & A_{nn} \end{pmatrix}$$

atunci:

diagonala principală a matricei A are elementele: $A_{11}\ A_{22}\ \ldots\ A_{n-1,n-1}\ A_{nn}$

diagonala secundară a matricei A are elementele: $A_{1n}\ A_{2,n-1}\ \ldots\ A_{n-1,2}\ A_{n1}$

elementele situate **deasupra diagonalei principale** sunt:

$$\begin{array}{llll} A_{12} & A_{13} & \ldots & A_{1,n-1} \quad A_{1,n} \\ & A_{23} & \ldots & A_{2,n-1} \quad A_{2,n} \\ & & \ldots & \\ & & & A_{n-2,n-1} \quad A_{n-2,n} \\ & & & \qquad\qquad A_{n-1,n} \end{array}$$

elementele situate **sub diagonala principală** sunt:

$$
\begin{array}{llll}
A_{21} & & & \\
A_{31} & A_{32} & & \\
\multicolumn{4}{l}{\ldots} \\
A_{n-1,1} & A_{n-1,2} & \ldots & A_{n-1,n-2} \\
A_{n1} & A_{n2} & \ldots & A_{n,n-2} \quad A_{n,n-1}
\end{array}
$$

elementele situate **deasupra diagonalei secundare** sunt:

$$
\begin{array}{llllll}
A_{11} & A_{12} & \ldots & A_{1,n-2} & A_{1,n-1} \\
A_{21} & A_{22} & \ldots & A_{2,n-2} & \\
\multicolumn{5}{l}{\ldots} \\
A_{n-2,1} & A_{n-2,2} & & & \\
A_{n-1,1} & & & &
\end{array}
$$

elementele situate **sub diagonala secundară** sunt:

$$
\begin{array}{llllll}
 & & & & A_{2n} \\
 & & & A_{3,n-1} & A_{3n} \\
 & & \ldots & & \\
 & A_{n-1,3} & \ldots & A_{m-1,n-1} & A_{n-1,n} \\
A_{n2} & A_{n3} & \ldots & A_{n,n-1} & A_{nn}
\end{array}
$$

Program pentru calculul elementului maxim de pe linia *i* a unei matrice *A*:

```
#include<iostream>
using namespace std;

int main()
{
    int m,n,i,j;
    double A[100][100],Max;
    cout<<"Nr linii m="; cin>>m;
    cout<<"Nr coloane n="; cin>>n;
    cout<<"Matricea A:"<<endl;
    for(i=1;i<=m;i++)
        for(j=1;j<=n;j++) cin>>A[i][j];
    cout<<"Numarul liniei i="; cin>>i;
    Max=A[i][1];
    for(j=2;j<=n;j++)
        if (A[i][j]>Max) Max=A[i][j];
    cout<<"Elementul maxim de pe linia "<<i<<" este Max="<<Max;
    return 0;
}
```

Program pentru calculul sumei elementelor impare de pe coloana *j* a unei matrice *A*:

```
#include<iostream>
using namespace std;
```

```
int main()
{
    int A[100][100],m,n,i,j,S,nr;
    cout<<"Nr linii m="; cin>>m;
    cout<<"Nr coloane n="; cin>>n;
    cout<<"Matricea A:"<<endl;
    for(i=1;i<=m;i++)
        for(j=1;j<=n;j++) cin>>A[i][j];
    cout<<"Numarul coloanei j="; cin>>j;
    S=0; nr=0;
    for(i=1;i<=m;i++)
        if (A[i][j]%2!=0)
        {  S=S+A[i][j];
           nr++;
        }
    if(nr==0) cout<<"Nu exista elemente impare pe coloana "<<j;
    else cout<<"Suma elementelor impare de pe coloana "<<j<<
                " este S="<<S;
    return 0;
}
```

Program pentru calculul numărului de elemente pozitive de pe diagonala principală a unei matrice pătratice *A*:

```
#include<iostream>
using namespace std;

int main()
{
    int n,i,j,nr;
    double A[100][100];
    cout<<"Nr linii(coloane) n="; cin>>n;
    cout<<"Matricea A:"<<endl;
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++) cin>>A[i][j];
    nr=0;
    for(i=1;i<=n;i++)
        if (A[i][i]>0) nr++;
    cout<<"Nr. de elem. pozitive de pe diag. principala este nr="
        <<nr;
    return 0;
}
```

Program pentru calculul produsului elementelor nenule de pe diagonala secundară a unei matrice pătratice *A*:

```
#include<iostream>
using namespace std;

int main()
{
    int n,i,j,nr;
    double A[100][100],P;
    cout<<"Nr linii(coloane) n="; cin>>n;
    cout<<"Matricea A:"<<endl;
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++) cin>>A[i][j];
    P=1; nr=0;
```

```
    for(i=1;i<=n;i++)
        if (A[i][n+1-i]!=0)
        {  P=P*A[i][n+1-i];
           nr++;
        }
    if (nr==0) cout<<"Nu exista elem. nenule pe diag. secundara";
    else cout<<"Produsul elem. nenule de pe diag. secundara este P="
             <<P;
    return 0;
}
```

Program pentru calculul mediei elementelor pare situate sub diagonala principală a unei matrice pătratice *A*:

```
#include<iostream>
using namespace std;
int main()
{   int A[100][100],n,i,j,nr;
    double S,M;
    cout<<"Nr linii(coloane) n="; cin>>n;
    cout<<"Matricea A:"<<endl;
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++) cin>>A[i][j];
    S=0; nr=0;
    for(i=2;i<=n;i++)
        for(j=1;j<=i-1;j++)
           if (A[i][j]%2==0)
           {  S=S+A[i][j];
              nr++;
           }
    if (nr==0) cout<<"Nu exista elem. pare sub diag. principala";
    else
    { M=S/nr;
      cout<<"Media elem. pare situate sub diag. principala este M="
          <<M;
    }
    return 0;
}
```

Program pentru calculul elementului minim situat deasupra diagonalei secundare a unei matrice pătratice *A*:

```
#include<iostream>
using namespace std;
int main()
{   int n,i,j;
    double A[100][100],min;
    cout<<"Nr linii(coloane) n="; cin>>n;
    cout<<"Matricea A:"<<endl;
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++) cin>>A[i][j];
    min=A[1][1];
    for(i=1;i<=n-1;i++)
        for(j=1;j<=n-i;j++)
           if (A[i][j]<min) min=A[i][j];
    cout<<"Elem. minim situat deasupra diag. secundare este min="
        <<min;
    return 0;
}
```

3.14 Pointeri

Pointerii se utilizează pentru a face referire la date prin adresele lor.
Într-o variabilă de tip pointer putem păstra adresa unei date în loc de a memora data însăşi sau putem păstra adresa unei funcţii. Schimbând adresa memorată în pointer, putem manipula informaţii din diverse locaţii de memorie.

Ca orice tip de variabilă, înainte de a fi utilizată, variabila de tip pointer trebuie declarată.

Pointerii oferă posibilitatea de a aloca dinamic memoria, ceea ce înseamnă că pe parcursul execuţiei unui program se pot aloca şi dealoca zone de memorie asociate lor.

O variabilă de tip pointer se declară utilizând formatul:

```
<tip>* <nume>;
```

ceea ce înseamnă că <*nume*> este un pointer către o zonă de memorie ce conţine o dată de tipul <*tip*>.

Caracterul "*" poate fi alăturat de <*tip*> sau de <*nume*> sau poate fi separat prin caractere spaţiu şi de <*tip*> şi de <*nume*>. El indică compilatorului că a fost declarată o variabilă pointer şi nu una obişnuită.

Construcţia <*tip*>* se spune ca reprezintă tipul *pointer*.

Exemplul 1:
```
int *p;
```
Aici se stabileşte faptul că `p` va conţine adrese de zone de memorie alocate datelor de tip `int.`

Exemplul 2:
```
float *p, t, *q;
```
Aici `p` şi `q` sunt pointeri către date de tip float, iar t este o variabilă de tip float.

Utilizarea pointerilor se face cu doi operatori unari:
`&` - *operatorul adresa (de referentiere)* - pentru aflarea adresei din memorie a unei variabile;
`*` - *operatorul de indirectare (de deferentiere)* - care furnizează valoarea din zona de memorie spre care pointează pointerul operand.

Dacă `x` este o variabilă atunci operatorul unar `&` aplicat lui `x, &x`, ne furnizează adresa lui `x`. Dacă dorim ca pointerul `p` să indice pe `x`, putem utiliza atribuirea:
```
p=&x;
```

Dacă p este o variabilă de tip pointer atunci operatorul unar * aplicat lui `p, *p,` ne furnizează variabila a cărei adresă este memorată în `p`.

Exemplu:
```
int a,*adr;
adr=&a;    // acum a şi *adr reprezintă aceeaşi dată.
a=100;     // este echivalentă cu: *adr=100;
*adr=200;  // este echivalentă cu: a=200;
```

Există cazuri în care dorim ca un pointer să fie utilizat cu mai multe tipuri de date. În acest caz, la declararea lui nu dorim să precizăm un tip anume. Aceasta se realizează astfel:

```
void *<nume>;
```

Utilizarea tipului `void*` implică conversii explicite de tip.
Exemplu:

```
void *p;
int x;

p=&x;  // Atribuire neacceptată deoarece tipul pointerului p este nedeterminat

(int*)p=&x;  //Atribuire corectă: tipul void* este convertit spre int*

*p=10;    // Atribuire neacceptată deoarece tipul pointerului p este nedeterminat

*(int*)p=10;  // Atribuire corectă: tipul void* este convertit spre int*
```

Deoarece pointerii reprezintă adrese, ei se folosesc la transferul prin referinţă al parametrilor.

Exemplu:

```
#include<iostream>
using namespace std;

void interschimba(int *x,int *y)
  { int aux=*x;
    *x=*y;
    *y=aux;
  }

int main()
  { int a,b;
    cout<<"a="; cin>>a;
    cout<<"b="; cin>>b;
    interschimba (&a,&b);
    cout<<"a="<<a<<"  b="<<b<<endl;
    return 0;
  }
```

În funcţia *interschimba*, parametrii formali `x, y` sunt pointeri la `int`, astfel încât la apel, parametrii actuali trebuie să fie adrese ale unor variabile de tip `int,` şi nu valori întregi. Orice modificare se va face la adresele transmise ca parametri actuali.

O variabilă pointer poate fi iniţializată cu adresa unei variabile, astfel:

```
int x=10;
int *p=&x;  // pointerul p se iniţializează cu adresa variabilei x
```

Un pointer la caractere poate fi iniţializat ca în exemplul următor:

```
char *q="abc";       // q memorează adresa  de unde începe şirul "abc"
```

Operaţii cu pointeri

Asupra pointerilor se pot face următoarele operaţii: atribuire, comparare, adunare, scădere, incrementare, decrementare.
Adunarea şi scăderea unui întreg dintr-un pointer.
Dacă `p` este un pointer având declaraţia:

```
<tip> *p;
```

atunci p+n furnizează valoarea lui p mărită cu n*sizeof(<*tip*>), iar p–n valoarea lui p micşorată cu n*sizeof(<*tip*>).
Asupra pointerilor se pot face operaţii de incrementare şi decrementare:
p++ şi ++p măresc adresa conţinută de p cu sizeof(<tip>)
p--, --p micşorează valoarea pointerului p cu sizeof(<tip>).

Exemple:

```
char *c;
int *k;
float *f1,*f2;
double *d;
c++;        //c= (adresa memorată în c) + 1
k+=5;       // k=(adresa memorată în k) + 5*2
f2=f1-5;    // f2=(adresa memorată în f1) – 5*4
d-=3;       //d=(adresa memorată în d) – 3*8
d--;        //d=(adresa memorată în d) – 1*8
```

Operaţiile de incrementare şi decrementare se pot aplica pointerului însuşi sau obiectului pe care-l punctează (memorează).
Instrucţiunea *a++ obţine mai întâi valoarea pe care o punctează a şi apoi a este incrementat pentru a puncta elementul următor.
Instrucţiunea (*a)++ incrementează obiectul pe care-l punctează a.

Legătura dintre pointeri şi tablouri

Numele unui tablou este un pointer şi el are ca valoare adresa primului său element.
Fie

```
int t[5];
int *p;
p=t;   // Atribuire corectă!
```

p va pointa spre primul element al tabloului t. Între p şi t există o diferenţă şi anume: valoarea lui p poate fi modificată dar a lui t nu poate fi modificată (t este un pointer constant), deci este interzisă o atribuire de forma t=p;

Un nume de tablou poate fi utilizat ca şi cum ar fi un pointer şi, reciproc, un pointer poate fi indexat ca şi cum ar fi un tablou.
Dacă x este un tablou:

```
<tip> x[<dim>];
```

atunci expresia x+n este corectă şi reprezintă un pointer către al n-lea element al tabloului (x+n==&x[n]), deci x[n] este echivalentă cu *(x+n).
Dacă p este un pointer:

```
<tip> *p;
```

atunci *(p+i) este echivalentă cu p[i].

O diferenţă între pointer şi tablou constă în alocarea de memorie. În cazul tabloului, se rezervă automat spaţiul necesar. În cazul pointerilor, spaţiul trebuie creat explicit de utilizator sau trebuie atribuită pointerului o adresă a unui spaţiu deja alocat.
Fie declaraţia

```
int t[5];
```

Elementele tabloului t sunt memorate în celule succesive de memorie şi sunt numerotate începând cu zero.
În exemplul nostru vom avea:

```
t[0], t[1], t[2], t[3], t[4].
```

Deoarece t[0] este o variabilă simplă, adresa sa este &t[0], deci vom avea:
```
t==&t[0].
```
Elementele unui tablou multidimensional sunt memorate în ordinea crescătoare a liniilor.
De exemplu, elementele tabloului descris prin:
```
int a[2][3];
```
vor fi memorate în ordinea
```
a[0][0], a[0][1], a[0][2], a[1][0], a[1][1], a[1][2].
```
a reprezintă adresa primului element din tablou. Tabloul a, având două linii, este considerat ca un tablou cu două elemente (tablouri unidimensionale): elementele a[0] şi a[1]. a este adresa elementului a[0], adică avem a==&a[0]. Cum a[0] este şi el un tablou, înseamnă ca a[0] este adresa primului său element, deci a[0]==&a[0][0]. Dar, cele două tablouri, tabloul bidimensional a[][], şi tabloul liniar a[0] încep de la aceeaşi adresă, astfel că:
```
a==&a[0]==a[0]==&a[0][0]
```
Dacă privim tabloul a ca o variabilă structurată, atunci adresa acestei variabile se poate afla cu &a, deci avem lanţul de egalitaţi:
```
&a==a==a[0]==&a[0]==&a[0][0].
```

Compararea a doi pointeri

Dacă doi pointeri pointează spre elementele aceluiaşi tablou, pot fi comparaţi folosind operatorii de relaţie şi egalitate.
Astfel
```
<tip> t[dim];
<tip> *p,*q;
```
Dacă p pointează spre t[i] şi q spre t[j], atunci
p<q dacă i<j,
p!=q dacă i≠j.
Un pointer mai poate fi comparat cu constanta NULL (zero binar) utilizând operatorii == şi !=. Astfel stabilim dacă o variabilă pointer conţine sau nu o adresă.
Doi pointeri care pointează elementele aceluiaşi tablou *pot fi scăzuţi*, astfel dacă p pointează pe t[i] şi q pe t[i+n], atunci q–p are valoarea n.
Observaţie. Nu se admite adunarea a doi pointeri.

Tablouri de pointeri

Datele de tip pointer pot fi organizate în tablouri la fel ca şi alte tipuri de date. Pentru a descrie un tablou de pointeri se foloseşte o construcţie de forma:
```
<tip> *<nume>[<dim>];
```
unde <*nume*> este un tablou având <*dim*> elemente de tip pointer ce memorează adrese ale unor date de tipul <*tip*>.
Exemplu:
```
int DenumireZi(int m)
  { static char *zi[7]={"luni","marti","miercuri",
                        "joi","vineri","sambata","duminica"};
    if (m<1||m>7) return 0; else {cout<<zi[m-1]; return 1;}
  }
```

Pentru memorarea denumirilor zilelor s-ar fi putut folosi un tablou cu două dimensiuni care să păstreze pe fiecare linie câte un şir de caractere corespunzător numelui zilei.
```
char nume[7][10]={"luni","marti","miercuri","joi","vineri",
```

```
                    "sambata","duminica"};
```

Soluţia cu pointeri are avantajul că liniile tabloului pot fi de lungimi diferite, conducând la o reprezentare eficientă a datelor.

În exemplul de mai jos, funcţia zi returnează un pointer către un şir de caractere:

```
#include<iostream>
using namespace std;

char* zi(int m)
 { static char *z[7]={"luni", "marti", "miercuri",
                 "joi","vineri","sâmbată","duminică"};
   if(m<1 || m>7) return 0; //pointer NULL
   return z[m-1];
 }

int main()
 { int n;
   char *pZi;
   cin>>n;
   pZi=zi(n);
   cout<<pZi<<endl;
   return 0;
 }
```

Exemple de funcţii de lucru cu şiruri de caractere. Varianta cu pointeri.

1) *Determinarea numărului de caractere dintr-un şir de caractere:*

```
int lungime(char *sir)
 {
   for(int k=0; *sir++; k++);
   return k;
 }
```

2) *Copierea unui şir de caractere:*

```
void copiaza(char *sursa, char *dest)
   { while(*dest++=*sursa++);  }
```

```
#include<iostream>
using namespace std;

int main()
  { char a[25],b[25];
    cout<<"Sirul a=";  cin>>a;
    copiaza(a,b);
    cout<<"Sirul a are "<<lungime(a)<<"  caractere."<<endl;
    cout<<"Sirul b este "<<b<<endl;
    return 0;
  }
```

3) *Funcţie pentru transformarea în majuscule a literelor unui sir de caractere*

```
#include<iostream>
using namespace std;
```

```
char* majuscule(char *s)
 { char *p=s;
   while(*p){ if(*p>='a'&&*p<='z') *p+='A'-'a';
              p++; }
   return s;
 }
int main()
{ char sir[1000];
  cout<<"Introduceti sirul:";  cin>>sir;
  cout<<majuscule(sir);
  return 0;
}
```

Observaţii:
-Incrementarea unui pointer este mai rapidă decât indexarea unui tablou.

-Biblioteca <*string.h*> conţine funcţii pentru determinarea lungimii unui şir de caractere, pentru compararea, localizarea şi copierea şirurilor, pentru transformarea caracterelor în majuscule sau minuscule(*strlen, strcmp, strchr, strstr, strcpy, strcat, strupr, strlwr etc.*).

3.15 Alocarea dinamică a memoriei

Necesitatea definirii în programe a datelor de tip dinamic, este dată de utilizarea mai bună a memoriei, lungimea unui program variind în funcţie de volumul datelor cu care se lucrează.

Limbajele C şi C++ îi permit utilizatorului să ceară în timpul rularii programului, în funcţie de necesităţi, să se aloce memorie suplimentară sau să se renunţe la ea.

Variabilele dinamice sunt acele variabile cărora, în mod explicit, li se alocă şi dealocă memorie şi a căror dimensiune se poate modifica pe parcursul execuţiei unui program în funcţie de opţiunile programatorului.

Zona de memorie în care se face alocarea dinamică a variabilelor se numeşte heap.

Alocarea dinamică se poate face pentru tipurile de date:
-fundamentale
-structurate: tablouri, liste, arbori etc.

Programele scrise în limbajul C standard, utilizează pentru alocarea dinamică a memoriei o familie de funcţii `malloc` şi `free`, destul de greoaie în folosire. Ele au fost păstrate în C++ doar pentru menţinerea compatibilităţii. În locul lor se folosesc operatorii `new` şi `delete`.

Operatorul `new` serveşte la alocarea dinamică a memoriei. El va returna un pointer la zona de memorie alocată dinamic. În cazul în care nu există memorie suficientă, alocarea nu va avea loc. Acest fapt se semnalează prin returnarea unui pointer `NULL` (zero binar). De aceea se recomandă ca, în cazul utilizării intensive a alocării dinamice, după fiecare utilizare a lui new să se testeze valoarea returnată.
Fie p un pointer către un tip de date, adică având o declaraţie de forma:

```
<tip> *p;
```

Operatorul `new` poate fi folosit utilizând următoarele formate:

```
p=new <tip>;
p=new <tip>(<expresie>);
p=new <tip>[<dim>];
```

În varianta 1. operatorul new alocă, dacă este posibil, spaţiul necesar tipului <*tip*>, şi returnează adresa zonei de memorie alocate.
În varianta 2. variabila dinamică creată cu new se iniţializează cu valoarea expresiei <*expresie*>.
Varianta 3. se foloseşte pentru alocarea a <*dim*> variabile dinamice de tipul <*tip*> (un tablou liniar cu <*dim*> elemente). Iniţializarea tablourilor nu este posibilă.
Exemple:

```
int *p, *q,*r;
p=new int;        // se alocă memorie pentru un întreg
q=new int(7);     // se alocă memorie pentru un întreg şi se iniţializează variabila cu 7
r=new long[10];   // se alocă un tablou de 10 întregi
```

Dezalocarea zonei de memorie alocată cu new, se face cu ajutorul operatorului delete, cu sintaxa:

```
delete <variabila>;
```

Exemple:

```
delete p;
delete r;
```

Prezentăm mai jos trei variante de utilizare a unui vector alocat dinamic:
1.

```
#include <iostream>
using namespace std;

int main()
  { int *a,n,i;
    cout<<"n=";
    cin>>n;
    a=new int[n];   // alocare vector
    for(i=0;i<n; i++)
      { cout<<"a"<<i<<"="; cin>>*(a+i);}
    cout<<"a=(";
    for(i=0;i<n; i++)
      cout<<*(a+i)<<",";
    cout<<"\b)"<<endl;
    delete a;
    return 0;
  }
```

2.

```
#include <iostream>
using namespace std;

int main()
  { int *a,n,i;
    cout<<"n=";
    cin>>n;
    a=new int[n];   // alocare vector
    for(i=0;i<n; i++)
      {cout<<"a"<<i<<"="; cin>>a[i];}
    cout<<"a=(";
    for(i=0;i<n; i++)
      cout<<a[i]<<",";
    cout<<"\b)"<<endl;
    delete a;
    return 0;
  }
```

3.
```
#include <iostream>
using namespace std;

int main()
 { int *a,n,i;
   cout<<"n="; cin>>n;
   a=new int[n]-1;   // astfel, elementele vectorului sunt a[1], a[2], ..., a[n]
   for(i=1;i<=n; i++) { cout<<"a"<<i<<"="; cin>>a[i]; }
   cout<<"a=(";
   for(i=1;i<=n; i++) cout<<a[i]<<",";
   cout<<"\b)"<<endl;
   a++; // revenire la adresa returnată de new
   delete a;
   return 0;
 }
```

3.16 Tipul referinţă

Pentru a simplifica lucrul cu pointeri, în C++ a fost introdus tipul *referinţă*. Tipul *referinţă* implementează perfect conceptul de transmitere a parametrilor prin referinţă.

O *referinţă* este un nume alternativ pentru un obiect.

Dacă avem tipul de dată `T`, prin `T&` sau `T &` (cu spaţiu după `T`) sau `T &` (cu spaţiu după `T` şi după `&`) vom înţelege o referinţă (o trimitere) la un obiect de tipul `T`.

Exemplu:
```
int x=10;
int& r=x;  // r şi x referă acum  acelaşi obiect
r=20;      // echivalent cu x=20;
x=30;      // echivalent cu r=30;
```
Variabila `r` de mai sus, este o referinţă la variabila `x` de tip `int`. Acest lucru înseamnă că identificatorii `r` şi `x` permit accesul la aceeaşi zonă de memorie. Prin urmare, `x` şi `r` sunt sinonime. Iniţializarea unei *referinţe* (trimiteri) în declaraţia sa este obligatorie (dacă nu este folosită ca argument al unei funcţii), dar această iniţializare nu trebuie confundată cu atribuirea; ea defineşte pur şi simplu un alt nume (un alias) al obiectului cu care a fost iniţializată. În exemplul de mai sus `r` este un nou nume pentru `x`. O referinţă nu mai poate fi modificată după iniţializare. Ea referă întotdeauna acelaşi obiect stabilit prin iniţializare să-l desemneze. Pentru a obţine un pointer la obiectul desemnat de referinţa `r`, se poate folosi `&r`.

Referinţele sunt utile şi când sunt folosite ca argumente pentru funcţii.

Exemplu:
```
#include<iostream>
using namespace std;
void interschimba(int& a,int& b)
  {int aux=a;  a=b;  b=c; }

int main()
  { int x=10,y=20;
    interschimba (x,y);
    cout<<"x="<<x<<"y="<<y;
    return 0 ;
  }
```

Semantica transmiterii argumentelor este aceea a iniţializării, la apel argumentele a şi b ale funcţiei *interschimba*, devin alte nume pentru variabilele x şi y şi de aceea operaţiile se fac direct asupra variabilelor x şi y.

Exemplu:
Intersecţia, reuniunea şi diferenţa a două mulţimi.

```
#include<iostream>
using namespace std;

void citire(int a[],int& n)
  { cout<<"numar de elemente:";cin>>n;
    int i;
    cout<<"tastati "<<n<<" elemente:";
    for(i=0;i<n;i++)cin>>a[i];
  }

void afisare(int a[],int n)
  { int i;
    cout<<"{";
    for(i=0;i<n;i++)cout<<a[i]<<" ";
    cout<<"}";
  }

void intersectie(int a[],int m,int b[],int n, int c[],int& p)
  { int i,j;
    p=0;
    for(i=0;i<m;i++)
     for(j=0;j<n;j++)
       if(a[i]==b[j]) { c[p++]=a[i];break;}
  }

void reuniune(int a[],int m,int b[],int n,int c[],int& p)
  { int i,j;
    for(i=0;i<m;i++)c[i]=a[i];
    p=m;
    int semafor; // semafor=1 daca b[j] apartine lui a  si semafor =0 daca nu apartine
    for(j=0;j<n;j++)
      { semafor =0; // b[j] nu apartine lui a
       for(i=0;i<m;i++)
         if(b[j]==a[i]) { semafor=1; break;}
       if(semafor==0)c[p++]=b[j];
      }
  }

void diferenta(int a[],int m,int b[],int n,int c[],int& p)
  { int i,j;
    p=0;
    int semafor;
    for(i=0;i<m;i++)
      { semafor=0;
        for(j=0;j<n;j++)
         if(a[i]==b[j]) { semafor=1;break;}
        if(semafor==0)c[p++]=a[i];
      }
  }
```

```
int main()
  { int u[100],v[100],w[100];
    int n1,n2,n3;
    cout<<"Multimea U:"<<endl;
    citire(u,n1);
    cout<<"Multimea V:"<<endl;
    citire(v,n2);
    int op;
    do
      {
        cout<<"U=";afisare(u,n1);cout<<endl;
        cout<<"V=";afisare(v,n2);cout<<endl;
        cout<<"1 - intersectie"<<endl;
        cout<<"2 - reuniune"<<endl;
        cout<<"3 - diferenta"<<endl;
        cout<<"4 - STOP"<<endl;
        cout<<"Tastati optiunea:"; cin>>op;

        switch(op)
         { case 1:intersectie(u,n1,v,n2,w,n3);
                  cout<<"U intersectat cu V=";
                  afisare(w,n3); cout<<endl;
                  break;
           case 2:reuniune(u,n1,v,n2,w,n3);
                  cout<<"U reunit cu V=";
                  afisare(w,n3);cout<<endl;
                  break;
           case 3:diferenta(u,n1,v,n2,w,n3);
                  cout<<"U-V="; afisare(w,n3); cout<<endl;
                  break;
         }
      }
    while(op!=4);
    return 0;
  }
```

Dacă tipul unei funcţii este o referinţă, atunci acea funcţie va întoarce o variabilă. În astfel de situaţii variabila returnată trebuie să fie statică sau alocată cu operatorul `new` ca mai jos:

Exemple:

```
int& f()
  { static int x;
    ...
    return x;
  }
```

Funcţia `f()` întoarce variabila de tipul `int` cunoscută în interiorul funcţiei prin identificatorul `x.` Au sens, `f()++` (incrementează variabila returnată) şi `&f()` (reprezentând adresa variabilei returnate).

```
int& g()
 { int& x= *new int;   //Am dat un nume variabilei anonime *new int
   ...
   return x;
 }
```

Funcţia `g()` întoarce variabila de tipul int alocată dinamic.

3.17 Tipul enumerare

Acest tip permite folosirea unor nume sugestive pentru valori numerice întregi.

O *enumerare* reprezintă o listă de constante întregi, pusă în corespondenţă cu o listă de identificatori.

Constantele întregi sunt adesea definite mai convenabil cu `enum`.

Exemplu:

```
enum{ IAN, FEB, MAR, APR, MAI};
```

defineşte patru constante întregi numite enumeratori şi le atribuie valori.

Deoarece valorile de enumerare sunt atribuite implicit începând de la zero, declaraţia de mai sus este echivalentă cu:

```
const int IAN=0, FEB=1, MAR=2, APR=3, MAI=4;
```

Dacă o enumerare poartă un nume, ca în exemplul de mai jos, acel nume devine sinonim cu `int`, nu este un nou tip.

Enumeratorii pot primi valori explicite care nu trebuie să fie distincte crescătoare sau pozitive.

Exemple:

```
enum taste {stanga=4,dreapta=6,sus=8,jos=2};
taste t;
...
t=dreapta;
...
```

```
enum culori {galben,albastru=5,rosu,verde=-5,alb} CULOARE;
```

În acest caz avem echivalenţa cu:

```
const galben=0, albastru=5, rosu=6, verde=-5, alb=-4;
```

În acest exemplu CULOARE este o variabilă de tip asumat `int`, deci putem scrie:

```
CULOARE=rosu;
```

sau

```
CULOARE=6;
```

sau

```
CULOARE=100;
```

3.18 Structuri şi uniuni

O *structură* reprezintă o colecţie de date de tipuri diferite.

Tipul unei astfel de date se spune că este definit de utilizator şi se numeşte *tip structurat*.

În C++ o structură se poate declara utilizând sintaxa:

```
struct <identificator structură>
  {<lista de declaraţii>} <listă de variabile>;
```

<listă de variabile> poate să lipsească.

Exemplu:

```
struct  complex{float re,im;} c1,c2,c3[10];
```

Variabilele c1 şi c2 sunt structuri cu câte două câmpuri de tip *float*, iar c3 este un tablou de asemenea structuri.

În C++ putem declara variabile de tip structurat prin:

```
<identificator structura> <lista de variabile>;
```

Exemplu:

```
complex z1,z2,*p;
```

Fie declaraţia:

```
complex x,y,*p;
```
Câmpurile componente ale unui date structurate pot fi referite în două feluri:

• *direct*, prin numele structurii urmat de "." şi de numele câmpului

Exemple: x.re, x.im, y.re, y.im

• *indirect*, prin adresa structurii urmată de "–>" şi de numele câmpului

Exemple:
```
p->re,  p->im
```
Această scriere este forma simplificată a scrierii `(*p).re, (*p).im`

Operatorul `"->"` are aceeaşi prioritate ca şi `"."`. Ambii operatori sunt de prioritate maximă.

Componentele unei date structurate pot fi ele însele date structurate.

Elementele unei date de tip structurat pot fi iniţializate astfel: în declaraţie, după numele variabilei structurate se scrie "=", iar după acesta, între acolade se iniţializează componentele structurii.

Exemple:
```
complex x={1,0},y={2,1};
struct student
  { char nume[20];
    int note[10];
  };
student s1={ "Popa Dan",{10,10,10,9,10}};
```

Probleme rezolvate

1) *În următorul program este definită structura student şi se exemplifică utilizarea ei.*

```
#include<iostream>
#include<iomanip.h>
using namespace std;

struct student
  { char nume[20];
    char adresa[40];
    long  telefon;
  };

int main()
 {student s[100]; // vector cu componente structurate
  int n;
  cout<<"n=";  cin>>n;
  for(int i=0;i<n; i++)
   { cin.get();       // extragerea caracterului <enter> din stream
     cout<<"student "<<i+1<<":"<<endl;
     cout<<" nume:";
     cin.getline(s[i].nume,30); // citeşte nume
     cout<<" adresa:";
     cin.getline(s[i].adresa,40); //citeşte adresa
     cout<<" telefon:"; cin>>s[i].telefon;
   }
  cout<<" Lista studentilor"<<endl;
  cout<<setiosflags(ios::left);   // aliniere la stânga
  for(int i=0;i<n;i++)
   { cout<<setw(20)<<s[i].nume<< setw(30)<<s[i].adresa
         <<s[i].telefon<<endl;
   }
  return 0;
 }
```

2) Calculul ariei unui poligon cu n laturi, n≥3, când se cunosc coordonatele rectangulare ale vârfurilor poligonului (x_i, y_i), *i=1, 2, ..., n utilizând formula:*

$$A=|\ (x_1+x_2)(y_1-y_2)+(x_2+x_3)(y_2-y_3)+...+\ \ (x_n+x_1)(y_n-y_1)\ |/2,$$

şi determinarea coordonatelor centrului de greutate al poligonului.

```
#include <iostream>
#include <iomanip>
#include <math.h>
using namespace std;

struct punct {float x,y;};

int cit( punct P[])
 { int n;
   cout<<"Nr. puncte=";  cin>>n;
   for(int i=0;i<n; i++)
    { cout<<"P"<<i+1<<"(x y)="; cin>>P[i].x>>P[i].y;
    }
   return n;
 }

float arie( punct P[],int n)
  { float s=(P[n-1].x+P[0].x)*(P[n-1].y-P[0].y);
    for(int i=0;i<n-1;i++)
      s+=(P[i].x+P[i+1].x)*(P[i].y-P[i+1].y);
    return fabs(s)/2;
  }

punct centruGr(punct P[],int n)
   // ne arată că o funcţie poate returna o dată structurată
  { punct G={0,0};
    for(int i=0;i<n;i++)
     { G.x+=P[i].x;  G.y+=P[i].y;
     }
    G.x/=n; G.y/=n;
    return G;
  }

int main()
  { punct P[100];
    int n =cit(P);
    float S=arie(P,n);
    punct C=centruGr(P,n);
    cout<<setprecision(4);
    cout<<"Aria poligonului="<<S<<endl;
    cout<<"Centru de greutate este C("<<C.x<<","
        <<C.y<<")"<<endl;
    return 0;
  }
```

O *uniune* este o structură de date care permite folosirea în comun a aceleiaşi zone de memorie, de două sau mai multe variabile diferite, la momente de timp diferite.

Forma generală a unei uniuni:

```
union nume_uniune
    {
     tip1 nume_câmp1;
     tip2 nume_câmp2;
        . . .
     tipn nume_câmpn;
    } lista_variabile_uniune;
```

Se observă că forma generală de declarare a unei uniuni este asemănătoare cu cea a unei structuri şi ceea ce s-a spus la structuri este valabil şi la uniuni.

Operatorul sizeof aplicat tipului de date union, adică sizeof (union nume_uniune) va furniza lungimea uniunii(lungimea celui mai mare membru al uniunii).

Deosebirea fundamentală dintre o uniune şi o structură constă în modul în care câmpurile folosesc memoria.

La structură, zonele de memorie rezervate câmpurilor sunt diferite pentru câmpuri diferite.

La uniune, toate câmpurile din uniune împart aceeaşi zonă de memorie. Aceasta înseamnă că numai valoarea unuia din câmpuri poate fi memorată la un moment dat în zona de memorie rezervată variabilei uniune.

Exemplu:

```
union h
 {
   int i;
   float t;
 }x;
```

Datele din variabila x vor fi privite ca întregi dacă selectăm x.i sau reale dacă selectăm x.t.
Câmpurile i şi t se referă la aceeaşi adresă:
în x.i se memorează sizeof(int) octeţi la această adresă;
în x.t se memorează sizeof(float) octeţi care încep la această adresă.

3.19 Recursivitatea în limbajul C++

Spunem despre o funcţie C că este recursivă dacă se autoapelează. Funcţia recursivă se poate reapela fie direct, fie indirect prin apelul altor funcţii.

Funcţiile recursive se definesc prin punerea în evidenţă a două seturi de instrucţiuni şi anume:

• un set care descrie modul în care funcţionează funcţia pentru anumite valori (iniţiale) ale unora dintre argumente;
• un set care descrie procesul recursiv de calcul.

Valorile unei funcţii recursive se calculează din aproape în aproape, pe baza valorilor cunoscute ale funcţiei pentru anumite argumente iniţiale. Pentru a calcula noile valori ale unei funcţii recursive, trebuie memorate valorile deja calculate, care sunt strict necesare. Acest fapt face ca implementarea în program a calculului unor funcţii recursive să necesite un consum mai mare de memorie, rezultând timpi mai mari de execuţie.

Recursivitatea poate fi transformată în iteraţie. În general, forma iterativă a unei funcţii este mai eficientă decât cea recursivă în ceea ce priveşte timpul de execuţie şi memoria consumată.

În alegerea căii (interativă sau recursivă) de rezolvare a unei probleme, trebuie consideraţi o serie de factori: uşurinţa programării, testării şi întreţinerii programului, eficienţa, complexitatea etc.

Dacă o problemă are o complexitate redusă este preferată varianta iterativă.

Forma recursivă este preferată acolo unde transformarea recursivităţii în iteraţie cere un efort de programare deosebit, algoritmul pierzându-şi claritatea, testarea şi întreţinerea devenind astfel foarte dificile.

La fiecare apel al funcţiei recursive, parametrii şi variabilele ei locale automatice se alocă pe stivă într-o zonă nouă, independentă. De asemenea în stivă se trece adresa de revenire în subprogramul chemător, adică adresa instrucţiunii următoare apelului. La revenire, se realizează curăţarea stivei, adică zona de pe stivă afectată la apel parametrilor şi variabilelor automatice, se eliberează.

Observaţii:

În general, recursivitatea permite o scriere mai compactă şi mai clară a programelor care conţin procese de calcul recursiv.

De obicei, recursivitatea nu conduce nici la economie de memorie şi nici la execuţia mai rapidă a programelor. În mod frecvent sunt variante nerecursive mai rapide decât variantele recursive şi conduc adesea şi la economie de memorie.

Apelurile recursive pot conduce la depăşirea stivei.

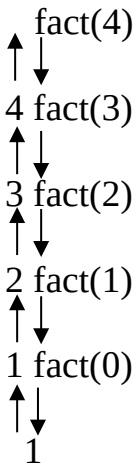
Pobleme rezolvate

1. Funcţia factorial: $fact : N \rightarrow N,\ fact(n) = \begin{cases} 1 & dacă\ n = 0, \\ n \cdot fact(n-1) & dacă\ n \geq 1 \end{cases}$

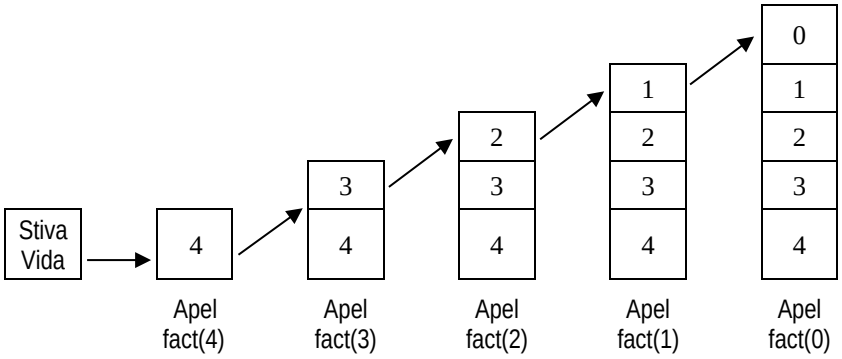
Varianta recursivă:

```
#include <iostream>
using namespace std;

long fact(int n)
 { if(n==0) return 1;
   return n*fact(n-1);
 }

int main()
 { int n;
   cout<<"n="; cin>>n;
   cout<<n<<"!="<<fact(n);
   return 0;
 }
```

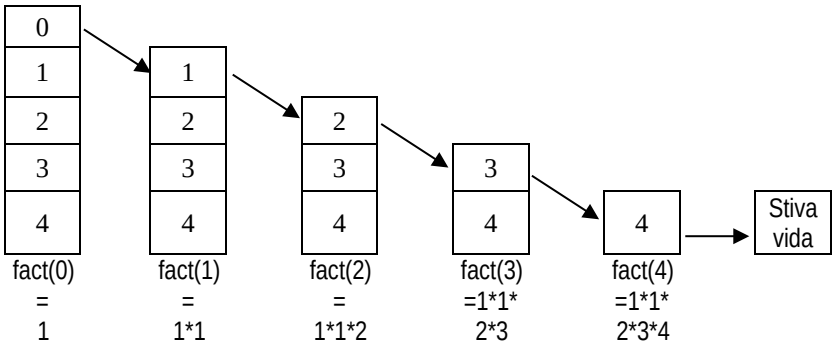
Apelul lui fact(4) declanşează un lanţ de apeluri ale lui fact pentru 3, 2, 1,0 după care urmează revenirea din apeluri şi evaluarea lui fact pentru 0,1,2,3,4.



Starea stivei in timpul execuţiei succesive a autoapelării:



Pentru fiecare din aceste apeluri, în stivă se vor depune parametrii actuali: 4,3,2,1,0. Stările stivei după ieşirea din autoapel sunt următoarele:

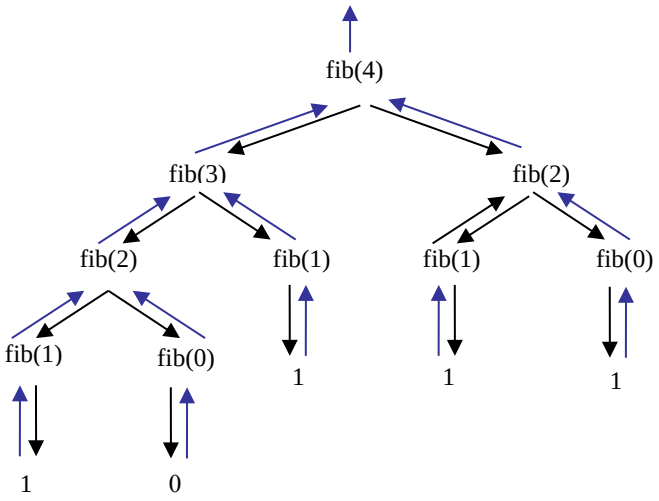


Rezolvarea fiecărui autoapel înseamnă deplasarea pe un nivel inferior.

Varianta iterativă:

```
#include <iostream>
using namespace std;

long fact(int n)
 { long P=1;
   for(int i=2;i<=n;i++) P=P*i;
   return P;
 }

int main()
 { int n;
   cout<<"n="; cin>>n;
   cout<<n<<"!="<<fact(n);
   return 0;
 }
```

2. *Funcţia lui Fibonacci:* $fib : N \to N, \quad fib(n) = \begin{cases} n, & n = 0 \text{ sau } n = 1, \\ fib(n-1) + fib(n-2), & n \geq 2 \end{cases}$



Varianta recursivă:

```
#include <iostream>
#include <iomanip>
using namespace std;

int n;
long fib(int n)
{
  if (n<=1) return n;
  else return fib(n-1)+fib(n-2);
}

int main()
{
 cout<<"n=";cin>>n;
 cout<<"fib(n)=fib("<<n<<")="<<fib(n)<<endl;
 return 0;
}
```

Varianta recursivă, folosind un vector *F* pentru memorarea valorilor *fib*(*i*) deja calculate (***tehnica memoizării***):

```
#include <iostream>
#include <iomanip>
using namespace std;

int n;
long F[100];

long fib(int n)   //memoizare
{
  if (n<=1) return n;
  else
    if (F[n]!=0) return F[n];
    else
    {  F[n]=fib(n-1)+fib(n-2);
       return F[n];
    }
}

int main()
{
 cout<<"n=";cin>>n;
 cout<<"fib(n)=fib("<<n<<")="<<fib(n)<<endl;
 return 0;
}
```

Varianta iterativă, folosind un vector *F* pentru memorarea valorilor *fib*(*i*), *i*=0,1,...,*n*:

```
#include <iostream>
#include <iomanip>
using namespace std;

int n;
long F[100];

void fib(long F[100],int n)
{ int k;
  F[0]=0; F[1]=1;
  for(k=2;k<=n;k++) F[k]=F[k-1]+F[k-2];
}

int main()
{
 cout<<"n=";cin>>n;
 fib(F,n);
 cout<<"F(n)=F("<<n<<")="<<F[n]<<endl;
 return 0;
}
```

Varianta iterativă, folosind doar trei variabile *a*, *b* și *c* pentru memorarea valorilor *fib*(*i*-2), *fib*(*i*-1), respectiv *fib*(*i*), pentru valorile succesive *i*=2,3,...,*n*:

```
#include <iostream>
#include <iomanip>
using namespace std;
```

```
int n;

long fib(int n)
{ long a,b,c;
  int i;
  if (n<=1) c=n;
  else
  {  a=0; b=1; //a=F(i-2), b=F(i-1), c=F(i) pentru i=2,3,...,n
     for(i=2;i<=n;i++)
     {   c=b+a; a=b; b=c;
     }
  }
  return c;
}

int main()
{
 cout<<"n=";cin>>n;
 cout<<"fib(n)=fib("<<n<<")="<<fib(n)<<endl;
 return 0;
}
```

3.20 Prelucrarea fişierelor în C++

Pentru a realiza operaţii de I/E cu fişiere, trebuie să includem în program fişierul `fstream.h`. În acest fişier sunt definite clasele `ifstream, ofstream,` şi `fstream.` Aceste clase sunt derivate din clasele *istream* şi *ostream* derivate la rândul lor din clasa *ios* clase definite în fişierul `iostream.h.` Deci declaraţiile claselor *ios*, *istream* şi *ostream* rămân valabile şi pentru lucru cu fişiere.

În C++, un fişier este deschis prin cuplarea lui la un stream.

Există trei tipuri de streamuri:

- de intrare;
- de ieşire;
- de intrare/ieşire.

Pentru a crea un stream de intrare, el trebuie declarat de tip `ifstream,` iar un stream de ieşire trebuie declarat de tip `ofstream.` Streamurile care realizează ambele tipuri de operaţii vor fi declarate de tip `fstream.`

Constructorii impliciţi ai claselor ifstream, ofstream şi fstream iniţializează streamuri fără a deschide fişiere.

Funcţia open este folosită pentru a asocia un fişier la un stream (după crearea streamului). Această funcţie este membră a tuturor celor trei clase de tip stream şi are prototipul

```
void open( char* numeFişier, int modDeschidere,
           int modProtectie);
```

Definiţii pentru modDeschidere:

Mod deschidere	Valoare	Comentariu
ios::in	0x01	Deschidere pentru citire. Fişierul trebuie să existe. Valoare implicită pentru *ifstream*
ios::out	0x02	Deschidere pentru scriere. Dacă fişierul nu există, se crează iar dacă există, conţinutul său se pierde.
ios::ate	0x04	Deschidere şi poziţionare la sfârşit de fişier.

Mod deschidere	Valoare	Comentariu
ios::app	0x08	Deschidere pentru scriere la sfârşit de fişier. Fişierul trebuie să existe.
ios::trunc	0x10	Deschidere şi trunchiere fişier (la lungime 0), dacă există.
ios::nocreate	0x20	Fişierul trebuie să existe la deschidere, altfel se produce eroare.
ios::noreplace	0x40	Fişierul trebuie să fie nou la deschidere, altfel deschiderea eşuează.
ios::binary	0x80	Opusul lui "text" (implicit): nu se traduc perechile "cr/lf".

Observaţie. Pentru activarea mai multor biţi se poate folosi operatorul "sau" binar `(|)`.

Argumentul *modProtectie* poate lua următoarele valori:

0 – fişier normal, fără restricţii de acces (valoare implicită);
1 – fişier de tip "read-only";
2 – fişier ascuns;
4 – fişier de tip sistem;
8 – bit de arhivare activ.

Se pot specifica mai multe dintre atribute folosind operatorul "sau" binar (|).

Exemple:

```
ifstream f;
f.open("f1.dat");
ofstream g;
g.open("f2.dat");
fstream h;
h.open("f3.dat",ios::in|ios::out);
    /* pentru a deschide un fişier "fstream", pentru operaţii de intrare/ieşire
          trebuie să specificăm atât ios::in cât şi ios::out. */
```

Clasele *ifstream, ofstream, fstream* conţin funcţii constructor care efectuează în mod automat operaţia de deschidere a fişierului. Ele au aceiaşi parametri şi valori implicite ca şi funcţia open:

Deci putem folosi direct:

```
ifstream f.open("f1.dat");
ofstream g.open("f2.dat");
fstream h("f3.dat",ios::in|ios::out);
```

Dacă operaţia de deschidere eşuează streamul va conţine valoarea zero, ceea ce înseamnă că putem testa uşor dacă operaţia de deschidere a reuşit. Destructorul clasei ifstream (respectiv ofstream sau fstream) goleşte tamponul fişierului şi închide fişierul (dacă nu este deja închis).

Închiderea unui fişier se face cu ajutorul funcţiei membre:

void close(void);

Funcţia membru `int eof()` întoarce o valoare diferită de zero dacă s-a atins sfârşitul fişierului şi 0 în caz contrar.

Scrierea şi citirea într-un, respectiv dintr-un fişier text deschis se poate face utilizând operatorii stream "<<" şi ">>" în mod similar cu cei utilizaţi la operaţiile de I/E utilizând consola cu excepţia faptului că, în loc să utilizăm dispozitivele standard cin, cout etc, folosim streamul cuplat la fişier. Toate informaţiile sunt memorate în fişier în acelaşi format ca şi cel utilizat la afişare.

Când efectuăm operaţii de I/E cu fişiere de tip text, caracterele newline se transformă în combinaţia de caractere `cr/lf`.

Funcţii de I/E de tip binar

Deşi fişierele text sunt utile în foarte multe aplicaţii, ele nu au flexibilitatea fişierelor de tip binar; din acest motiv, limbajul C++ asigură numeroase funcţii de I/E de tip binar: get, put, read, write etc.

Funcţia `get` are mai multe forme şi anume:

```
int get();
```

Extrage din streamul asociat un caracter şi returnează codul caracterului extras.

```
  istream& get(char& c);
```

Citeşte din streamul asociat un caracter şi îl memorează în variabila de ieşire `c.` Funcţia înapoiază o referinţă la streamul asociat care va avea valoarea zero dacă s-a detectat sfârşitul de fişier.

```
  istream& get(char *p,int n, char delimitator='\n');
```

Citeşte n caractere din stream sau până la întâlnirea delimitatorului dat ca al treilea argument şi îi memorează în zona pointată de `p.` Valoarea implicită a delimitatorului este '\n', şi nu este extras din stream.

```
  istream& getline(char *p,int n, char delimitator='\n');
```

Este similară cu funcţia precedentă, dar extrage, eventual, şi delimitatorul.

Funcţia put are prototipul:

```
 ostream& put (char c);
```

Scrie în streamul asociat caracterul dat ca argument.

Funcţia

```
int peek();
```

ne furnizează următorul caracter din streamul de intrare fără să-l extragă din stream.

Funcţia

```
 istream& putback(char c)
```

reinserează un caracter în stream.

Pentru a citi/scrie blocuri de date binare, se utilizează funcţiile read/write, care au următoarele prototipuri:

```
     istream& read(unsigned char * p, int n);
     ostream& write(const unsigned char *p, int n);
```

Funcţia `read` citeşte `n` octeţi din streamul asociat şi îi plasează în zona de adresă `p`. Dacă se detectează sfârşitul fişierului înainte de citirea tuturor octeţilor, funcţia read opreşte citirea.

Pentru a afla numărul caracterelor citite utilizăm funcţia:

```
    int gcount ();
```

Ea înapoiază numărul de caractere citite la ultima operaţie de intrare.

Accesul de tip aleator la fişiere binare

Sistemul de intrare/ieşire din C++ utilizează doi pointeri pentru accesul la fişiere. Primul pointer numit "get pointer" indică poziţia din fişier de unde se va efectua citirea, iar al doilea numit "put pointer" indică poziţia de la care se va face următoarea scriere. Ori de câte ori se va efectua o operaţie de intrare/ieşire pointerul corespunzător va fi avansat (în mod automat) secvenţial.

Accesul aleator la fişiere se poate efectua prin utilizarea funcţiilor `seekg()` şi `seekp().`

Funcţia

```
    ostream& seekg(long n);
```

poziţionează "get pointerul" pe octetul n în fişier (numerotarea se face de la 0).

Funcţia

```
    ostream& seekg(long n, seek_dir origine);
```

deplasează "get pointerul" asociat fişierului respectiv cu n octeţi faţă de origine.
`seek_dir` este un tip enumerare cu valorile ios::beg=0, ios::cur=1, ios::end=2 semnificând începutul, poziţia curentă, respectiv sfârşitul streamului.

Funcţiile

```
ostream& seekp(long n);
ostream& seekp(long n, seek_dir origine);
```

sunt similare cu seekg dar se aplică pointerului "put pointer".

Exemplu: *Sortările crescătoare și descrescătoare pentru un șir de numere.* Șirul nesortat se citește dintr-un fișier, iar șirurile sortate se scriu (afișează) tot în fișiere.

```
#include<iostream>   //SORTARI VECTOR UTILIZAND FISIERE
#include<fstream>
using namespace std;

void citire(char *nume,int x[20], int &nx)
{ ifstream f(nume);
  int a;
  nx=0;
  f>>a;
  while(!f.eof())
  { x[++nx]=a;
    f>>a;
  }
  f.close();
}

void afisare(char *nume,int x[20],int nx)
{ int i;
  ofstream f(nume);
  for(i=1;i<=nx;i++) f<<x[i]<<" ";
  f.close();
}

void sort_selectie(int x[20], int nx) //sortare crescatoare
{ int i,j,min,jmin,aux;
  for(i=1;i<nx;i++)
  {  min=x[i];jmin=i;
     for(j=i+1;j<=nx;j++)
       if(x[j]<min)
       { min=x[j]; jmin=j;
       }
     aux=x[i];x[i]=x[jmin];x[jmin]=aux;
  }
}

void sort_interschimbare(int x[20], int nx) //sortare descrescatoare
{ int i,j,aux;
  for(i=nx;i>1;i--)
     for(j=1;j<i;j++)
       if(x[j]<x[j+1])
       { aux=x[j];x[j]=x[j+1];x[j+1]=aux;
       }
}

int main(void)
{ int n,v[20];
  citire("sort_fis.dat",v,n);
```

```
  sort_selectie(v,n);
  afisare("sort_cr.dat",v,n);
  citire("sort_fis.dat",v,n);
  sort_interschimbare(v,n);
  afisare("sort_dcr.dat",v,n);
  return 0;
}
```