

MANUAL C/C++

CUPRINS

PARTEA I

CAPITOLUL 1 Noțiuni introductive

1. Structura generală a unui sistem de calcul
2. Algoritmi
- 2.1. Noțiuni generale
- 2.2. Definiții și caracteristici
- 2.3. Reprezentarea algoritmilor
- 2.3.1. Reprezentarea prin scheme logice
- 2.3.2. Reprezentarea prin pseudocod
3. Teoria rezolvării problemelor
- Întrebări și exerciții

CAPITOLUL 2 Date, operatori și expresii

1. Limbajele C și C++
2. Programe în limbajul C/C++
3. Preprocesorul
4. Elemente de bază ale limbajului
- 4.1. Vocabularul
- 4.2. Unitățile lexicale
5. Date în limbajul C/C++
- 5.1. Tipuri de date
- 5.2. Constante
- 5.2.1. Constante întregi
- 5.2.2. Constante numerice, reale
- 5.2.3. Constante caracter
- 5.2.4. Constante șir de caractere
- 5.3. Variabile
- 5.3.1. Declararea variabilelor
- 5.3.2. Inițializarea variabilelor în declarații
- 5.3.3. Operații de intrare/ieșire
6. Operatori și expresii
- 6.1. Operatori
- 6.2. Expresii
- 6.3. Conversii de tip
- Întrebări și exerciții

CAPITOLUL 3 Implementarea structurilor de control

1. Implementarea structurii secvențiale ...

2. Implementarea structurii de decizie ...
3. Implementarea structurilor repetitive (ciclice)
- 3.1. Implementarea structurilor ciclice cu test inițial
- 3.2. Implementarea structurilor ciclice cu test final
4. Facilități de întrerupere a unei secvențe
- Întrebări și exerciții

CAPITOLUL 4 Tablouri

1. Declararea tablourilor
2. Tablouri unidimensionale
3. Tablouri bidimensionale
4. Șiruri de caractere
- Întrebări și exerciții

CAPITOLUL 5 Pointeri

1. Variabile pointer
- 1.1. Declararea variabilelor pointer ...
- 1.2. Inițializarea variabilelor pointer ...
- 1.3. Pointeri generici
2. Operații cu pointeri
3. Pointeri și tablouri
- 3.1. Pointeri și șiruri de caractere ...
- 3.2. Pointeri și tablouri bidimensionale
4. Tablouri de pointeri
5. Pointeri la pointeri
6. Modificatorul const în declararea pointerilor
- Întrebări și exerciții

CAPITOLUL 6 Funcții

1. Structura unei funcții
2. Apelul și prototipul unei funcții
3. Transferul parametrilor unei funcții ...
- 3.1. Transferul parametrilor prin valoare
- 3.2. Transferul prin pointeri
- 3.3. Transferul prin referință

3.4. Transferul parametrilor către funcția main	
4. Tablouri ca parametri	
5. Funcții cu parametri implicați	
6. Funcții cu număr variabil de parametri ..	
7. Funcții predefinite	
7.1. Funcții matematice	
7.2. Funcții de clasificare (testare) a caracterelor	
7.3. Funcții de conversie a caracterelor	
7.4. Funcții de conversie din șir în număr	
7.5. Funcții de terminare a unui proces (program)	
7.6. Funcții de intrare/ieșire	
8. Clase de memorare	
9. Moduri de alocare a memoriei	
10. Funcții recursive	
11. Pointeri către funcții	
Întrebări și exerciții	

CAPITOLUL 7 Tipuri de date definite de utilizator

1. Tipuri definite de utilizator	
2. Structuri	
3. Câmpuri de biți	
4. Declarații de tip	
5. Unioni	
6. Enumerări	
Întrebări și exerciții	

CAPITOLUL 8 Fișiere

1. Caracteristicile generale ale fișierelor. . .	
2. Deschiderea unui fișier	
3. Închiderea unui fișier	
4. Prelucrarea fișierelor text	
4.1. Prelucrarea la nivel de caracter . .	
4.2. Prelucrarea la nivel de cuvânt. . .	
4.3. Prelucrarea la nivel de șir de caractere	
4.4. Intrări/ieșiri formate	
5. Intrări/ieșiri binare	
6. Poziționarea într-un fișier	
7. Funcții utilitare pentru lucrul cu fișiere. .	
8. Alte operații cu fișiere	
Întrebări și exerciții	

PARTEA a II a

CAPITOLUL 9 Concepte de bază ale programării orientate obiect

1. Introducere	
2. Abstractizarea datelor	
3. Moștenirea	
3.1. Moștenirea unică	
3.1. Moștenirea multiplă	
4. Încapsularea informației	
5. Legarea dinamică (târzie)	
6. Alte aspecte	

CAPITOLUL 10 Clase și obiecte

1. Definiția claselor și accesul la Membri	
1.1. Legătura clasă-structură-Uniune	
1.2. Declararea claselor	
1.3. Obiecte	
1.4. Membrii unei clase	
1.5. Pointerul <i>this</i>	
1.6. Domeniul unui nume, vizibilitate și timp de viață . . .	
2. Funcții <i>inline</i>	
3. Constructori și destructori	
3.1. Inițializarea datelor	
3.2. Constructori	
3.2.1. Constructori cu liste de inițializare. .	
3.2.2. Constructori de copiere	
3.3. Destructori	
3.4. Tablouri de obiecte	
4. Funcții prietene.	
Întrebări și exerciții	

CAPITOLUL 11 Supraîncărcarea operatorilor

1. Moduri de supraîncărcare a operatorilor	
1.1. Supraîncărcarea prin funcții membre	
1.2. Supraîncărcarea prin funcții prietene	
2. Restricții la supraîncărcarea operatorilor	
3. Supraîncărcarea operatorilor unari . . .	
4. Membrii constanți ai unei clase	

5. Supraîncărcarea operatorilor insertor și extractor	
6. Supraîncărcarea operatorului de atribuire =	
7. Supraîncărcarea operatorului de indexare []	
8. Supraîncărcarea operatorilor <i>new</i> și <i>delete</i>	
9. Supraîncărcarea operatorului ()	
10. Supraîncărcarea operatorului ->	
11. Conversii	
11.1. Conversii din tip predefinit1 în tip predefinit2.	
11.2. Conversii din tip predefinit în clasă	
11.3. Conversii din clasă în tip predefinit	
11.4. Conversii din clasă1 în clasă2	
Întrebări și exerciții	

CAPITOLUL 12 Crearea ierarhiilor de clase

1. Mecanismul moștenirii	
2. Modul de declarare a claselor derivate.	
3. Constructorii claselor derivate	

4. Moștenirea simplă	
5. Moștenirea multiplă	
6. Redefinirea membrilor unei clase de bază în clasa derivată	
7. Metode virtuale	
Întrebări și exerciții	

CAPITOLUL 13 Intrări/ieșiri

1. Principiile de bază ale sistemului de I/O din limbajul C++	
2. Testarea și modificarea stării unui flux	
3. Formatarea unui flux	
3.1. Formatarea prin manipulatori	
3.2. Formatarea prin metode	
4. Metodele clasei <i>istream</i>	
5. Metodele clasei <i>ostream</i>	
6. Manipulatori creați de utilizator	
7. Fluxuri pentru fișiere	
8. Fișiere binare	
Întrebări și exerciții	

1. Structura generală a unui sistem de calcul
2. Algoritmi
2.1. Noțiuni generale

2.2. Definiții și caracteristici
2.3. Reprezentarea algorimilor
3. Teoria rezolvării problemelor

1. Structura generală a unui sistem de calcul

Calculatorul reprezintă un sistem electronic (ansamblu de dispozitive și circuite diverse) complex care prelucrează datele introduse într-o formă prestabilită, efectuează diverse operații asupra acestora și furnizează rezultatele obținute (figura 1).

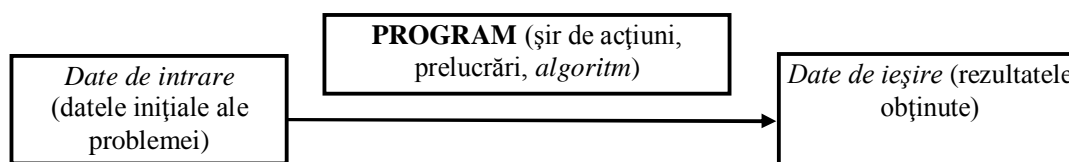


Figura 1. Calculatorul - sistem automat de prelucrare a datelor.

Principalele avantaje ale folosirii calculatorului constau în:

- viteza mare de efectuare a operațiilor;
- capacitatea extinsă de prelucrare și memorare a informației.

Deși construcția unui calculator - determinată de tehnologia existentă la un moment dat, de domeniul de aplicație, de costul echipamentului și de performanțele cerute - a evoluat rapid în ultimii ani, sistemele de calcul, indiferent de model, serie sau generație, au o serie de *caracteristici comune*. Cunoașterea acestor caracteristici ușurează procesul de înțelegere și învățare a modului de funcționare și de utilizare a calculatorului.

În orice sistem de calcul se vor găsi două părți distincte și la fel de importante: *hardware*-ul și *software*-ul.

- Hardware-ul este reprezentat de totalitatea echipamentelor și dispozitivelor fizice;
- Software-ul este reprezentat prin totalitatea programelor care ajută utilizatorul în rezolvarea problemelor sale (figura 2).

Software-ul are două componente principale:

- *Sistemul de operare* (de exploatare) care coordonează întreaga activitate a echipamentului de calcul. Sistemul de operare intră în funcțiune la pornirea calculatorului și asigură, în principal, trei funcții:
 - Gestiunea echitabilă și eficientă a resurselor din cadrul sistemului de calcul;
 - Realizarea interfeței cu utilizatorul;
 - Furnizarea suportului pentru dezvoltarea și execuția aplicațiilor.

Exemple de sisteme de operare: RSX11, CP/M, MS-DOS, LINUX, WINDOWS NT, UNIX.

- *Sistemul de aplicații* (de programare): medii de programare, editoare de texte, compilatoare, programe aplicative din diverse domenii (economic, științific, financiar, divertisment).

Componentele unui sistem de calcul pot fi grupate în unități cu funcții complexe, dar bine precizate, numite *unități funcționale*. Modelul din figura 3 face o prezentare simplificată a structurii unui calculator, facilitând înțelegerea unor noțiuni și concepte de bază privind funcționarea și utilizarea acestuia. Denumirea fiecărei unități indică funcția ei, iar săgețile - modul de transfer al informației.

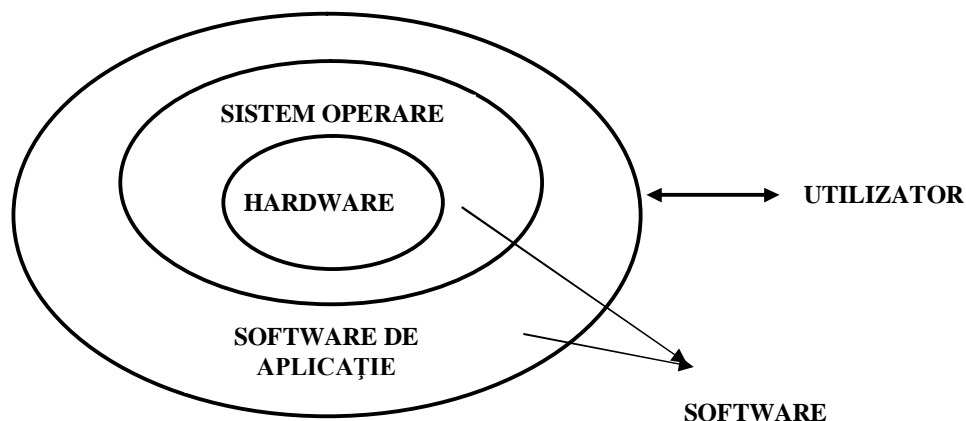


Figura 2. Echipamentul de calcul ca un sistem hardware-software.

Se vor utiliza în continuare termenii de *citire* pentru operația de introducere (de intrare) de la tastatură a datelor inițiale ale unei probleme, și *scriere* pentru operația de afișare (de ieșire) a rezultatelor obținute. În cazul în care utilizatorul dorește să rezolve o problemă cu ajutorul calculatorului, informația de intrare (furnizată calculatorului de către utilizator) va consta din datele inițiale ale problemei de rezolvat și dintr-un program (numit program sursă). În programul sursă utilizatorul implementează (traduce) într-un limbaj de programare un *algoritm* (acțiunile executate asupra datelor de intrare pentru a obține rezultatele). Această informație de intrare este prezentată într-o *formă externă*, accesibilă omului (numere, text, grafică) și va fi transformată de către calculator într-o *formă internă*, binară.

Unitatea de intrare (cu funcția de citire) realizează această conversie a informației din format extern în cel intern. Din punct de vedere logic, fluxul (informația) de intrare este un șir de caractere, din exterior către memoria calculatorului. Din punct de vedere fizic, unitatea de intrare standard este tastatura calculatorului. Tot ca unități de intrare, pot fi enumerate: mouse-ul, joystick-ul, scanner-ul (pentru introducerea informațiilor grafice).

Unitatea de ieșire (cu funcția de scriere, afișare) realizează conversia inversă, din formatul intern în cel extern, accesibil omului. Din punct de vedere fizic, unitatea de ieșire standard este monitorul calculatorului. Ca unități de ieșire într-un sistem de calcul, mai putem enumera: imprimanta, plotter-ul, etc.

Informația este înregistrată în *memorie*.

Memoria internă (memoria **RAM - Random Acces Memory**) se prezintă ca o

succesiune de octeți (octet sau **byte** sau locație de memorie). Un octet are 8 **biți**. **Bit**-ul reprezintă unitatea elementară de informație și poate avea una din valorile: 0 sau 1.

Capacitatea unei memorii este dată de numărul de locații pe care aceasta le conține și se măsoară în multiplii de 1024 (2^{10}). De exemplu, 1 Mbyte=1024Kbytes; 1Kbyte=1024bytes.

Numărul de ordine al unui octet în memorie se poate specifica printr-un cod, numit *adresă*. Ordinea în care sunt adresate locațiile de memorie nu este impusă, memoria fiind un dispozitiv cu acces aleator la informație.

În memorie se înregistrează două categorii de informații:

- *Date* - informații de prelucrat;
- *Programe* - conțin descrierea (implementarea într-un limbaj de programare) a acțiunilor care vor fi executate asupra datelor, în vederea prelucrării acestora.

În memoria internă este păstrată doar informația prelucrată la un moment dat. Memoria internă are capacitate redusă; accesul la informația pastrată în aceasta este extrem de rapid, iar datele nu sunt păstrate după terminarea prelucrării (au un caracter temporar).

Unitatea centrală prelucrează datele din memoria internă și coordonează activitatea tuturor componentelor fizice ale unui sistem de calcul. Ea înglobează:

- **Microprocesorul**- circuit integrat complex cu următoarele componente de bază:
 - Unitatea de execuție (realizează operații logice și matematice);
 - Unitatea de interfață a magistralei (transferă datele la/de la microprocesor).
- **Coprocessorul matematic** – circuit integrat destinat realizării cu viteză sporită a operațiilor cu numere reale.

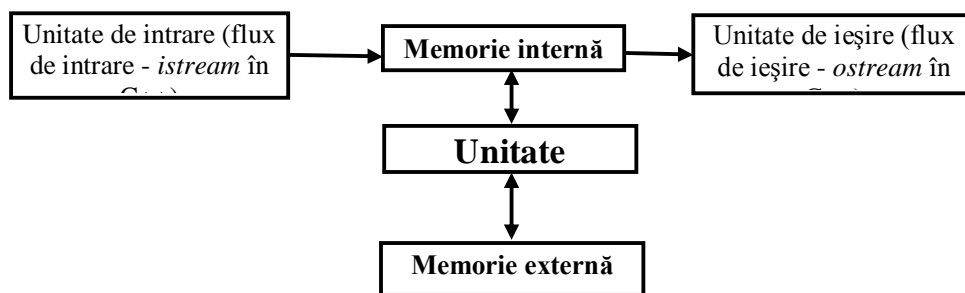


Figura 3. Unitățile funcționale ale unui sistem de calcul.

În funcție de numărul de biți transferați simultan pe magistrala de date, microprocesoarele pot fi clasificate astfel: microprocesoare pe 8 biți (Z80, 8080); microprocesoare pe 16 biți (8086, 8088, 80286) cu coprocesoarele corespunzătoare (8087, 80287); familii de procesoare pe 32 biți (80386DX, 80486, PENTIUM) cu coprocesoarele corespunzătoare (începând de la 486, coprocesoare sunt încorporate microprocesoarelor).

Memoria externă este reprezentată, fizic, prin unitățile de discuri (discuri dure-*hard disk*, discuri flexibile-*floppy disk*, discuri de pe care informația poate fi doar citită-CDROM, DVDROM, etc). Spre deosebire de memoria internă, memoria externă are

capacitate mult mai mare, datele înregistrate au caracter permanent, în dezavantajul timpului de acces la informație.

2. Algoritmi

2.1. Noțiuni generale. Algoritmul este *conceptul fundamental* al informaticii. Orice echipament de calcul poate fi considerat o mașină algoritmică. Într-o *definiție aproximativă* algoritmul este un set de pași care definește modul în care poate fi dusă la îndeplinire o anumită sarcină. Exemplu de algoritm: algoritmul de interpretare a unei bucăți muzicale (descrie în partitură). Pentru ca o mașină de calcul să poată rezolva o anumită problemă, programatorul trebuie mai întâi să stabilească un algoritm care să conducă la efectuarea la sarcinii respective.

Exemplu: Algoritmul lui Euclid pentru determinarea celui mai mare divizor comun (cmmdc) a 2 numere întregi pozitive.

Date de intrare: cele 2 numere întregi

Date de ieșire: cmmdc

1. Se notează cu A și B - cea mai mare, respectiv cea mai mică, dintre datele de intrare;
2. Se împarte A la B și se notează cu R restul împărțirii;
3. a. Dacă R diferit de 0, se atribuie lui A valoarea lui B și lui B valoarea lui R. Se revine la pasul 2.
b. Dacă R este 0, atunci cmmdc este B.

Probleme legate de algoritmi. Descoperirea unui algoritm care să rezolve o problemă echivalează în esență cu descoperirea unei soluții a problemei. După descoperirea algoritmului, pasul următor este ca algoritmul respectiv să fie reprezentat într-o formă în care să poată fi comunicat unei mașini de calcul. Algoritmul trebuie transcris din forma conceptuală într-un set clar de instrucțiuni. Aceste instrucțiuni trebuie reprezentate într-un mod lipsit de ambiguitate. În acest domeniu, studiile se bazează pe cunoștințele privitoare la gramatică și limbaj și au dus la o mare varietate de scheme de reprezentare a algoritmilor (numite limbaje de programare), bazate pe diverse abordări ale procesului de programare (numite paradigme de programare).

Căutarea unor algoritmi pentru rezolvarea unor probleme din ce în ce mai complexe a avut ca urmare apariția unor întrebări legate de limitele proceselor algoritmice, cum ar fi:

- Ce probleme pot fi rezolvate prin intermediul proceselor algoritmice?
- Cum trebuie procedat pentru descoperirea algoritmilor?
- Cum pot fi îmbunătățite tehnicile de reprezentare și comunicare a algoritmilor?
- Cum pot fi aplicate cunoștințele dobândite în vederea obținerii unor mașini algoritmice mai performante?
- Cum pot fi analizate și comparate caracteristicile diversilor algoritmi?

2.2. Definiții și caracteristici.

Definiții: Algoritmul unei prelucrări constă într-o secvență de primitive care descrie prelucrarea. Algoritmul este un set ordonat de pași executabili, descriși fără echivoc, care definesc un proces finit.

Proprietățile fundamentale ale algoritmilor:

- *Caracterul finit:* orice algoritm bine proiectat se termină într-un număr finit de pași;
- *Caracterul unic și universal:* orice algoritm trebuie să rezolve toate problemele dintr-o clasă de probleme;
- *Realizabilitatea:* orice algoritm trebuie să poată fi codificat într-un limbaj de programare;
- *Caracterul discret:* fiecare acțiune se execută la un moment dat de timp;
- *Caracterul determinist:* ordinea acțiunilor în execuție este determinată în mod unic de rezultatele obținute la fiecare moment de timp.

Nerespectarea acestor caracteristici generale conduce la obținerea de algoritmi neperformanți, posibil infiniți sau nerealizabili.

2.3. Reprezentarea algoritmilor. Reprezentarea (descrierea) unui algoritm nu se poate face în absența unui limbaj comun celor care vor să îl înțeleagă. De aceea s-a stabilit o mulțime bine definită de *primitive* (blocuri elementare care stau la baza reprezentării algoritmilor). Fiecare primitivă se caracterizează prin *sintaxă* și *semantică*. Sintaxa se referă la reprezentarea simbolică a primitivei; semantica se referă la semnificația primitivei. Exemplu de primitivă: aer-din punct de vedere sintactic este un cuvânt format din trei simboluri (litere); din punct de vedere semantic este o substanță gazoasă care înconjoară globul pământesc.

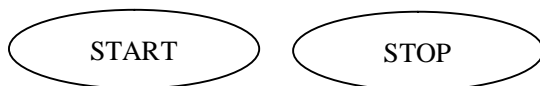
Algoritmii se reprezintă prin:

- scheme logice;
- pseudocod.

2.3.1. Reprezentarea algoritmilor prin scheme logice. Primitivele utilizate în schemele logice sunt simboluri grafice, cu funcțiuni (reprezentând procese de calcul) bine precizate. Aceste simboluri sunt unite prin arce orientate care indică ordinea de execuție a proceselor de calcul.

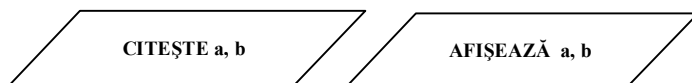
Categorii de simboluri:

- Simboluri de *început* și *sfârșit*



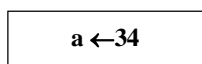
Simbolul START desemnează începutul unui program sau al unui subprogram.
Simbolul STOP desemnează sfârșitul unui program sau al unui subprogram. Prezența lor este obligatorie.

- Simbolul *paralelogram*



Semnifică procese (operații) de intrare/ieșire (citirea sau scrierea)

- Simbolul *dreptunghi*



Semnifică o atribuire (modificarea valorii unei date).

□ Simbolul romb

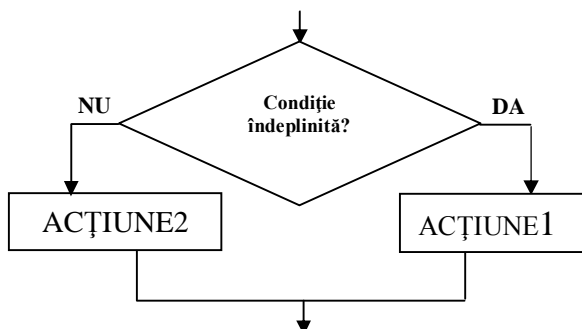


Figura 4. Structura de decizie.

Simbolul romb este utilizat pentru decizii (figura 4). Se testează îndeplinirea condiției din blocul de decizie. **Dacă** această condiție este îndeplinită, se execută ACȚIUNE1. Dacă nu, se execută ACȚIUNE2. La un moment dat, se execută **sau** ACȚIUNE1, **sau** ACȚIUNE2.

Cu ajutorul acestor simboluri grafice se poate reprezenta orice algoritm.

Repetarea unei secvențe se realizează prin combinarea simbolurilor de decizie și de atribuire.

Structurile repetitive obținute pot fi: cu test inițial sau cu test final.

Structuri repetitive cu test inițial

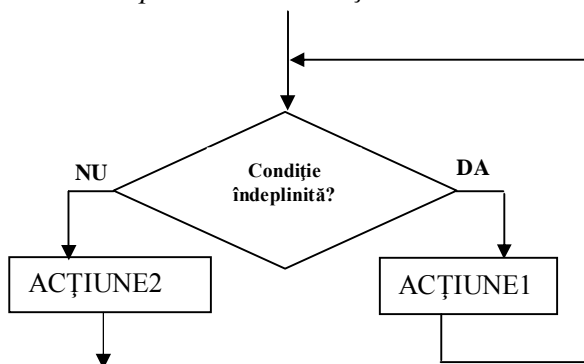


Figura 5. Structură repetitivă cu test inițial.

Se evaluează condiția de test (figura 5). Dacă aceasta este îndeplinită, se execută ACȚIUNE1. Se revine apoi și se testează iar condiția. Dacă este îndeplinită, se execută (se repetă) ACȚIUNE1, ș.a.m.d. Abia în momentul în care condiția nu mai este îndeplinită, se trece la execuția ACȚIUNE2.

Astfel, **cât timp** condiția este îndeplinită, se repetă ACȚIUNE1. În cazul în care, la prima testare a condiției, aceasta nu este îndeplinită, se execută ACȚIUNE2. Astfel, este posibil ca ACȚIUNE1 **să nu fie executată** niciodată.

Exisă și situații în care se știe de la început de câte ori se va repeta o anumită acțiune. În aceste cazuri se folosește tot o structură de control repetitivă cu test inițial. Se utilizează un contor (numeric) pentru a ține o evidență a numărului de execuții ale acțiunii. De câte ori se execută acțiunea, contorul este incrementat.

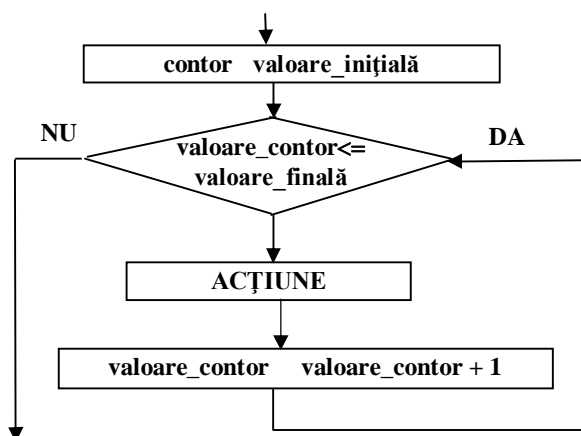


Figura 6. Structură repetitivă cu test inițial, cu număr cunoscut de pași.

Se atribuie contorului valoarea inițială (figura 6). **Cât timp condiția** (valoarea contorului este mai mică sau egală cu valoarea finală) este îndeplinită, **se repetă:** ACȚIUNE incrementare contor (se adună 1 la valoarea anterioară a contorului).

Structură repetitivă cu test final:

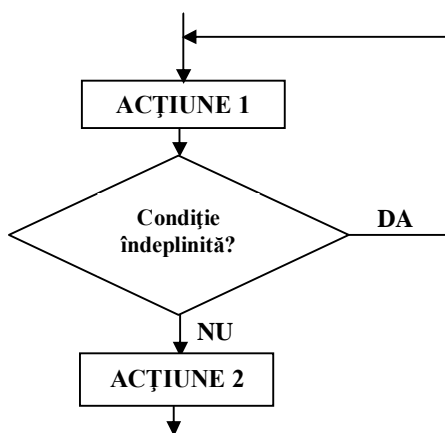


Figura 7. Structură repetitivă cu test final.

Se execută mai întâi ACȚIUNE1. Se testează apoi condiția (figura 7). Se **repetă** ACȚIUNE1 **cât timp** condiția este îndeplinită. În acest caz, corpul ciclului (ACȚIUNE1) este executat cel puțin o dată.

2.3.2. Reprezentarea algoritmilor prin pseudocod. Pseudocodul este inspirat din limbajele de programare, nefiind însă atât de formalizat ca acestea. Pseudocodul reprezintă o punte de legătură între limbajul natural și limbajele de programare. Nu există un standard pentru regulile lexicale. Limbajul pseudocod permite comunicarea între oameni, și nu comunicarea om-mașina (precum limbajele de programare). Pseudocodul utilizează cuvinte cheie (scrise cu majuscule subliniate) cu următoarele semnificații:

Sfârșit algoritm:	<u>SFÂRȘIT</u>	
Început algoritm:	<u>ÎNCEPUT</u>	
Citire (introducere) date:	<u>CITEȘTE</u>	lista
Scriere (afișare) date:	<u>SCRIE</u>	lista
Atribuire:	<u><-</u>	
Structura de decizie (alternativă):	<u>DACĂ</u>	condiție
	<u>ATUNCI</u>	acțiune1

Structuri repetitive cu test inițial: ALTFEL acțiune2
 CÂT TIMP condiție
 REPETĂ acțiune
PENTRU contor=val_iniț LA val_fin [PAS]
REPETĂ acțiune;

Structuri repetitive cu test final:

REPETĂ acțiune CÂT TIMP condiție
sau:
 REPETĂ acțiune PÂNĂ CÂND condiție

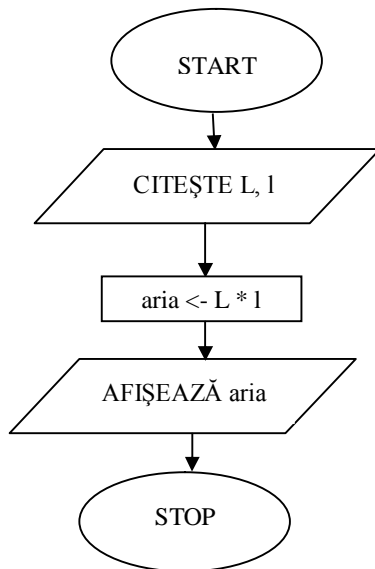
Pe lângă cuvintele cheie, în reprezentarea algoritmilor în pseudocod pot apare și propoziții nestandard a caror detaliere va fi realizată ulterior.

În cazul în care se realizează un algoritm modularizat, pot apare cuvintele cheie:

SUBALGORITM nume (lista_intrări)
CHEAMĂ nume (lista_valori_efective_de_intrare)

Exemple: Se vor reprezenta în continuare algoritmii de rezolvare pentru câteva probleme simple (pentru primele 2 probleme se va exemplifica și modul de implementare a acestor algoritmi în limbajul C++).

1. Se citesc 2 valori numerice reale, care reprezintă dimensiunile (lungimea și lățimea unui dreptunghi). Să se calculeze și să se afișeze aria dreptunghiului.



ALGORITM aflare_arie_drept
 INCEPUT
 CITEȘTE L, l
 aria <- L * l
 AFIȘEAZA aria
 SFARȘIT

Implementare:

```

#include <iostream.h>
void main( )
{ double L, l;
  cout<<"Lungime="; cin>>L;
  cout<<"Lațime="; cin>>l;
  double aria = L * l;
  cout << "Aria="<< aria;
}
  
```

2. Se citesc 2 valori reale. Să se afișeze valoarea maximului dintre cele 2 numere.

```
ALGORITM max_2_nr
INCEPUT
    CITEȘTE a, b
    DACA a >= b
        ATUNCI max<-a
    ALTFEL max<-b
    AFISEAZA max

#include <iostream.h>
void main( )
{ float a, b, max;
  cout<<"a="; cin>>a;
  cout<<"b="; cin>>b;
  if (a >= b)
      max = a;
  else max = b;
  cout<<"Maximul este:"<<max;}
```

Sau:

```
ALGORITM max_2_nr
ÎNCEPUT
    CITEȘTE a, b
    DACA a >= b
        ATUNCI AFISEAZA a
    ALTFEL AFISEAZA b
    SFARȘIT

#include <iostream.h>
void main( )
{ float a, b;
  cout<<"a="; cin>>a;
  cout<<"b="; cin>>b;
  if (a >= b)
      cout<<"Maximul este:"<<a;
  else
      cout<<"Maximul este:"<<b; }
```

3. Să se citească câte 2 numere întregi, până la întâlnirea perechii de numere 0, 0. Pentru fiecare pereche de numere citite, să se afișeze maximul. Algoritm care utilizează structură repetitivă cu test inițial:

```
ALGORITM max_perechi1
INCEPUT
    CITEȘTE a,b
    CAT TIMP(a#0sau b#0)REPETA
        INCEPUT
            DACA (a>=b)
                ATUNCI AFISEAZA a
            ALTFEL AFISEAZA b
        CITEȘTE a,b
    SFARSIT
SFARSIT
```

```
ALGORITM max_perechi2
INCEPUT
    a ← 3
    CAT TIMP (a#0 sau b#0) REPETA
        INCEPUT
            CITEȘTE a, b
            DACA (a>=b)
                ATUNCI AFISEAZA a
            ALTFEL AFISEAZA b
        SFARSIT
    SFARSIT
```

Algoritm care utilizează structură repetitivă cu test final:

```
ALGORITM max_perechi3
INCEPUT
    REPETA
        INCEPUT
            CITEȘTE a,b
            DACA (a>=b)
                ATUNCI AFIȘEAZA a
            ALTFEL AFIȘEAZA b
        SFARȘIT
    CAT TIMP (a#0 sau b#0)
    SFARȘIT
```

3. Teoria rezolvării problemelor

Creșterea complexității problemelor supuse rezolvării automate (cu ajutorul calculatorului) a determinat ca activitatea de programare să devină, de fapt, un complex de activități.

Pentru *rezolvarea unei probleme* trebuie parcurse următoarele *etape*:

- Analiza problemei (înțelegerea problemei și specificarea cerințelor acesteia). Se stabilește *ce* trebuie să facă aplicația, și *nu cum*. Se stabilesc datele de intrare (identificarea mediului inițial) și se stabilesc obiectivele (identificarea mediului final, a rezultatelor);
- Proiectarea (conceperea unei metode de rezolvare a problemei printr-o metodă algoritmică);
- Implementarea (codificarea algoritmului ales într-un limbaj de programare);
- Testarea aplicației obținute (verificarea corectitudinii programului);
- Exploatarea și întreținerea (mentenanța, activitatea de modificare a aplicației la cererea beneficiarului sau în urma unor deficiențe constatate pe parcursul utilizării aplicației).

În acest context, activitatea de programare a devenit o activitate organizată, definindu-se metode formale de dezvoltare a fiecărei etape. Etapele descrise anterior alcătuiesc *ciclul de viață al unui produs software* și constituie obiectul de studiu al disciplinei numite *ingineria sistemelor de programe (software engineering)*.

Teoreticienii ingineriei programării consideră că rezolvarea unei probleme se poate face pe 3 direcții:

- Rezolvarea *orientată pe algoritm* (pe acțiune), în care organizarea datelor este neesențială;
- Rezolvarea *orientată pe date*, acțiunile fiind determinate doar de organizarea datelor;
- Rezolvarea *orientată obiect*, care combină tendințele primelor două abordări.

Abordarea aleasă determină modelarea problemei de rezolvat.

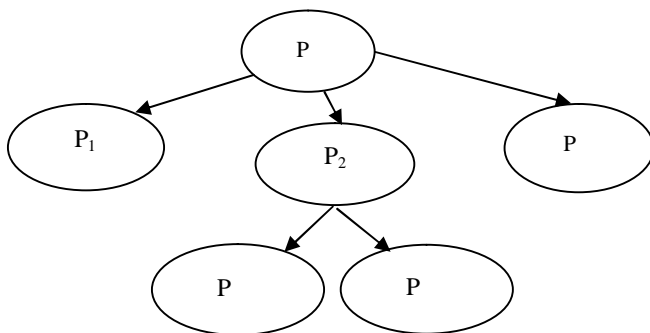
Dintre *metodele de proiectare orientate pe algoritm* amintim: *metoda programării structurate* și *metoda rafinării succesive*. Ambele au ca punct de plecare *metoda de proiectare top-down*, considerată ca fiind o metodă clasică de formalizare a procesului de dezvoltare a unui produs software.

La baza metodei top-down stă *descompunerea funcțională* a problemei P , adică găsirea unui număr de subprobleme P_1, P_2, \dots, P_n , cu următoarele proprietăți:

- Fiecare subproblemă P_i ($1 \leq i \leq n$) poate fi rezolvată independent. Dacă nu constituie o problemă elementară, poate fi, la randul ei, descompusă;
- Fiecare subproblemă P_i este mai simplă decât problema P ;
- Soluția problemei P se obține prin reuniunea soluțiilor subproblemelor P_i ;
- Procesul de descompunere se oprește în momentul în care toate subproblemele P_i obținute sunt elementare, deci pot fi implementate.

Comunicarea între aceste subprobleme se realizează prin intermediul parametrilor. Implementarea metodei top-down într-un limbaj de programare se face cu ajutorul modulelor de program (funcții sau proceduri în limbajul Pascal, funcții în

limbajul C).



Descompunerea funcțională a unui program P constă în identificarea funcțiilor (task-urilor, sarcinilor) principale ale programului (P , P , P), fiecare dintre aceste funcții reprezentând un subprogram (figura 8). Problemele de pe același nivel i sunt independente unele față de altele.

Figura 8. Descompunerea funcțională.

📖 Etapele rezolvării unei probleme cu ajutorul calculatorului. Se detaliază în continuare *etapa de implementare*. După analiza problemei și stabilirea algoritmului, acesta trebuie tradus (implementat) într-un limbaj de programare.

- **Scrierea (editarea) programului sursă.** Programele sursă sunt fișiere text care conțin instrucțiuni (cu sintactica și semantica proprii limbajului utilizat). Programul (fișierul) sursă este creat cu ajutorul unui *editor de texte* și va fi salvat pe disc (programele sursă C primesc, de obicei, extensia *.c*, iar cele C++, extensia *.cpp*). Pentru a putea fi executat, programul sursă trebuie *compilat* și *linkeditat*.
- **Compilarea.** Procesul de compilare este realizat cu ajutorul compilatorului, care translatează codul sursă în cod obiect (cod mașină), pentru ca programul să poată fi înțeles de calculator. În cazul limbajului C, în prima fază a compilării este invocat *preprocesorul*. Acesta recunoaște și analizează mai întâi o serie de instrucțiuni speciale, numite *directive procesor*. Verifică apoi codul sursă pentru a constata dacă acesta respectă sintaxa și semantica limbajului. Dacă există erori, acestea sunt semnalate utilizatorului. Utilizatorul trebuie să corecteze erorile (modificând programul sursă). Abia apoi codul sursă este translatat în cod de asamblare, iar în final, în cod mașină, binar, propriu calculatorului. Acest cod binar este numit cod obiect și de obicei este memorat într-un alt fișier, numit *fișier obiect*. Fișierul obiect va avea, de obicei, același nume cu fișierul sursă și extensia *.obj*.
- **Linkeditarea.** După ce programul sursă a fost translatat în program obiect, el este va fi supus operației de linkeditare. Scopul fazei de linkeditare este acela de a obține o formă finală a programului, în vederea execuției acestuia. Linkeditorul “leagă” modulele obiect, rezolvă referințele către funcțiile externe și rutinele din biblioteci și produce cod executabil, memorat într-un alt fișier, numit *fișier executabil* (același nume, extensia *.exe*).
- **Execuția.** Lansarea în execuție constă în încărcarea programului executabil în memorie și startarea execuției sale.

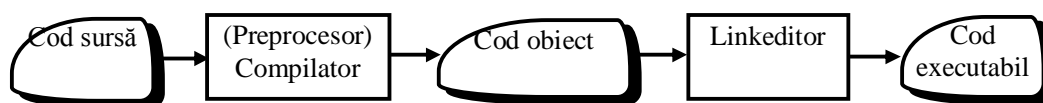


Figura 9. Etapele necesare obținerii fișierului executabil.

Observații:

1. Mediile de programare integrate (BORLANDC, TURBOC) înglobează editorul, compilatorul, linkeditorul și depanatorul (utilizat în situațiile în care apar erori la execuție);
2. Dacă nu se utilizează un mediu integrat, programatorul va apela în mod explicit (în linie de comandă) un editor de texte, compilatorul, linkeditorul. Lansarea în execuție se va face tot din linie de comandă.
3. Extensiile specificate pentru fișierele sursă, obiect și executabile sunt

ÎNTREBĂRI ȘI EXERCITII

Întrebări teoretice. Enumerați unitățile funcționale componente ale unui sistem de calcul.

1. Care sunt diferențele între soft-ul de aplicație și sistemul de operare?
2. Care este deosebirea între algoritm și program?
3. Care sunt proprietățile fundamentale ale algoritmilor?
4. Care sunt modalitățile de reprezentare a algoritmilor?

Exerciții practice.

1. Reprezentați algoritmul lui Euclid (pentru calculul celui mai mare divizor comun a 2 numere întregi) prin schema logică.
2. Proiectați un algoritm care să rezolve o ecuație de gradul I (de forma $ax + b = 0$), unde a, b sunt numere reale. Discuție după coeficienți.
3. Proiectați un algoritm care să rezolve o ecuație de gradul II (de forma $ax^2 + bx + c = 0$), unde a, b, c sunt numere reale. Discuție după coeficienți.
4. Proiectați un algoritm care să testeze dacă un număr întreg dat este număr prim.
5. Proiectați un algoritm care să afișeze toți divizorii unui număr întreg introdus de la tastatură.
6. Proiectați un algoritm care să afișeze toți divizorii primi ai unui număr întreg introdus de la tastatură.
7. Proiectați un algoritm care calculează factorialul unui număr natural dat. (Prin definiție $0!=1$)

1. Limbajele C și C++	5.1. Tipuri de date
2. Programe în limbajul C/C++	5.2. Constante
3. Preprocesorul	5.3. Variabile
4. Elemente de bază ale limbajului	6. Operatori și expresii
4.1. Vocabularul	6.1. Operatori
4.2. Unitățile lexicale	6.2. Expresii
5. Date în limbajul C/C++	7. Conversii de tip

1. Limbajele C și C++

Așa cum comunicarea dintre două persoane se realizează prin intermediul limbajului natural, comunicarea dintre om și calculator este mijlocită de un limbaj de programare. Limbajele C și C++ sunt limbaje de programare de nivel înalt.

Limbajul C a apărut în anii 1970 și a fost creat de Dennis Ritchie în laboratoarele AT&T Bell. Limbajul C face parte din familia de limbaje concepute pe principiile programării structurate, la care ideea centrală este ”structurează pentru a stăpâni o aplicație”. Popularitatea limbajului a crescut rapid datorită eleganței și a multiplelor posibilități oferite programatorului (puterea și flexibilitatea unui limbaj de asamblare); ca urmare, au apărut numeroase alte implementări. De aceea, în anii '80 se impune necesitatea standardizării acestui limbaj. În perioada 1983-1990, un comitet desemnat de ANSI (American National Standards Institute) a elaborat un compilator ANSI C, care permite scrierea unor programe care pot fi portate fără modificări, pe orice sistem.

Limbajul C++ apare la începutul anilor '80 și îl are ca autor pe Bjarne Stroustrup. El este o variantă de limbaj C îmbunătățit, mai riguroasă și mai puternică, completată cu construcțiile necesare aplicării principiilor programării orientate pe obiecte (POO). Limbajul C++ păstrează toate elementele limbajului C, beneficiind de eficiența și flexibilitatea acestuia. Limbajul C++ este un superset al limbajului C. Incompatibilitățile sunt minore, de aceea, modulele C pot fi încorporate în proiecte C++ cu un efort minim.

2. Programe în limbajul C/C++

Un *program* scris în limbajul C (sau C++) este compus din unul sau mai multe *fișiere sursă*. Un fișier sursă este un fișier text care conține codul sursă (în limbajul C) al unui program. Fiecare fișier sursă conține una sau mai multe *funcții* și eventual, referințe către unul sau mai multe *fișiere header* (figura 1).

Funcția principală a unui program este numită *main*. Execuția programului începe cu execuția acestei funcții, care poate apela, la rândul ei, alte funcții. Toate funcțiile folosite în program trebuie descrise în fișierele sursă (cele scrise de către programator), în fișiere header (funcțiile predefinite, existente în limbaj), sau în biblioteci de funcții.

Un fișier header este un fișier aflat în sistem sau creat de către programator, care conține declarații și definiții de funcții și variabile.

Acțiunile din fiecare funcție sunt codificate prin *instrucțiuni* (figura 2.a.). Există mai multe tipuri de instrucțiuni, care vor fi discutate în **capitolul** următor. O instrucțiune este orice expresie validă (de obicei, o asignare sau un apel de funcție), urmată de simbolul;. În figura 2.b. este dat un exemplu de instrucțiune simplă. Uneori, ca instrucțiune poate apare instrucțiunea nulă (doar;), sau instrucțiunea compusă (privită ca o succesiune de instrucțiuni simple, încadrate între acoladele delimitatoare { }).

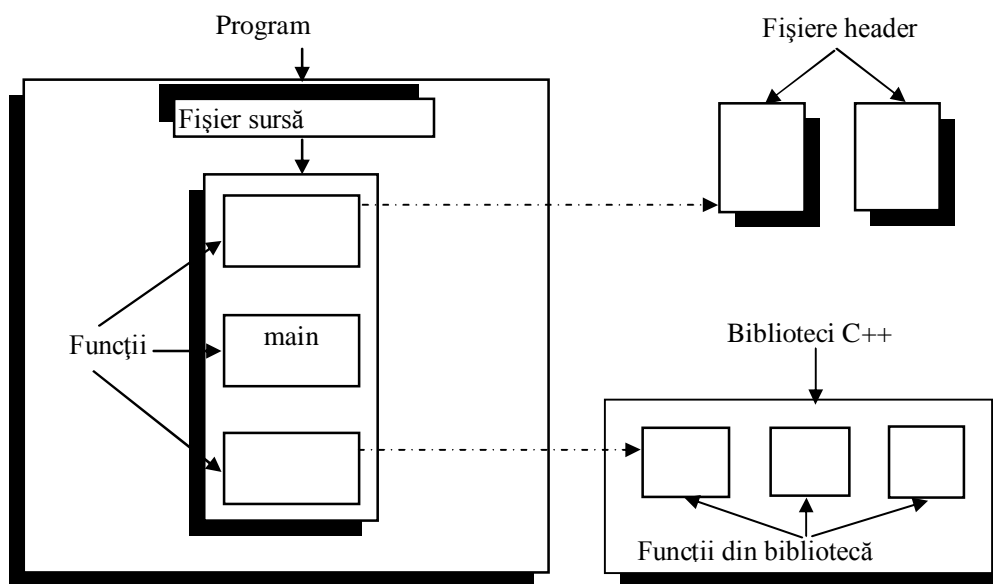


Figura 1. Structura unui program în limbajul C.

O expresie este o structură corectă sintactic, formată din operanzi și operatori (figura 2.c.).

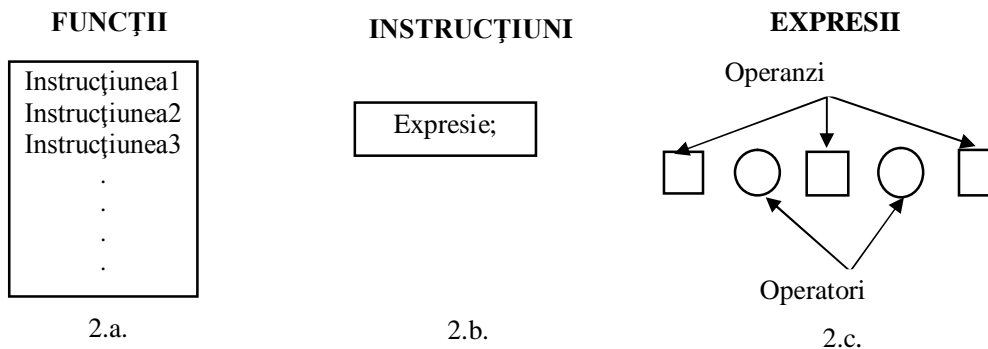


Figura 2. Funcție, instrucțiune, expresie.

Pentru a înțelege mai bine noțiunile prezentate, să considerăm un exemplu foarte simplu. Programul următor afișează pe ecran un mesaj (mesajul *Primul meu program*). Informația de prelucrat (de intrare) este însuși mesajul (o constantă șir), iar prelucrarea ei constă în afișarea pe ecran.

Exemplu:

```
#include <iostream.h> // linia 1
void main()           // linia 2 - antetul funcției main
{                     /* linia 3 - începutul corpului funcției, a unei instrucțiuni
                      compuse */
cout<<"Primul meu program în limbajul C++\n"; // linia 5
}                     // linia 6-sfârșitul corpului funcției
```

Prima linie este o *directivă preprocesor* (indicată de simbolul #) care determină includerea în fișierul sursă a fișierului header cu numele `iostream.h`. Acest header permite realizarea afișării pe monitor.

Programul conține o singură funcție, *funcția principală*, numită `main`, al cărui *antet* (linia 2) indică:

- tipul valorii returnate de funcție (`void`, ceea ce înseamnă că funcția nu returnează nici o valoare);
- numele funcției (`main`);
- lista argumentelor primite de funcție, încadrată de cele 2 paranteze rotunde.

Funcțiile comunică între ele prin argumente. Aceste argumente reprezintă datele de intrare ale funcției. În cazul nostru, nu avem nici un argument în acea listă, deci puteam să scriem antetul funcției și astfel:

```
void main(void)
```

Ceea ce urmează după simbolul `//`, până la sfârșitul liniei, este un *comentariu*, care va fi ignorat de către compilator. Comentariul poate conține un text explicativ; informații lămuritoare la anumite aspecte ale problemei sau observații. Dacă vrem să folosim un comentariu care cuprinde mai multe linii, vom delimita începutul acestuia indicat prin simbolul `/*`, iar sfârșitul - prin `*/` (vezi liniile 3, 4). Introducerea comentariilor în programele sursă ușurează înțelegerea acestora. În general, se recomandă introducerea unor comentarii după antetul unei funcții, pentru a preciza prelucrările efectuate în funcție, anumite limite impuse datelor de intrare, etc.

Începutul și sfârșitul corpului funcției `main` sunt indicate de cele două acoale

{ (linia3) și }(linia 6). Corpul funcției (linia 5) este format dintr-o singură instrucțiune, care implementează o operație de scriere. Cuvantul `cout` este un cuvânt predefinit al limbajului C++ - console **output** - care desemnează dispozitivul logic de ieșire; simbolul `<<` este operatorul de transfer a informației. Folosite astfel, se deschide un canal de comunicație a datelor către dispozitivul de ieșire, în cazul acesta, monitorul. După operator se specifică informațiile care vor fi afișate (în acest exemplu, un șir de caractere constant). Faptul că este un șir constant de caractere este indicat de ghilimelele care îl încadrează. Pe ecran va fi afișat fiecare caracter din acest șir, cu excepția grupului `\n`. Deși grupul este format din două caractere, acesta va fi interpretat ca un singur caracter - numit *caracter escape* - care determină poziționarea cursorului la începutul următoarei linii. O secvență `escape` (cum este `\n`) furnizează un mecanism general și extensibil pentru reprezentarea caracterelor invizibile sau greu de obținut. La sfârșitul instrucțiunii care implementează operația de scriere, apare `;`.

3. Preprocesorul

Așa cum s-a menționat în **capitolul 1**, în faza de compilare a fișierului sursă este invocat întâi preprocesorul. Acesta tratează directivele speciale - numite *directive preprocesor* - pe care le găsește în fișierul sursă. Directivele preprocesor sunt identificate prin simbolul `#`, care trebuie să fie primul caracter, diferit de spațiu, dintr-o linie. Directivele preprocesor sunt utilizate la includerea fișierelor header, la definirea numelor constantelor simbolice, la definirea macro-urilor, sau la realizarea altor funcții (de exemplu, compilarea condiționată), așa cum ilustrează exemplele următoare:

➤ **Includerea fișierelor header în codul sursă:**

Exemplul 1:

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

Când procesorul întâlnește această linie, datorită simbolului `#`, o recunoaște ca fiind o directivă preprocesor, localizează fișierul header indicat (parantezele unghiulare `<>` indică faptul că este vorba de un *fișier header sistem*).

Exemplul 2:

```
#include "headerul_meu.h"
```

Numele fișierului header inclus între ghilimele, indică faptul că `headerul_meu.h` este un fișier header creat de utilizator. Preprocesorul va căuta să localizeze acest fișier în directorul curent de lucru al utilizatorului. În cazul în care fișierul header nu se află în directorul curent, se va indica și calea către acesta.

Exemplul 3:

```
#include "c:\\bc\\head\\headerul_meu.h"
```

În acest exemplu, pentru interpretarea corectă a caracterului backslash `\`, a fost necesară "dublarea" acestuia, din motive pe care le vom prezenta în alt paragraf.

➤ **Asignarea de nume simbolice constantelor:**

Exemplu:

```
#define TRUE    1
#define FALSE   0
```

Tratarea acestor directive preprocesor are ca efect asignarea (atribuirea) valorii întregi 1 numelui (constantei simbolice) `TRUE`, și a valorii 0 numelui simbolic `FALSE`. Ca urmare, înaintea compilării propriu-zise, în programul sursă, aparițiile numelor

TRUE și FALSE vor fi înlocuite cu valorile 1, respectiv 0.

- **Macrodefiniții:** Directiva `#define` este folosită și în macrodefiniții. Macrodefinițiile permit folosirea unor nume simbolice pentru expresiile indicate în directivă.

Exemplu:

```
#define NEGATIV(x)  -(x)
```

Între numele macrodefiniției și paranteza stângă (`NEGATIV(...)`) nu sunt permise spații albe. La întâlnirea în programul sursă a macrodefiniției `NEGATIV`, preprocesorul substituie argumentul acesteia cu expresia (negativarea argumentului). Macrodefiniția din exemplu poate fi folosită în programul sursă astfel: `NEGATIV(a+b)`. Când preprocesorul întâlnește numele expresiei, substituie literalii din paranteză, `a+b`, cu argumentul din macrodefiniție, `x`, obținându-se `-(a+b)`.

Dacă macrodefiniția ar fi fost de forma:

```
#define NEGATIV(x)  -x
```

`NEGATIV(a+b)` ar fi fost tratată ca `-a+b`.

4. Elemente de bază ale limbajului

4.1. Vocabularul. În scrierea programelor în limbajul C/C++ pot fi folosite doar anumite simboluri care alcătuiesc *alfabetul limbajului*. Acesta cuprinde:

- Literele mari sau mici de la A la Z (a-z);
- Caracterul subliniere (`_` underscore), folosit, de obicei, ca element de legătura între cuvintele compuse;
- Cifrele zecimale (0-9);
- Simboluri speciale:
- Caractere:
 - operatori (Exemple: `+`, `*`, `!=`);
 - delimitatori (Exemple: blank (spațiu), `\t`, `\n`, cu rolul de a separa cuvintele);
- Grupuri (perechi de caractere).

Grupurile de caractere, numite adesea *separatori*, pot fi:

- () - Încadrează lista de argumente ale unei funcții sau sunt folosite în expresii pentru schimbarea ordinii de efectuare a operațiilor (în ultimul caz, fiind operator);
- { } - Încadrează instrucțiunile compuse;
- // - Indică începutul unui comentariu care se poate întinde până la sfârșitul liniei;
- /* */ - Indică începutul și sfârșitul unui comentariu care poate cuprinde mai multe linii;
- " " - Încadrează o constantă șir (un șir de caractere);
- ' ' - Încadrează o constantă caracter (un caracter imprimabil sau o secvență escape).

4.2. Unitățile lexicale. Unitățile lexicale (cuvintele) limbajului C/C++ reprezintă grupuri de caractere cu o semnificație de sine stătătoare. Acestea sunt:

- Identificatori;
- Cuvinte cheie ale limbajului.

Identificatorii reprezintă numele unor date (constante sau variabile), sau ale unor funcții. Identificatorul este format dintr-un șir de litere, cifre sau caracterul de subliniere (underscore), trebuie să înceapă cu o literă sau cu caracterul de subliniere și să fie sugestiv.

Exemple: viteză, greutate_netă, Viteza, Viteza1, GreutateNetă

Identificatorii pot conține litere mici sau mari, dar limbajul C++ este sensibil la majuscule și minuscule (case-sensitive). Astfel, identificatorii viteza și Viteza sunt diferiți.

Nu pot fi folosiți ca identificatori cuvintele cheie. Identificatorii pot fi standard (ca de exemplu numele unor funcții predefinite: scanf, clear, etc.) sau aleși de utilizator.

Cuvintele cheie sunt cuvinte ale limbajului, împrumutate din limba engleză, cărora programatorul nu le poate da o altă utilizare. Cuvintele cheie se scriu cu litere mici și pot reprezenta:

- Tipuri de date (Exemple: int, char, double);
- Clase de memorare (Exemple: extern, static, register);
- Instrucțiuni (Exemple: if, for, while);
- Operatori (Exemplu: sizeof).

Sensul cuvintelor cheie va fi explicat pe măsură ce vor fi prezentate construcțiile în care acestea apar.

5. Date în limbajul C/C++

Așa cum s-a văzut în **capitolul 1**, un program realizează o prelucrare de informație. Termenul de prelucrare trebuie să fie considerat într-un sens foarte general (de exemplu, în programul prezentat în **paragraful 2**, prelucrarea se referea la un text și consta în afișarea lui). În program datele apar fie sub forma unor *constante* (valori cunoscute anticipat, care nu se modifică), fie sub formă de *variabile*. Constantele și variabilele sunt obiectele informaționale de bază manipulate într-un program.

Fiecare categorie de date este caracterizată de atributele:

- Nume;
- Valoare;
- Tip;
- Clasa de memorare.

Numele unei date. Numele unei date este un identificator și, ca urmare, trebuie să respecte regulile specifice identificatorilor. De asemenea, numărul de caractere care intră în compunerea unui identificator este nelimitat, însă, implicit, numai primele 32 de caractere sunt luate în considerare. Aceasta înseamnă că doi identificatori care au primele 32 de caractere identice, diferențiindu-se prin caracterul 33, vor fi considerați identici.

5.1. Tipuri de date. *Tipul unei date* constă într-o mulțime de valori pentru care

s-a adoptat un anumit mod de reprezentare în memoria calculatorului și o *mulțime de operatori* care pot fi aplicați acestor valori. Tipul unei date determină *lungimea zonei de memorie* ocupată de acea dată. În general, lungimea zonei de memorare este dependentă de calculatorul pe care s-a implementat compilatorul. Tabelul 1 prezintă lungimea zonei de memorie ocupată de fiecare tip de dată pentru compilatoarele sub MS-DOS și UNIX/LINUX.

Tipurile de bază sunt:

- `char` - un singur octet (1 byte=8 biți), capabil să conțină codul unui caracter din setul local de caractere;
- `int` - număr întreg, reflectă în mod tipic mărimea naturală din calculatorul utilizat;
- `float` - număr real, în virgulă mobilă, simplă precizie;
- `double` - număr real, în virgulă mobilă, dublă precizie.

În completare există un număr de *calificatori*, care se pot aplica tipurilor de bază `char`, `int`, `float` sau `double`: `short`, `long`, `signed` și `unsigned`. Astfel, se obțin *tipurile derivate de date*. `Short` și `long` se referă la mărimea diferită a întregilor, iar datele de tip `unsigned int` sunt întotdeauna pozitive. S-a intenționat ca `short` și `long` să furnizeze diferite lungimi de întregi, `int` reflectând mărimea cea mai "naturală" pentru un anumit calculator. Fiecare compilator este liber să interpreteze `short` și `long` în mod adecvat propriului hardware; în nici un caz, însă, `short` nu este mai lung decât `long`. Toți acești calificatori pot aplicați tipului `int`. Calificatorii `signed` (cel implicit) și `unsigned` se aplică tipului `char`. Calificatorul `long` se aplică tipului `double`. Dacă într-o declarație se omite tipul de bază, implicit, acesta va fi `int`.

Tabelul 1.

Tip	Lungimea zonei de memorie ocupate (în biți)		Descriere
	MS-DOS	UNIX (LINUX)	
<code>char</code>	8	8	Valoarea unui singur caracter; poate fi întâlnit în expresii cu extensie de semn
<code>unsigned char</code>	8	8	Aceeași ca la <code>char</code> , fără extensie de semn
<code>signed char</code>	8	8	Aceeași ca la <code>char</code> , cu extensie de semn obligatorie
<code>int</code>	16	32	Valoare întreagă
<code>long</code> (<code>long int</code>)	32	64	Valoare întreagă cu precizie mare
<code>long long int</code>	32	64	Valoare întreagă cu precizie mare
<code>short int</code>	16	32	Valoare întreagă cu precizie mică
<code>unsigned int</code>	16	32	Valoare întreagă, fără semn
<code>unsigned long int</code>	32	64	Valoare întreagă, fără semn
<code>float</code>	32	32	Valoare numerică cu zecimale, simplă precizie (6)
<code>double</code>	64	64	Valoare numerică cu zecimale, dublă precizie (10)
<code>long double</code>	80	128	Valoare numerică cu zecimale, dublă precizie

Se consideră, de exmplu, tipul `int`, folosit pentru date întregi (pozitive sau negative). Evident că mulțimea valorilor pentru acest tip va fi, de fapt, o *submulțime finită* de numere întregi. Dacă pentru memorarea unei date de tip `int` se folosesc 2 octeți

de memorie, atunci valoarea maximă pentru aceasta va fi $\frac{1}{2} \times 2^{16} - 1$, deci $2^{15} - 1$ (32767), iar valoarea minimă va fi $-\frac{1}{2} \times 2^{16}$, deci -2^{15} (-32768). Încercarea de a calcula o expresie de tip `int` a cărei valoare se situează în afara acestui domeniu va conduce la o eroare de execuție.

Mulțimea valorilor pentru o dată de tip `unsigned int` (întreg fără semn) va fi formată din numerele întregi situate în intervalul $[0, 2^{16} - 1]$.

În header-ul **<values.h>** sunt definite constantele simbolice (cum ar fi: `MAXINT`, `MAXSHORT`, `MAXLONG`, `MINDOUBLE`, `MINFLOAT`, etc.) care au ca valoare limitele inferioară și superioară ale intervalului de valori pentru tipurile de date enumerate. (de exemplu `MAXINT` reprezintă valoarea întregului maxim care se poate memora, etc.)

Fără a detalia foarte mult modul de reprezentare a datelor reale (de tip `float` sau `double`), vom sublinia faptul că, pentru acestea, este importantă și *precizia de reprezentare*. Deoarece calculatorul poate reprezenta doar o submulțime finită de valori reale, în anumite cazuri, pot apărea erori importante.

Numerele reale pot fi scrise sub forma:

$$N = \text{mantisa} \times \text{baza}^{\text{exponent}}$$

unde: baza reprezintă baza sistemului de numerație; mantisa (coeficientul) este un număr fracționar normalizat (în fața virgulei se află 0, iar prima cifră de după virgulă este diferită de zero); exponentul este un număr întreg. Deoarece forma internă de reprezentare este binară, baza=2. În memorie vor fi reprezentate doar mantisa și exponentul. Numărul de cifre de după virgulă determină *precizia* de exprimare a numărului. Ce alte cuvinte, pe un calculator cu o precizie de 6 cifre semnificative, două valori reale care diferă la a 7-a cifră zecimală, vor avea aceeași reprezentare. Pentru datele de tip `float`, precizia de reprezentare este 6; pentru cele de tip `double`, precizia este 14, iar pentru cele de tip `long double`, precizia este 20.

Lungimea zonei de memorie ocupate de o dată de un anumit tip (pe câți octeți este memorată data) poate fi aflată cu ajutorul operatorului `sizeof`.

Exemplu:

```
cout<<"Un int este memorat pe "<<sizeof(int)<<"octeți.\n";
```

Instrucțiunea are ca efect afișarea pe monitor a mesajului: *Un int este memorat pe 2 octeți.*

5.2. Constante. O constantă este un *literal* (o formă externă de reprezentare) *numeric*, *caracter* sau *șir de caractere*. Numele și valoarea unei constante sunt identice. Valoarea unei constante nu poate fi schimbată în timpul execuției programului în care a fost utilizată. Tipul și valoarea ei sunt determinate în mod automat, de către compilator, pe baza caracterelor care compun literalul.

5.2.1. Constante întregi. Constantele întregi sunt literal numerici (compuși din cifre), fără punct zecimal.

- Constante întregi în baza 10, 8 sau 16
 - Constante întregi în baza 10

Exemple:

45

-78 // constante întregi decimale (în baza 10), tip **int**

- Constante *întregi octale*

Dacă în fața numărului apare cifra zero (0), acest lucru indică faptul că acea constantă este de tipul **int**, în baza opt (constantă octală).

Exemple:

056

077 // constante întregi octale, tip **int**

- Constante *întregi hexazecimale*

Dacă în fața numărului apar caracterele zero (0) și x (sau X), acest lucru indică faptul că acea constantă este de tipul **int**, în baza 16 (constantă hexagesimală). Amintim că în baza 16 cifrele sunt: 0-9, A (sau a) cu valoare 10, B (sau b) cu valoare 11, C (sau c) cu valoare 12, D (sau d) cu valoare 13, E (sau e) cu valoare 14, F (sau f) cu valoare 15.

Exemple:

0x45

0x3A

0Xbc // constante întregi hexagesimale, tip **int**

➤ Constante *întregi, de tipuri derivate*

Dacă secvența de cifre este urmată de L sau l, tipul constantei este **long int**.

Exemple:

145677L

897655l // tip decimal **long int**

Dacă secvența de cifre este urmată de U sau u, tipul constantei este **unsigned int**.

Exemple:

65555u

Dacă secvența de cifre este urmată de U (u) și L (l), tipul constantei este **unsigned long int**.

Exemple: 7899UL //tip decimal **unsigned long int**

5.2.2. Constante numerice, reale.

Dacă o constantă numerică conține punctul zecimal, ea este de tipul **double**.

Exemplu:

3.1459 //tip **double**

Dacă numărul este urmat de F sau f, constante este de tip **float**.

Dacă numărul este urmat de L sau l, este de tip **long double**.

Exemplu:

0.45f //tip **float**

9.788L //tip **long double**

- Constante reale în format științific. Numărul poate fi urmat de caracterul e sau E și de un număr întreg, cu sau fără semn. În acest caz, constanta este în *notație științifică*. În această formă externă de reprezentare, numărul din fața literei E reprezintă *mantisa*, iar numărul întreg care urmează caracterului E reprezintă *exponentul*. În forma externă de reprezentare, baza de numerație este 10, deci valoarea constantei va fi dată de $\text{mantisa} \times 10^{\text{exponent}}$.

Exemplu:

1.5e-2 //tip **double**, în notație științifică, valoare 1.5×10^{-2}

Exercițiu: Să se scrie următorul program și să se urmărească rezultatele execuției acestuia.

```
#include <iostream.h>
#include <values.h>
#define PI 3.14359
int main()
{
    cout<<"Tipul int memorat pe: "<<sizeof(int)<<" octeti\n";
    cout<<"Tipul int memorat pe: "<<sizeof(23)<<" octeti\n";
    //23-const. zecimala int
    cout<<"Int maxim="<<MAXINT<<'\n';
    //const. simbolice MAXINT, MAXLONG, etc. - definite in <values.h>
    cout<<"Const. octala 077 are val decimale:"<<077<<'\n';
    cout<<"Const. hexagesimala d3 are val decimale:"<<0xd3<<'\n';
    cout<<"Tipul unsigned int memorat pe:"<<sizeof(unsigned int)<<" octeti\n";
    cout<<"Tipul unsigned int memorat pe: "<<sizeof(23U)<<" octeti\n";
    cout<<"Tipul unsigned int memorat pe: "<<sizeof(23u)<<" octeti\n";
    cout<<"Tipul long int memorat pe: "<<sizeof(long int)<<" octeti\n";
    cout<<"Tipul long int memorat pe: "<<sizeof(23L)<<" octeti\n";
    cout<<"Tipul long int memorat pe: "<<sizeof(23l)<<" octeti\n";
    //23L sau 23l-const. decimale long int
    cout<<"Long int maxim="<<MAXLONG<<'\n';
    cout<<"Tipul unsigned long memorat pe:";
    cout<<sizeof(unsigned long int)<<" octeti\n";
    cout<<"Tipul unsigned long memorat pe: "<<sizeof(23UL)<<" octeti\n";
    cout<<"Tipul unsigned long memorat pe: "<<sizeof(23ul)<<" octeti\n";
    //23UL sau 23ul-const. decimale unsigned long int
    cout<<"Tipul long long int memorat pe: ";
    cout<<sizeof(long long int)<<" octeti\n";
    cout<<"Tipul long long int memorat pe: "<<sizeof(d)<<" octeti\n";
    cout<<"Tipul short int memorat pe: "<<sizeof(short int)<<" octeti\n";
    cout<<"Short int maxim="<<MAXSHORT<<'\n';
    cout<<"Tipul float memorat pe: "<<sizeof(float)<<" octeti\n";
    cout<<"Tipul float memorat pe: "<<sizeof(23.7f)<<" octeti\n";
    //23.7f-const. decimale float
    cout<<"Float maxim="<<MAXFLOAT<<'\n';
    cout<<"Float minim="<<MINFLOAT<<'\n';
    cout<<"Tipul double memorat pe: "<<sizeof(double)<<" octeti\n";
    cout<<"Tipul double memorat pe: "<<sizeof(23.7)<<" octeti\n";
    //23.7-const. decimale double
    cout<<"Const. decim. dubla in notatie stiintifica:"<<23.7e-5<<'\n';
    cout<<"Const. PI este:"<<PI<<'\n';
    cout<<"Constanta PI este memorata pe:"<<sizeof(PI)<<"octeti\n";
    cout<<"Double                                maxim="<<MAXDOUBLE<<'\n'<<"Double
    minim="<<MINDOUBLE<<'\n';
    cout<<"Tipul long double memorat pe: "<<sizeof(long double)<<" octeti\n";
    cout<<"Tipul long double memorat pe: "<<sizeof(23.7L)<<" octeti\n";
    //23.7L-const. decimale long double
    cout<<"Cifra A din HEXA are val.:"<<0xA<<"\n";
    cout<<"Cifra B din HEXA are val.:"<<0xB<<"\n";
    cout<<"Cifra C din HEXA are val.:"<<0xC<<"\n";
    cout<<"Cifra D din HEXA are val.:"<<0xD<<"\n";
    cout<<"Cifra E din HEXA are val.:"<<0xE<<"\n";
    cout<<"Cifra F din HEXA are val.:"<<0xF<<"\n";
    cout<<"Val. const. hexa 0x7ac1e este: "<<0x7ac1e<<'\n';
```

```
cout<<"Val. const. octale 171 este: "<<0171<<"\n";
cout<<"O const. octala se memoreaza pe "<<sizeof(011)<<" octeti\n";
cout<<"O const.oct.long se mem pe ";cout<<sizeof(011L)<<" octeti\n";}
```

5.2.3. Constante caracter.

Constantele caracter sunt încadrate între apostrofuri.

Exemplu:

```
'a' //tip char
```

O constantă caracter are ca valoare **codul ASCII** al caracterului pe care îl reprezintă.

- Acest set de caractere are următoarele *proprietăți*:
- Fiecărui caracter îi corespunde o valoare întreagă distinctă (ordinală);
- Valorile ordinale ale literelor mari sunt ordonate și consecutive ('A' are codul ASCII 65, 'B' - codul 66, 'C' - codul 67, etc.);
- Valorile ordinale ale literelor mici sunt ordonate și consecutive ('a' are codul ASCII 97, 'b' - codul 98, 'c' - codul 99, etc.);
- Valorile ordinale ale cifrelor sunt ordonate și consecutive ('0' are codul ASCII 48, '1' - codul 49, '2' - codul 50, etc.).

- Constante caracter *corespunzătoare caracterelor imprimabile*. O constantă caracter corespunzătoare unui caracter imprimabil se reprezintă prin caracterul respectiv inclus între apostrofuri.

Exemplu:

Constantă caracter	Valoare
'A'	65
'a'	97
'0'	48
'*'	42

Excepții de la regula de mai sus le constituie *caracterele imprimabile apostrof (')* și *backslash (\)*. Caracterul *backslash* se reprezintă: '\\'. Caracterul *apostrof* se reprezintă: '\ '.

- Constante caracter *corespunzătoare caracterelor neimprimabile*. Pentru caracterele neimprimabile, se folosesc *secvențe escape*. O secvență escape furnizează un mecanism general și extensibil pentru reprezentarea caracterelor invizibile sau greu de obținut. În tabelul 2 sunt prezentate câteva caractere escape utilizate frecvent.

Tabelul 2.

Constantă caracter	Valoare (Cod ASCII)	Denumirea caracterului	Utilizare
'\n'	10	LF	rând nou (Line Feed)
'\t'	9	HT	tabulator orizontal
'\r'	13	CR	poziționează cursorul în coloana 1 din rândul curent
'\f'	12	FF	salt de pagină la imprimantă (Form Feed)
'\a'	7	BEL	activare sunet

O constantă caracter pentru o secvență escape poate apare însă, și sub o formă în care se indică codul ASCII, în octal, al caracterului dorit:

'\ddd' unde d este o cifră octală.

Exemple:

'\11' (pentru '\t')

reprezintă constanta caracter backspace, cu codul 9 în baza 10, deci codul 11 în baza 8.

'\15' (pentru '\r')

reprezintă constanta caracter CR, cu codul 13 în baza 10, deci codul 11 în baza 8.

Exercițiu: Să se scrie următorul program și să se urmărească rezultatele execuției acestuia.

```
#include <iostream.h>
void main(void)
{
    cout<<"Un caracter este memorat pe "<<sizeof(char)<<" octet\n";
    cout<<"Caracterul escape \n este memorat pe ";
    cout<<sizeof('\n')<<" octet\n";
    cout<<"Caracterul escape '\\n' este memorat pe "<<sizeof('\n')<<" octet\n";
    cout<<"Caracterul '9' este memorat pe "<<sizeof('9')<<" octet\n";
    cout<<'B';cout<<' ';cout<<'c';cout<<'t';
    cout<<'t';cout<<'9';cout<<'b';cout<<'a';
    cout<<'L';cout<<'v';cout<<'L';
    cout<<' ';cout<<'t';cout<<' ";cout<<'\\';cout<<'n';
    cout<<'a';cout<<'7';
}
```

5.2.4. Constante șir de caractere. Constanta șir este o succesiune de zero sau mai multe caractere, încadrate de ghilimele. În componența unui șir de caractere, poate intra orice caracter, deci și caracterele escape. Lungimea unui șir este practic nelimitată. Dacă se dorește continuarea unui șir pe rândul următor, se folosește caracterul backslash.

Caracterele componente ale unui șir sunt memorate într-o zonă continuă de memorie (la adrese succesive). Pentru fiecare caracter se memorează codul ASCII al acestuia. După ultimul caracter al șirului, compilatorul plasează automat *caracterul NULL* (\0), caracter care reprezintă *marcatorul sfârșitului de șir*. Numărul de octeți pe care este memorat un șir va fi, deci, mai mare cu 1 decât numărul de caractere din șir.

Exemple:

```
"Acesta este un șir de caractere" //constantă șir memorată pe 32 octeți
"Șir de caractere continuat"
pe rândul următor!" //constantă șir memorată pe 45 octeți
"Șir \t cu secvențe escape\n" //constantă șir memorată pe 26 octeți
'\n' //constantă caracter memorată pe un octet
"\n" //constanta șir memorată pe 2 octeți (codul caracterului escape și terminatorul de șir)
"a\4" /*Șir memorat pe 4 octeți:
      Pe primul octet: codul ASCII al caracterului a
      Pe al doilea octet: codul ASCII al caracterului escape \a
      Pe al treilea octet: codul ASCII al caracterului 4
      Pe al patrulea octet: terminatorul de șir NULL, cod ASCII 0*/
"\\ASCII\\" /*Șir memorat pe 8 octeți:
      Pe primul octet: codul ASCII al caracterului backslash
      Pe al doilea octet: codul ASCII al caracterului A
      Pe al treilea octet: codul ASCII al caracterului S
```

Pe al patrulea octet: codul ASCII al caracterului S
 Pe al 6-lea octet: codul ASCII al caracterului I
 Pe al 7-lea octet: codul ASCII al caracterului I
 Pe al 8-lea octet: codul ASCII al caracterului backslash
 Pe al 9-ea octet: terminatorul de șir NULL, de cod ASCII 0 */
 "1\175a" /*Șir memorat pe 4 octeți:
 Primul octet: Codul ASCII al caracterul 1
 Al 2-lea octet: codul ASCII 125 (175 în octal) al caracterului }
 Al 3-lea octet: codul ASCII al caracterului a
 Al 4-lea octet: codul ASCII 0 pentru terminatorul șirului */

Exercițiu: Să se scrie următorul program și să se urmărească rezultatele execuției acestuia.

```
#include <iostream.h>
void main()
{ cout<<"Șirul \"Ab9d\" este memorat pe:"<<sizeof("Ab9d")<<" octeți\n";
  cout<<"Șirul \"Abcd\t\" este memorat pe:"<<sizeof("Abcd\t")<<" octeți\n";
  cout<<"Șirul \"\n\" este memorat pe "<<sizeof("\n")<<" octeți\n";
  cout<<"Șirul \"\\n\" este memorat pe "<<sizeof("\\n")<<" octeți\n";
  cout<<"Șirul \"ABCDE\" se memorează pe "<<sizeof("ABCDE")<<" octeți\n";}
```

5.3. Variabile. Spre deosebire de constante, variabilele sunt date (obiecte informaționale) ale căror valori se pot modifica în timpul execuției programului. Și variabilele sunt caracterizate de attributele *nume*, *tip*, *valoare* și *clasă de memorare*. Variabilele sunt *nume simbolice* utilizate pentru memorarea valorilor introduse pentru datele de intrare sau a rezultatelor. Dacă la o constantă ne puteam referi folosind caracterele componente, la o variabilă ne vom referi prin numele ei. Numele unei variabile ne permite accesul la valoarea ei, sau schimbarea valorii sale, dacă este necesar acest lucru. Numele unei variabile este un identificator ales de programator. Ca urmare, trebuie respectate regulile enumerate în secțiunea identificatori.

Dacă o dată nu are legături cu alte date (de exemplu, relația de ordine), vom spune că este o dată *izolată*. O dată izolată este o *variabilă simplă*. Dacă datele se grupează într-un anumit mod (în tablouri - vectori, matrici - sau structuri), variabilele sunt *compuse (structurate)*.

În cazul constantelor, în funcție de componența literalului, compilatorul stabilea, automat, tipul constantei. În cazul variabilelor este necesară specificarea tipului fiecăreia, la declararea acesteia. Toate variabilele care vor fi folosite în program, trebuie declarate înainte de utilizare.

5.3.1. Declararea variabilelor. Modul general de declarare a variabilelor este:

```
tip_variabile listă_nume_variabile;
```

Se specifică tipul variabilei(lor) și o listă formată din unul sau mai mulți identificatori ai variabilelor de tipul respectiv. Într-un program în limbajul C++, declarațiile de variabile pot apare în orice loc în programul sursă. La declararea variabilelor, se rezervă în memorie un număr de octeți corespunzător tipului variabilei, urmând ca ulterior, în acea zonă de memorie, să fie depusă (memorată, înregistrată) o anumită valoare.

Exemple:

```
int i, j; /*declararea var. simple i, j, de tip int. Se rezervă pentru i și j câte 16 biți (2octeți)*/
char c; /* declararea variabilei simple c, de tip char. Se rezervă un octet. */
float lungime; /* declararea variabilei simple lungime; se rezervă 4 octeți */
```

5.3.2. Inițializarea variabilelor în declarații. În momentul declarării unei variabile, acestea i se poate da (asigna, atribui) o anumită valoare. În acest caz, în memorie se rezervă numărul de locații corespunzător tipului variabilei respective, iar valoarea va fi depusă (memorată) în acele locații.

Forma unei declarații de variabile cu atribuire este:

```
tip_variabilă nume_variabilă=expresie;
```

Se evaluează expresia, iar rezultatul acesteia este asignat variabilei specificate.

Exemple:

```
char backslash='\\'; //declararea și inițializarea variabilei simple backslash
int a=7*9+2; /* declararea variabilei simple a, de tip int și inițializarea ei cu valoarea 65*/
float radiani, pi=3.14; /*declararea variabilei radiani; declararea și inițializarea var. pi*/
short int z=3; //declararea și inițializarea variabilei simple z
char d='\011';
char LinieNoua='\n';
double x=9.8, y=0;
```

Compilerul C++ furnizează mecanisme care permit programatorului să influențeze codul generat la compilare, prin așa-numiții *calificatori*.

Aceștia sunt:

- `const`;
- `volatile`.

Calificatorul *const* asociat unei variabile, nu va permite modificarea ulterioară a valorii acesteia, prin program (printr-o atribuire). Calificatorul *volatile* (cel implicit) are efect invers calificatorului *const*. Dacă după calificator nu este specificat tipul datei, acesta este considerat tipul implicit, adică *int*.

Exemple:

```
const float b=8.8;
volatile char terminator; terminator='@'; terminator='*'; //permis
b=4/5; //nepermisa modificarea valorii variabilei b
const w; volatile g; //w, g de tip int, implicit
```

5.3.3. Operații de intrare/ieșire. Limbajele C/C++ nu posedă instrucțiuni de intrare/ieșire, deci de citire/scriere (ca limbajul PASCAL, de exemplu). În limbajul C aceste operații se realizează cu ajutorul unor funcții (de exemplu, `printf` și `scanf`), iar în limbajul C++ prin supraîncărcarea operatorilor (definirea unor noi proprietăți ale unor operatori existenți, fără ca proprietățile anterioare să dispară), mai precis a operatorilor `>>` și `<<`. Se va folosi în continuare abordarea limbajului C++, fiind, în momentul de față, mai simplă. În limbajul C++ sunt predefinite următoarele dispozitive logice de intrare/ieșire:

`cin` - console **input** - dispozitivul de intrare (tastatura);

`cout` - console **output** - dispozitivul de ieșire (monitorul).

Așa cum se va vedea în **capitolul 9**, `cin` și `cout` sunt, de fapt, obiecte (predefinite). Transferul informației se realizează cu **operatorul `>>`** pentru intrare și **operatorul `<<`** pentru ieșire. Utilizarea dispozitivelor de intrare/ieșire cu operatorii corespunzători determină deschiderea unui canal de comunicație a datelor către dispozitivul respectiv. După operator se specifică informațiile care vor fi citite sau afișate.

Exemple:

```
cout << var; /* afișează valoarea variabilei var pe monitor*/
```

```
cin >> var; /* citește valoarea variabilei var de la tastatură */
```

Sunt posibile operații multiple, de tipul:

```
cout << var1 << var2 << var3;
cin >> var1 >> var2 >> var3;
```

În acest caz, se efectuează succesiv, de la stânga la dreapta, scrierea, respectiv citirea valorilor variabilelor `var1`, `var2` și `var3`.

Operatorul `>>` se numește *operator extractor* (extrage valori din fluxul datelor de intrare, conform tipului acestora), iar operatorul `<<` se numește *operator insertor* (inserează valori în fluxul datelor de ieșire, conform tipului acestora). Tipurile de date citite de la tastatură pot fi toate tipurile numerice, caracter sau șir de caractere. Tipurile de date transferate către ieșire pot fi: toate tipurile numerice, caracter sau șir de caractere. Operanzii operatorului extractor (`>>`) pot fi doar nume de variabile. Operanzii operatorului insertor (`<<`) pot fi nume de variabile (caz în care se afișează valoarea variabilei), constante sau expresii. Utilizarea dispozitivelor și operatorilor de intrare/ieșire în C++ impune includerea fișierului **`iostream.h`**.

Exemple:

```
char c;
cout<<"Aștept un caracter:"; //afișarea constantei șir de caractere, deci a mesajului
cin>>c; //citirea valorii variabilei c, de tip caracter
int a, b, e; double d;
cin>>a>>b>>e>>d; //citirea valorilor variabilelor a, b, e, d de tip int, int, int, double
cout<<"a="<<a<<"Valoarea expresiei a+b este:"<<a+b<<"\n";
```

6. Operatori și expresii

Datele (constante sau variabile) legate prin operatori, formează *expresii*. Operatorii care pot fi aplicați datelor (operanzilor) depind de tipul operanzilor, datorită faptului că tipul unei date constă într-o mulțime de valori pentru care s-a adoptat un anumit mod de reprezentare în memoria calculatorului și o *mulțime de operatori* care pot fi aplicați acestor valori.

Operatorii pot fi:

- unari (necesită un singur operand);
- binari (necesită doi operanzi);
- ternari (trei operanzi).

O *expresie* este o combinație corectă din punct de vedere sintactic, formată din operanzi și operatori. Expresiile, ca și operanzii, au *tip* și *valoare*.

6.1. Operatori.

- Operatorul unar ***adresă &***, aplicat identificatorului unei variabile, furnizează adresa la care este memorată aceasta. Poate fi aplicat *oricărui tip de date* și se mai numește *operator de referențiere*.

Exemplu:

```
int a;
cout<<"Adresa la care este memorata variabila a este:"<<&a;
```

- Operatorul ***de atribuire (de asignare)*** este un operator *binar* care se aplică tuturor tipurilor de variabile. Este folosit sub formele următoare:

```
nume_variabilă=expresie;                   sau:                   expresie1=expresie2;
```

Se evaluează expresia din membrul drept, iar valoarea acesteia este atribuită

variabilei din membrul stâng. Dacă tipurile membrilor stâng și drept diferă, se pot realiza anumite conversii.

Exemplu:

```
float x; int a,b; x=9.18;
a=b=10;
int s; s=a+20*5;           //rezultat: s=110
s=x+2;                     //rezultat s=11, deoarece s este int.
```

Așa cum se observă în linia a 2-a din exemplul precedent, operatorul de atribuire poate fi utilizat de mai multe ori în aceeași expresie. Asociativitatea operatorului are loc de la dreapta la stânga. Astfel, mai întâi $b=10$, apoi $a=b$.

Exercițiu: Să se scrie următorul program și să se urmărească rezultatele execuției acestuia.

```
#include <iostream.h>
void main()
{
float x,y=4.25; char car='A'; int a,b,c;
cout<<"Val. lui y este:"<<y<<'\\n';           //Afișare: Val. lui y este:4.25
x=y; cout<<"Val. lui x este:"<<x<<'\\n';       //Afișare: Val. lui x este:4.25
a=x;cout<<"Val.lui a este:"<<a<<'\\n'; //Afișare:Val. lui a este:4, deoarece a de tip int!!!
c=b=a; cout<<"b="<<b<<"\\tc="<<c<<'\\n';       //Afișare: b=4      c=4
cout<<"Introduceți val. lui c:"; cin>>c;         // citire val. pentru c
cout<<"Val. lui c este:"<<c<<'\\n';           //Afișare: Val. lui c este:4
}
```

Operatorul poate fi aplicat tipurilor de date întregi, reale, caracter, și chiar șiruri de caractere, așa cum se va vedea în capitolele următoare (exemplu: `char șir [10]="a5dfgthklj"`).

- Operatori *aritmetici unari*:

Operator	Semnificație	Exemple
-	Minus unar	-a
++	Operator de incrementare (adună 1 la valoarea operandului)	a++ sau ++a
--	Operator de decrementare (scade 1 din valoarea operandului)	a-- sau --a

- Operatorul - unar schimbă semnul operandului.

Exemplu:

```
int a,b; cout<<"a="<<-a<<'\\n'; b=-a;
cout<<"b="<<b<<'\\n';
```

Operatorul - unar poate fi aplicat datelor întregi, reale, caracter.

- Operatorii de incrementare și decrementare pot fi aplicați *datelor numerice sau caracter*.

Ambii operatori pot fi folosiți în formă *prefixată*, înaintea operandului, ($++a$, respectiv $--a$) sau *postfixată*, după operand ($a++$, respectiv $a--$).

Operatorul de decrementare $--$ care poate fi folosit în formă *prefixată* ($--a$) sau *postfixată* ($a--$).

Utilizarea acestor operatori în expresii, în formă prefixată sau postfixată, determină evaluarea acestora în moduri diferite, astfel:

$y=++x$	este echivalent cu:	$x=x+1; y=x;$
$y=x++$	este echivalent cu:	$y=x; x=x+1;$
$y=--x$	este echivalent cu:	$x=x-1; y=x;$

$y=x--$ este echivalent cu: $y=x; x=x-1;$

Exercițiu: Să se scrie următorul program și să se urmărească rezultatele execuției acestuia.

```
#include <iostream.h>
void main()
{
    int a=9; cout<<"a++="<<a++<<"\n"; //Afișare: a++=9
    cout<<"a="<<a<<"\n"; //Afișare: a=10
    a=9; //Revenire în situația anterioară
    cout<<"++a="<<++a<<"\n"; //Afișare: ++a=10
    cout<<"a="<<a<<"\n"; //Afișare: a=10
    a=9; cout<<"a--="<<a--<<"\n"; //Afișare: a--=9
    cout<<"a="<<a<<"\n"; //Afișare: a=8
    a=9; //Revenire în situația anterioară
    cout<<"--a="<<--a<<"\n"; //Afișare: --a=8
    cout<<"a="<<a<<"\n"; //Afișare: a=8
    int z,x=3; z=x++-2;
    cout<<"z="<<z<<"\n"; //Afișare: z=1
    cout<<"x="<<x<<"\n"; //Afișare: x=4
    x=3; z=++x-2; cout<<"z="<<z<<"\n"; //Afișare: z=2
    cout<<"x="<<x<<"\n"; //Afișare: x=4
}
```

□ Operatori *aritmetici binari*:

Operator	Semnificație	Exemple
+	Adunarea celor doi operanzi	a+b
-	Scăderea celor doi operanzi	a-b
*	Înmulțirea celor doi operanzi	a*b
/	Împărțirea celor doi operanzi	a/b
%	Operatorul modulo (operatorul rest)	a%b

(furnizează restul împărțirii operatorului stâng la operatorul drept).

Operatorul modulo se aplică numai operanzilor întregi (de tip int sau char).

Ceilați operatori aritmetici binari pot fi aplicați datelor întregi sau reale.

Dacă într-o expresie cu 2 operanzi și un operator binar aritmetic, ambii operanzi sunt întregi, rezultatul expresiei va fi tot un număr întreg. De exemplu, la evaluarea expresiei $9/2$, ambii operanzi fiind întregi, rezultatul furnizat este numărul întreg 4.

Operatorii prezentați respectă o serie de reguli de precedență (prioritate) și asociativitate, care determină precis modul în care va fi evaluată expresia în care aceștia apar. În tabelul 3 sunt prezentați operatorii anteriori, în ordinea descrescătoare a priorității. Precedența operatorilor poate fi schimbată cu ajutorul parantezelor.

Tabelul 3.

Clasă de operatori	Operatori	Asociativitate
Unari	- (unar) ++ --	de la dreapta la stânga
Multiplicativi	* / %	de la stânga la dreapta
Aditivi	+ -	de la stânga la dreapta
Atribuire	=	de la dreapta la stânga

Exercițiu: Să se scrie următorul program și să se urmărească rezultatele execuției acestuia.

```
#include <iostream.h>
void main()
{
    int rezult, a=20,b=2,c=25,d=4; rezult=a-b;
    cout<<"a-b="<<rezult<<'\\n'; // Afișare: a-b=18
    rezult=a+b; cout<<"a+b="<<rezult<<'\\n'; // Afișare: a+b=22
    rezult=a*b;cout<<"c*b="<<rezult<<'\\n'; // Afișare: c*b=50
    rezult=a/d; cout<<"a/d="<<rezult<<'\\n'; // Afișare: a/d=5
    rezult=c%b; cout<<"c%b="<<rezult<<'\\n'; // Afișare: c%b=1
    rezult=c/b*d; cout<<"c/b*d="<<rezult<<'\\n'; // Afișare: c/b*d=48
    rezult= -b+a; cout<<"-b+a="<<rezult<<'\\n'; // Afișare: -b+a=18
    rezult= -(b+a); cout<<"-(b+a)="<<rezult<<'\\n'; // Afișare: -(b+a)=-22
    rezult=b+c*d;cout<<"b+c*d="<<rezult<<'\\n'; // Afișare: b+c*d=102
    rezult=(b+c)*d;cout<<"(b+c)*d="<<rezult<<'\\n'; // Afișare: (b+c)*d=108
}
```

□ Operatori *aritmetici binari compuși*

Operator	Semnificație	Exemple
+=	a=a+b	a+=b
-=	a=a-b	a-=b
*=	a=a*b	a*=b
/=	a=a/b	a/=b
%=	a=a%b	a%=b

Acești operatori se obțin prin combinarea operatorilor aritmetici binari cu operatorul de atribuire și sunt folosiți sub forma următoare:

expresie1 operator= expresie2;

Rezultatul obținut este același cu rezultatul obținut prin:

expresie1 = expresie1 operator expresie2;

Toți acești operatorii modifică valoarea operandului stâng prin adunarea, scăderea, înmulțirea sau împărțirea acestuia prin valoarea operandului drept.

Construcția x+=1 generează același rezultat ca expresia x=x+1.

Observațiile referitoare la operatorii aritmetici binari sunt valabile și pentru operatorii aritmetici binari compuși. Operatorii aritmetici binari compuși au aceeași prioritate și asociativitate ca și operatorul de atribuire.

Exercițiu: Să se scrie următorul program și să se urmărească rezultatele execuției acestuia.

```
#include <iostream.h>
void main()
{
    int a,b; float c=9.3; a=3; b=8;
    cout<<"a="<<a<<"\n";           //Afișare a=3
    a+=b; cout<<"a="<<a<<"\n";       //Afișare a=11
    a-=b; cout<<"a="<<a<<"\n";       //Afișare a=-5
    a*=b; cout<<"a="<<a<<"\n";       //Afișare a=24
    a/=b; cout<<"a="<<a<<"\n";       //Afișare a=0
    a%=b; cout<<"a="<<a<<"\n";       //Afișare a=3
}
```

□ Operatori *relaționali binari*

Operator	Semnificație	Exemple
==	Egal cu	a==b
!=	Diferit de	a!=b
<	Mai mic decât	a<b
<=	Mai mic sau egal	a<=b
>	Mai mare decât	a>b
>=	Mai mare sau egal	a>=b

Primii doi operatori mai sunt numiți *operatori de egalitate*. Operatorii relaționali servesc la compararea valorilor celor doi operanzi și nu modifică valorile operanzilor. Rezultatul unei expresii în care apare unul din operatorii relaționali binari este întreg și are valoarea zero (0) dacă relația este falsă, sau valoarea unu (1) (sau diferită de 0 în cazul compilatoarelor sub UNIX), dacă relația este adevărată. Acești operatorii pot fi aplicați datelor de tip întreg, real sau char.

Regulile de precedență și asociativitate ale acestor operatori sunt prezentate în tabelul 4.

Tabelul 4.

Clasă de operatori	Operatori	Asociativitate
Unari	- (unar) ++ --	de la dreapta la stânga
Multiplicativi	* / %	de la stânga la dreapta
Aditivi	+ -	de la stânga la dreapta
Atribuire	=	de la dreapta la stânga
Relaționali	< <= > >=	de la stânga la dreapta
De egalitate	== !=	de la stânga la dreapta
Atribuire și aritmetici binari	= *= /= %= += -=	de la dreapta la stânga

Observație: Deosebirea dintre operatorii == (relațional, de egalitate) și = (de atribuire) constă în faptul că primul nu modifică valoarea nici unuia dintre operanzii săi, pe când cel de-al doilea modifică valoarea operandului stâng (vezi exemplul următor).

Exercițiu: Să se scrie următorul program și să se urmărească rezultatele execuției acestuia.

```
#include <iostream.h>
void main()
{
    int a=1, b=20, lim=100; int rezult; rezult=a<b;
    cout<<"a<b="<<rezult<<"\n";
    // Afișare: a<b=1 (sau o altă valoare diferită de zero pentru alte compilatoare)
    rezult=a<=b;
```

```
//operatorul relațional <= are prioritate mai mare decât cel de atribuire
cout<<"a<=b="<<rezult<<'\\n' ;
// Afisare: a<b=1 (sau o alta valoare diferită de zero pentru alte compilatoare)
rezult=a>b; cout<<"a>b="<<rezult<<'\\n' ; // Afisare: a<b=0
rezult=a+10>=lim; cout<<"a+10>=lim="<<rezult<<'\\n' ;
/* Operatorul + are prioritate mai mare decât operatorul >=. Afisare: a+10>=lim=0 */
rezult=a+(10>=lim); cout<<"a+(10>=lim)="<<rezult<<'\\n' ;
/* Schimbarea priorității operatorilor prin folosirea parantezelor; Afisare: a+(10>=lim)=1 */
rezult=a==b;
cout<<"a==b="<<rezult<<'\\n' ; // Afisare: a==b=0
cout<<"a="<<a<<'\\n' ; // Afisare: a=1
cout<<"b="<<b<<'\\n' ; // Afisare: b=20
rezult=a=b; cout<<"a=b="<<rezult<<'\\n' ; // Afisare: a=b=20
cout<<"a="<<a<<'\\n' ; // Afisare: a=20
cout<<"b="<<b<<'\\n' ; // Afisare: b=20
rezult=5>b>10;cout<<"b="<<b<<'\\n' ; // Afisare: b=20
cout<<"5>b>10="<<rezult<<'\\n' ; //Echivalent cu (5>b)>10 Afisare: 5>b>10=0
}
```

□ Operatori *logici pe cuvânt*

Operator	Semnificație	Exemple
!	Not (negație logică)	!(a==b)
&&	And (conjunție, și logic)	(a>b)&&(b>c)
	Or (disjunție, sau logic)	(a>b) (b>c)

Acești operatori pot fi aplicați datelor de tip întreg, real sau caracter. Evaluarea unei expresii în care intervin operatorii logici se face conform tabelului 5.

Tabelul 5.

x	y	!x	x&& y	x y
adevărat (1)	adevărat (1)	fals (0)	adevărat (1)	adevărat (1)
adevărat (1)	fals (0)	fals (0)	fals (0)	adevărat (1)
fals (0)	adevărat (1)	adevărat (1)	fals (0)	adevărat (1)
fals (0)	fals (0)	adevărat (1)	fals (0)	fals (0)

Expresia `!expresie` are valoarea 0 (fals) dacă expresia-operand are o valoare diferită de zero și valoarea unu (adevărat) dacă expresia-operand are valoarea zero.

Expresia `expresie1 || expresie2` are valoarea diferită de 0 (true) dacă FIE expresie1, FIE expresie2 au valori diferite de zero.

Expresia `expresie1 && expresie2` are valoarea diferită de 0 (true) dacă AMBELE expresii-operand (expresie1 și expresie2) au valori diferite de zero.

Exercițiu: Să se scrie următorul program și să se urmărească rezultatele execuției acestuia.

```
#include <iostream.h>
void main()
{ int a=0, b=10, c=100, d=200; int rezult; rezult=a&&b;
cout<<"a&&b="<<rezult<<'\\n' ; //Afisare a&&b=0
rezult=a||b; cout<<"a||b="<<rezult<<'\\n' ;//Afisare a||b=1 (sau valoare nenula)
rezult=!a;cout<<"!a="<<rezult<<'\\n' ; //Afisare !a=1 (sau valoare nenula)
rezult=!b; cout<<"!b="<<rezult<<'\\n' ; //Afisare !b=0
rezult=(a>b) || (b>c);cout<<"(a>b) || (b>c)="<<rezult<<'\\n' ;
//Afisare (a>b) || (b>c)=1(sau valoare nenula)
rezult=!(c<d);cout<<"!(c<d)="<<rezult<<'\\n' ;//Afisare !(c>d)=0
rezult=(a-b)&&1;cout<<"(a-b)&&1="<<rezult<<'\\n' ;
//Afisare (a-b)&&1=1(sau valoare nenula)
rezult=d||b&&a;cout<<"d||b&&a="<<rezult<<'\\n' ;//Afisare d||b&&a=1
```

// În evaluarea expresiilor din exemplu, s-au aplicat prioritățile operatorilor, indicate în tabelul 6.

Tabelul 6.

Clasă de operatori	Operatori	Asociativitate
Unari	! - (unar) ++ --	de la dreapta la stânga
Multiplcativi	* / %	de la stânga la dreapta
Aditivi	+ -	de la stânga la dreapta
Atribuire	=	de la dreapta la stânga
relaționali	< <= > >=	de la stânga la dreapta
de egalitate	== !=	de la stânga la dreapta
logici	&&	de la stânga la dreapta
logici		de la stânga la dreapta
atribuire și aritmetici binari	= *= /= %= += -=	de la dreapta la stânga

Exercițiu: Să se scrie un program care citește un număr real și afișează 1 dacă numărul citit aparține unui interval ale cărui limite sunt introduse tot de la tastatură, sau 0 în caz contrar.

```
#include <iostream.h>
void main()
{
double lmin, lmax, nr;cout<<"Numar=";cin>>nr;
cout<<"Limita inferioară a intervalului:"; cin>>lmin;
cout<<"Limita superioară a intervalului:"; cin>>lmax;
cout<<(nr>=lmin && nr<=lmax); }
```

□ Operatori *logici pe bit*

Operator	Semnificație	Exemple
~	Negație (cod complementar față de unu)	~a
&	AND (Conjunție, și logic pe bit)	a & 0377
	OR (Disjunție, sau logic pe bit)	a 0377
^	XOR (Sau exclusiv logic pe bit)	a^b
<<	Deplasare stânga	0377 << 2
>>	Deplasare dreapta	0377 >> 2

Acești operatori nu se aplică numerelor reale, ci numai datelor de tip întreg sau caracter. Primul operator este unar, ceilalți binari. Operatorii acționează la nivel de bit, la nivelul reprezentării interne (în binar), conform tabelului 7.

Tabelul 7.

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

Operatorul ~ are aceeași prioritate ca și ceilalți operatori unari. El furnizează complementul față de unu al unui întreg, adică va schimba fiecare bit de pe 1 în zero și invers. Operatorii de deplasare pe bit (<< și >>) efectuează deplasarea la stânga sau la dreapta a operandului stâng, cu numărul de biți indicați de operandul drept. Astfel, x<<2 deplasează biții din x la stânga, cu două poziții, introducând zero pe pozițiile rămase vacante.

Exemple:

```
int a=3;      //Reprezentare internă a lui a (pe 2 octeți): 0000000000000011
int b=5;      //Reprezentare internă a lui b (pe 2 octeți): 0000000000000101
int rez=~a;
cout<<"~"<<a<<"'<<rez<<"\n"; //~3=-4
//Complementul față de unu este: 111111111111100 (în octal: 017777774 (!a=-4)
rez=a & b; cout<<a<<"'<<b<<"'<<rez<<"\n"; //3&5=1
//a&b=0000000000000001
rez=a^b; cout<<a<<"'<<b<<"'<<rez; //3^5=6
//a^b=0000000000000110
rez=a|b; cout<<a<<"'<<b<<"'<<rez; //3|5=7
//a|b=0000000000000111
rez=a<<2; cout<<a<<"<<"<<3<<"'<<rez; //3<<2=16=2*2^3
//a<<2=0000000001100000
rez=5>>2; cout<<b<<">>"<<2<<"'<<rez; //5>>2=1=5/2^2
//b>>2=0000000000000001
```

Operatorul binar ^ își găsește o utilizare tipică în expresii ca: $x \& 077$, care maschează ultimii 6 biți ai lui x pe zero.

Operatorul & este adesea utilizat în expresii ca $x \& 0177$, unde setează toți biții pe zero, cu excepția celor de ordin inferior din x.

Operatorul | este utilizat în expresii ca: $x \& \text{MASK}$, unde setează pe unu biții care în x și masca MASK sunt setați pe unu.

Operatorii logici pe bit & și | sunt diferiți de operatorii logici && și || (pe cuvânt).

Deplasarea la stânga a unei date cu n poziții este echivalentă cu înmulțirea valorii acesteia cu 2^n . Deplasarea la dreapta a unei date fără semn cu n poziții este echivalentă cu împărțirea valorii acesteia cu 2^n .

Combinând operatorii logici pe bit cu operatorul de atribuire, se obțin operatorii:

$\&=$, $\wedge=$, $|=$, $\ll=$, $\gg=$.

- Operatorul **condițional**. Este un operator ternar (necesită 3 operanzi), utilizat în construcții de forma:

$\text{expresie1?expresie2:expresie3}$

Se evaluează *expresie1*. Dacă aceasta are o valoare diferită de zero, atunci tipul și valoarea întregii expresii vor fi aceleași cu tipul și valoarea *expresie2*. Altfel (dacă expresie1 are valoarea zero), tipul și valoarea întregii expresii vor fi aceleași cu tipul și valoarea *expresie3*. Deci operatorul condițional este folosit pentru a atribui întregii expresii tipul și valoarea expresiei2 sau a expresiei3, în funcție de o anumită condiție. Acest lucru este echivalent cu:

Dacă *expresie1* diferită de zero

Atunci evaluează *expresie2*

Altfel evaluează *expresie3*

Exemplu:

$\text{int semn}=(x<0)?-1:1$

Dacă $x<0$, atunci $\text{semn}=-1$, altfel $\text{semn}=1$.

- Operatorul **virgulă**. Este utilizat în construcții de forma:

$\text{expresie1}, \text{expresie2}$

Operatorul virgulă forțează evaluarea unei expresii de la stânga la dreapta. Tipul și valoarea întregii expresii este dată de tipul și valoarea expresiei2. Operatorul virgulă este folosit în instrucțiunea **for**. Operatorul virgulă are cea mai mică prioritate.

Exemplu:

```
int x, c, y;
cout<<"Astept val. ptr. y:"; cin>>y;
x=(c=y, c<=5); /* c va primi valoarea lui y (citită); se verifică dacă c este mai mic sau
                egal cu 5. Dacă nu, x=0; dacă da, x=1 sau x=valoare diferită de zero)*/
x++, y--;      //întâi este incrementat x, apoi este decrementat y
```

- Operatorul **sizeof()**. Este un operator unar, care are ca rezultat numărul de octeți pe care este memorată o dată de un anumit tip. Operandul este *un tip* sau *o dată (constantă sau variabilă) de un anumit tip*.

Exemple:

```
cout<<sizeof(int); // afișează numărul de octeți pe care este memorat un întreg (2)
cout<<sizeof("ab6*"); // afișează 5, nr. de octeți pe care este memorată constanta șir "ab6*"
```

- Operatorul (**tip**). Este un operator unar care apare în construcții numite "cast" și convertește tipul operandului său la tipul specificat între paranteze.

Exemple:

```
int a; (float) a; // convertește operandul a (care era de tip întreg) în float
```

În afara operatorilor prezentați, există și alții, pe care îi vom enumera în continuare. Despre acești operatori vom discuta în capitolele viitoare, când cunoștințele acumulate vor permite acest lucru.

- Operatorul unar *****. Este operator unar, numit și *operator de deferențiere*. Se aplică unei expresii de tip pointer și este folosit pentru a accesa conținutul unei zone de memorie spre care pointează operatorul. Operatorii & (adresă) și * sunt complementari.

Exemplu: Expresia *a este înlocuită cu valoarea de la adresa conținută în variabila pointer a.

- Operatorii **paranteză**. Parantezele rotunde () se utilizează în expresii, pentru schimbarea ordinii de efectuare a operațiilor, sau la apelul funcțiilor. La apelul funcțiilor, parantezele rotunde încadrează lista parametrilor efectivi. Din acest motiv, parantezele rotunde sunt numite și *operatori de apel de funcție*.

Exemplu:

```
double sum(double a, double b);
/*declar. funcției sum, care primește 2 argumente reale(double) și returnează o valoare tip double */
void main()
{
    . . .
    double a=sum(89.9, 56.6); //apelul funcției sum, cu parametri efectivi 89.9 și 56.6
    int s0=6; double s1=(s0+9)/a; //folosirea parantezelor în expresii
    . . .
}
```

- Operatorii **de indexare**. Operatorii de indexare sunt parantezele pătrate []. Acestea includ expresii întregi care reprezintă indici ai unui tablou.
- Operatori **de acces la membri structurilor**. Operatorii ::, ., ->, .* și ->* permit accesul la componentele unei structuri.

În tabelul 8 sunt prezentați toți operatorii, grupați pe categorii, cu prioritățile lor și regulile de asociativitate. Operatorii dintr-o categorie au aceeași prioritate.

Tabelul 8.

Nr.	Clasă de operatori	Operatori	Asociativitate
1.	Primari	() [] . -> ::	de la stânga la dreapta
2.	Unari	! ~ ++ -- sizeof (tip) -(unar) *(deferențiere) &(referențiere)	de la stânga la dreapta
3.	Multiplcativi	* / %	de la stânga la dreapta
4.	Aditivi	+ -	de la stânga la dreapta
5.	Deplasare pe bit	<< >>	de la stânga la dreapta
6.	Relaționali	< <= > >=	de la stânga la dreapta
7.	De egalitate	== !=	de la stânga la dreapta
8.		& (ȘI logic pe bit)	de la stânga la dreapta
9.		^ (XOR pe bit)	de la stânga la dreapta
10.		(SAU logic pe bit)	de la stânga la dreapta
11.		&&	de la stânga la dreapta
12.			de la stânga la dreapta
13.	Condițional	?:	de la dreapta la stânga
14.	De atribuire	= += -= *= %= &= ^= = <<= >>=	de la dreapta la stânga
15.	Virgulă	,	de la stânga la dreapta

6.2. Expresii. Prin combinarea operanzilor și a operatorilor se obțin *expresii*. Tipul unei expresii este dat de tipul rezultatului obținut în urma evaluării acesteia. La evaluarea unei expresii se aplică regulile de prioritate și asociativitate a operatorilor din expresie. Ordinea de aplicare a operatorilor poate fi schimbată prin folosirea parantezelor. La alcătuirea expresiilor, este indicată evitarea expresiilor în care un operand apare de mai multe ori.

6.3. Conversii de tip. La evaluarea expresiilor, se realizează conversii ale tipului operanzilor. Conversiile sunt:

- Automate;
- Cerute de evaluarea expresiilor;
- Cerute de programator (prin construcțiile cast), explicite.

Conversiile automate sunt realizate de către compilator și sunt realizate de fiecare dată când într-o expresie apar operanzi de tipul char sau short int.:

char, short int -> int

Conversiile cerute de evaluarea expresiilor sunt efectuate în cazurile în care în expresii apar operanzi de tipuri diferite. Înaintea aplicării operatorilor, se realizează conversia unuia sau a ambilor operanzi:

- Dacă un operand este de tip long int, celălalt este convertit la același tip; tipul expresiei este long int.
- Dacă un operand este de tipul double, celălalt este convertit la același tip; tipul expresiei este double.
- Dacă un operand este de tipul float, celălalt este convertit la același tip; tipul expresiei este float.

Conversiile explicite (cerute de programator) se realizează cu ajutorul construcțiilor cast.

Exemplu: int x=3; float y; y=(float)x/2;

Înainte de a se efectua împărțirea celor 2 operanzi, operandul x (întreg) este convertit în număr real simplă precizie. După atribuire, valoarea lui y va fi 1.5. Dacă nu ar fi fost folosit operatorul de conversie în expresia y=x / 2, operanzii x și 2 fiind întregi, rezultatul împărțirii este întreg, deci y ar fi avut valoarea 1.

ÎNTREBĂRI ȘI EXERCII

Întrebări teoretice.

1. Ce reprezintă datele și care sunt atributele lor?
2. Care sunt diferențele între constante și variabile?
3. Cine determină tipul unei constante?
4. Ce sunt identificatorii?
5. Ce sunt directivele preprocesor?
6. Ce reprezintă variabilele?
7. Ce sunt constantele?
8. Enumerați tipurile simple de variabile.
9. Câte tipuri de directive preprocesor cunoașteți? Exemple.
10. Care este modalitatea de a interzice modificarea valorii unei variabile?
11. Ce loc ocupă declararea variabilelor în cadrul unui program sursă scris în limbajul C++?
12. Ce conțin fișierele header?
13. Ce tipuri de variabile se utilizează pentru datele numerice?
14. Care sunt calificatorii folosiți alături de tipurile de bază pentru obținerea tipurilor derivate de date?
15. Ce semnifică parantezele unghiulare <> care încadrează numele unui fișier header?
16. Care este diferența între constantele 35.2e-1 și 3.52 ? Dar între "\t" și "t"?
17. Ce tip are constanta 6.44 ?
18. Care este diferența între operatorii = și == ?
19. Ce reprezintă caracterele "escape"?
20. Constante întregi.
21. Constante caracter.
22. Ce tipuri de conversii cunoașteți?
23. Care sunt conversiile realizate în mod automat, de către compilator?
24. Constante șir de caractere.
25. Constante reale.
26. Ce operatori ternari cunoașteți?
27. Operatorul virgulă.
28. Operatorul sizeof.
29. Operatori aritmetici binari compuși.
30. Operatorul de referențiere.
31. Operatori relaționali binari.

Exerciții aplicative

1. Să se scrie declarațiile pentru definirea constantelor simbolice: pi, g (acelerația gravitațională), unghi_drept, dimensiune_MAX.
2. Care va fi rezultatul afișat pe ecran în urma execuției următoarelor secvențe de instrucțiuni:
 - ❑ `double a=9/2; cout<<a*5<<'\\n';`
 - ❑ `double a=9.7, b=5.6; cout<<(a+6<b)<<'\\n';`
 - ❑ `double a=9/4; cout<<a*6<<'\\n';`
 - ❑ `double x=3;int y=++x+5;cout<<y<<'\\n';`
 - ❑ `int a=7; cout<<(!a)<<'\\n';`
 - ❑ `int a=10.5; cout<<a++<<'\\n'; cout<<a<<'\\n';`
 - ❑ `int a=7; cout<<++a<<'\\n'; cout<<a<<'\\n';`
 - ❑ `int a=10; cout<<a++<<'\\n'; cout<<a<<'\\n';`

- ☐ `double a=7/2; cout<<a<<'\\n';`
 - ☐ `int x=3; int y=x+-2; cout<<y<<'\\n';`
 - ☐ `int x=3; int y=++x+5; cout<<y<<'\\n';`
 - ☐ `double a=5.6, b=7.45; cout<<(a>b)<<'\\n';`
3. Să se verifice corectitudinea următoarelor secvențe. Pentru cele incorecte, explicați sursa erorilor.
- ☐ `double a=9.7, b=5.2; int c=(a+6<b)++; cout<<c<<'\\n';`
 - ☐ `double a=7/5; double c=a*5++; cout<<c<<'\\n';`
 - ☐ `double a=9.7, b=5.6; int c=(a%6<b)++; cout<<c<<'\\n';`
 - ☐ `double a=5.6, b=7.45; cout<<+(a+5>b)<<'\\n';`
 - ☐ `double a=9.8; double b=9.7; cout<<a%b<<'\\n';`
 - ☐ `cout<<&(a+8)<<'\\n';`
 - ☐ `int I=8; cout<<(I+10)++<<'\\n';`
 - ☐ `double a=8.7; A=(a+8)/56; cout<<A<<'\\n';`
 - ☐ `int x=3/5; int y=x++; char x='J'; cout<<"y="<<y<<'\\n';`
 - ☐ `char a='X'; const int b=89; b+=8; cout<<"b="<<b<<" a="<<a<<'\\n';`
4. Să se scrie un program care afișează următoarele mesaje:
- ☐ Sirul "este dupa-amiaza" este memorat pe octeți.
 - ☐ O marime întreaga este memorată pe ... octeți.
 - ☐ O marime reală, în simplă precizie este memorată pe ... octeți!
 - ☐ O marime reală, în dubla precizie este memorată pe ... byți!
 - ☐ Constanta caracter 'Q' memorată pe ... octeți!
 - ☐ Sirul "a\\n\\n" este memorat pe ... octeți!
 - ☐ Sirul "\\n" este memorat pe ... biți!
 - ☐ Caracterul '\\' este memorat pe biți.
5. Să se evalueze expresiile, știind că: `int i=1;int j=2;int k=-7;double x=0;double y=2.3;`
- ☐ `-i - 5 * j >= k + 1`
 - ☐ `3 < j < 5`
 - ☐ `i + j + k == -2 * j`
 - ☐ `x && i || j - 3`
6. Ce operație logică și ce mască trebuie să folosiți pentru a converti codurile ASCII ale literelor mici în litere mari? Dar pentru conversia inversă?
7. O deplasare la dreapta cu 3 biți este echivalentă cu o rotație la stânga cu câți biți?
8. Să se seteze pe 1 toți biții dintr-un octet, cu excepția bitului cel mai semnificativ.
9. Să se scrie un program care citește o valoare întreagă. Să se afișeze un mesaj care să indice dacă numărul citit este par sau impar.
10. Să se citească două valori întregi. Să se calculeze și să se afișeze restul împărțirii celor două numere.

Implementarea structurilor de control

III

3.1. Implementarea structurii secvențiale

3.2. Implementarea structurii de decizie

3.3. Implementarea structurilor repetitive

3.4. Facilități de întrerupere a unei secvențe

Algoritmul proiectat pentru rezolvarea unei anumite probleme trebuie implementat într-un limbaj de programare; prelucrarea datelor se realizează cu ajutorul *instrucțiunilor*. Instrucțiunea descrie un proces de prelucrare pe care un calculator îl poate executa. O instrucțiune este o construcție validă (care respectă sintaxa limbajului) urmată de; . Ordinea în care se execută instrucțiunile unui program definește așa-numita *structură de control* a programului.

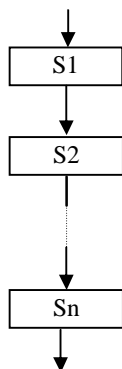
Limbajele moderne sunt alcătuite pe principiile *programării structurate*. Conform lui C. Bohm și G. Jacobini, orice algoritm poate fi realizat prin combinarea a trei structuri fundamentale:

- structura secvențială;
- structura alternativă (de decizie, de selecție);
- structura repetitivă (ciclică).

1. Implementarea structurii secvențiale

Structura secvențială este o înșiruire de secvențe de prelucrare (instrucțiuni), plasate una după alta, în ordinea în care se dorește execuția acestora.

Reprezentarea structurii secvențiale cu ajutorul *schemei logice* (figura 1):



Reprezentarea structurii secvențiale cu ajutorul *pseudocodului*:

```
instr1;  
instr2;  
. . . . .
```

In **Figura 1.** Schema logică pentru structura ază cu ajutorul instrucțiunilor:
secvențială.

Sintaxa: **;**

Instrucțiunea vidă nu are nici un efect. Se utilizează în construcții în care se cere prezența unei instrucțiuni, dar nu se execută nimic (de obicei, în instrucțiunile repetitive).

Exemple:

```
int a;  
.  
.  
int j;  
;  
for ( ; )  
{  
.  
.  
}
```

□ Instrucțiunea *expresie*

Sintaxa: **expresie;**

sau: **apel_funcție;**

Exemple:

```
int b, a=9;  
double c;  
b=a+9;  
cout<<a;  
c=sqrt(a);  
clrcsr(); //apelul funcției predefinite care șterge ecranul; prototipul în headerul conio.h
```

□ Instrucțiunea compusă (instrucțiunea **bloc**)

```
Sintaxa: {  
.  
.  
declaratii;  
instr1;  
.  
instr2;  
.  
.  
}
```

Într-un bloc se pot declara și variabile care pot fi accesate doar în corpul blocului. Instrucțiunea bloc este utilizată în locurile în care este necesară prezența unei singure instrucțiuni, însă procesul de calcul este mai complex, deci trebuie descris în mai multe secvențe.

2. Implementarea structurii de decizie (alternative, de selecție)

Reprezentarea prin schemă logică și prin pseudocod a structurilor de decizie și repetitive sunt descrise în **capitolul 1**. Se vor prezenta în continuare doar instrucțiunile care le implementează.

□ Instrucțiunea **if**:

Sintaxa:

```
if (expresie)  
    instrucțiune1;  
[ else  
    instrucțiune2; ]
```

Ramura **else** este opțională.

La întâlnirea instrucțiunii **if**, se evaluează *expresie* (care reprezintă o condiție) din paranteze. Dacă valoarea expresiei este 1 (condiția este îndeplinită) se execută *instrucțiune1*; dacă valoarea expresiei este 0 (condiția nu este îndeplinită), se execută *instrucțiune2*. Deci, la un moment dat, se execută doar una dintre cele două instrucțiuni: *fie* instrucțiune1, *fie* instrucțiune2. După execuția

instrucțiunii if se trece la execuția instrucțiunii care urmează acesteia.

Observații:

1. Instrucțiunile 1 și instrucțiunea 2 pot fi instrucțiuni compuse (blocuri), sau chiar alte instrucțiuni if (if-uri imbricate).
2. Deoarece instrucțiunea if testează valoarea numerică a expresiei (condiției), este posibilă prescurtarea: if (expresie), în loc de if (expresie != 0).
3. Deoarece ramura else a instrucțiunii if este opțională, în cazul în care aceasta este omisă din secvențele if-else imbricate, se produce o ambiguitate. De obicei, ramura else se asociază ultimei instrucțiuni if.

Exemplu:

```
if (n>0)
    if (a>b)
        z=a;
    else z=b;
```

4. Pentru claritatea programelor sursă se recomandă alinierea instrucțiunilor prin utilizarea tabulatorului orizontal.

5. Deseori, apare construcția:

```
if (expresie1)
    instrucțiune1;
else
    if (expresie2)
        instrucțiune2;
    else
        if (expresie3)
            instrucțiune3;
        . . . . .
        else
            instrucțiune_n;
```

Aceeași construcție poate fi scrisă și astfel:

```
if (expresie1)
    instrucțiune1;
else if (expresie2)
    instrucțiune2;
else if (expresie3)
    instrucțiune3;
. . . . .
else
    instrucțiune_n;
```

Expresiile sunt evaluate în ordine; dacă una dintre expresii are valoarea 1, se execută instrucțiunea corespunzătoare și se termină întreaga înlănțuire. Ultima parte a lui else furnizează cazul când nici una dintre expresiile 1,2,..., n-1 nu are valoarea 1.

6. În cazul în care instrucțiunile din cadrul if-else sunt simple, se poate folosi operatorul condițional.

Exerciții:

1. Să se citească de la tastatură un număr real. Dacă acesta se află în intervalul [-1000, 1000], să se afișeze 1, dacă nu, să se afișeze -1.

```
#include <iostream.h>
void main()
{
    double nr; cout<<"Aștept numar:"; cin>>nr;
    int afis = (nr>= -1000 && nr <= 1000 ? 1 : -1); cout<<afis;
    /* int afis;
    if (nr >= -1000 && nr <= 1000)
        afis = 1;
    else afis= -1;
    cout<<afis; */
}
```

2. Să se calculeze valoarea funcției $f(x)$, știind că x este un număr real introdus de la tastatură:

$$f(x) = \begin{cases} -6x + 20 & , & \text{dacă } x \in [-\infty, -7] \\ x + 30 & , & \text{dacă } x \in (-7, 0] \\ \sqrt{x} & , & \text{dacă } x > 0 \end{cases}$$

```
#include <iostream.h>
#include <math.h>
void main()
{
double x,f;cout<<"x=";cin>>x;
if (x <= -7)
    f= -x* 6 +20;
else
    if ( x<=0 )
        f= x+30;
    else f=sqrt(x);
cout<<"f="<<f<<"\n";
}
```

Sau:

```
#include <iostream.h>
#include <math.h>
void main()
{
double x,f;cout<<"x=";cin>>x;
if (x <=-7)
    f= -x* 6 +20;
if (x>=-7 && x<=0 )
    f= x+30;
if (x>0) f=sqrt(x);
    cout<<"f="<<f<<"\n";
}
```

Uneori, construcția if-else este utilizată pentru a compara valoarea unei variabile cu diferite valori constante, ca în programul următor:

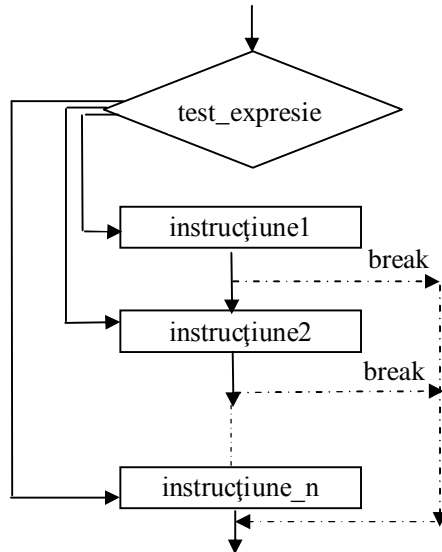
3. Se citește un caracter reprezentând un operator aritmetic binar simplu. În funcție de caracterul citit, se afișează numele operației pe care acesta o poate realiza.

```
#include <iostream.h>
void main()
{
char oper;
cout<<"Introdu operator aritmetic, simplu, binar:"; cin>>oper;
if (oper == '+')
    cout<<"Operatorul de adunare!\n";
else if (oper=='-')
    cout<<"Operatorul de scadere!\n";
else if (oper=='*')
    cout<<"Operatorul de inmultire!\n";
else if (oper=='/')
    cout<<"Operatorul de impartire!\n";
else if (oper=='%')
    cout<<"Operatorul rest!\n";
else cout<<"Operator ilegal!!!\n";
}
```

- Instrucțiunea **switch**. În unele cazuri este necesară o decizie multiplă specială. Instrucțiunea **switch** permite acest lucru.

□

Reprezentare prin schema logică (figura 2):



Reprezentare prin pseudocod:

```

Dacă expresie=expr_const_1
    instrucțiune1;
    [ieșire;]
Altfel dacă expresie=expr_const_2
    instrucțiune2;
    [ieșire;]
. . . . .
Altfel dacă expresie=expr_const_n-1
    instrucțiune_n-1;
    [ieșire;]
Altfel instrucțiune_n;

```

Figura 2. Decizia multiplă.

Se testează dacă valoarea pentru *expresie* este una dintre constantele specificate (*expr_const_1*, *expr_const_2*, etc.) și se execută instrucțiunea de pe ramura corespunzătoare. În schema logică *test_expresie* este una din condițiile: *expresie=expr_const_1*, *expresie=expr_const_2*, etc.

Sintaxa:

```

switch (expresie)
{
case expresie_const_1:    instructiune_1;
                        [break;]
case expresie_const_2:    instructiune_2;
                        [break;]
. . . . .
case expresie_const_n-1:  instructiune_n-1;
                        [break;]
[ default: instructiune_n; ]
}

```

Este evaluată *expresie* (expresie aritmetică), iar valoarea ei este comparată cu valoarea expresiilor constante 1, 2, etc. (expresii constante=expresii care nu conțin variabile). În situația în care valoarea *expresie* este egală cu valoarea *expr_const_k*, se execută instrucțiunea corespunzătoare acelei ramuri (*instrucțiune_k*). Dacă se întâlnește instrucțiunea **break**, parcurgerea este întreruptă, deci se va trece la execuția primei instrucțiuni de după switch. Dacă nu este întâlnită instrucțiunea *break*, parcurgerea continuă. Break-ul cauzează deci, ieșirea imediată din switch.

În cazul în care valoarea expresiei nu este găsită printre valorile expresiilor constante, se execută cazul marcat cu eticheta **default** (când acesta există). Expresiile *expresie*, *expresie_const_1*, *expresie_const_2*, etc., trebuie să fie

întregi. În exemplul următor, ele sunt de tip `char`, dar o dată de tip `char` este convertită automat în tipul `int`.

Exercițiu: Să rescriem programul pentru problema 3, utilizând instrucțiunea `switch`.

```
#include <iostream.h>
void main()
{
    char oper;
    cout<<"Introdu operator aritmetic, simplu, binar:";
    cin>>oper;
    switch (oper)
    {
        case ('+'):
            cout<<"Operatorul de adunare!\n";
            break;
        case ('-'):
            cout<<"Operatorul de scadere!\n";
            break;
        case ('*'):
            cout<<" Operatorul de inmultire!\n";
            break;
        case ('/'):
            cout<<"Operatorul de impartire!\n";
            break;
        case ('%'):
            cout<<"Operatorul rest!\n";
            break;
        default:
            cout<<"Operator ilegal!\n";
    }
}
```

3. Implementarea structurilor repetitive (ciclice)

Există două categorii de instrucțiuni ciclice: cu test inițial și cu test final.

3.1. Implementarea structurilor ciclice cu test inițial. Structura ciclică cu test inițial este implementată prin instrucțiunile ***while*** și ***for***.

□ Instrucțiunea ***while***

Sintaxa:

```
while (expresie)
    instructiune;
```

La întâlnirea acestei instrucțiuni, se evaluează *expresie*. Dacă aceasta are valoarea 1 - sau diferită de 0 - (condiție îndeplinită), se execută *instructiune*. Se revine apoi în punctul în care se evaluează din nou valoarea expresiei. Dacă ea este tot 1, se repetă *instructiune*, ș.a.m.d. Astfel, instrucțiunea (corpul ciclului) *se repetă atât timp cât expresie are valoarea 1*. În momentul în care *expresie* ia valoarea 0 (condiție neîndeplinită), se iese din ciclu și se trece la următoarea instrucțiune de după *while*.

Observații:

1. În cazul în care la prima evaluare a expresiei, aceasta are valoarea zero, corpul instrucțiunii *while* nu va fi executat niciodată.
2. Instrucțiune din corpul ciclului *while* poate fi compusă (un bloc), sau o altă instrucțiune ciclică.

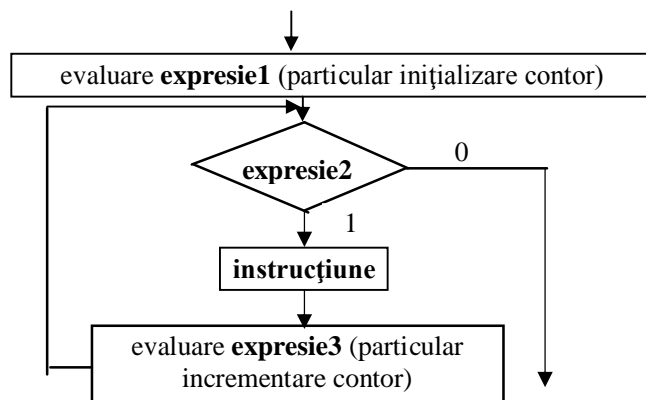
3. Este de dorit ca instrucțiunea din corpul ciclului `while` să modifice valoarea expresiei. Dacă nu se realizează acest lucru, corpul instrucțiunii `while` se repetă de un număr infinit de ori.

Exemplu:

```
int a=7;
while (a==7)
    cout<<"Buna ziua!\n";    // ciclu infinit; se repetă la infinit afișarea
                                mesajului
```

- Instrucțiunea **for**. În majoritatea limbajelor de programare de nivel înalt, instrucțiunea `for` implementează structura ciclică cu număr cunoscut de pași (vezi reprezentarea prin schema logică și pseudocod din **capitolul 1**). În limbajul C instrucțiunea `for` poate fi utilizată într-un mod mult mai flexibil.

Reprezentare prin schema logică (figura 3):



Reprezentare în pseudocod:

```
evaluare expresie1
CÂT TIMP expresie2
  REPETĂ
    ÎNCEPUT
    instrucțiune
    evaluare expresie3
  SFÂRȘIT
```

Figura 3. Structura ciclică cu test inițial.

Sintaxa:

```
for (expresie1; expresie2; expresie3)
    instrucțiune;
```

Nu este obligatorie prezența expresiilor, ci doar a instrucțiunilor vide.

Exemplu:

```
for (; expresie2; )           sau:           for (; )
    instrucțiune;                instrucțiune;
```

3.2. Implementarea structurilor ciclice cu test final.

- Instrucțiunea **do-while**

Sintaxa:

```
do    instrucțiune;
while(expresie);
```

Se execută instrucțiune. Se evaluează apoi expresie. Dacă aceasta are valoarea 1, se execută instrucțiune. Se testează din nou valoarea expresiei. Se repetă instrucțiune *cât timp* valoarea expresiei este 1 (condiția este îndeplinită). În cazul instrucțiunii `do-while`, corpul ciclului se execută cel puțin o dată.

Exerciții:

1. Se citește câte un caracter, până la întâlnirea caracterului `@`. Pentru fiecare caracter citit, să se afișeze un mesaj care să indice dacă s-a citit o literă mare, o literă mică, o cifră sau un alt caracter. Să se afișeze câte litere mari au fost introduse, câte litere mici,

câte cifre și câte alte caractere. Se prezintă trei modalități de implementare (cu instrucțiunea `while`, cu instrucțiunea `for` și cu instrucțiunea `do-while`).

```
#include <iostream.h>
#include <conio.h>
void main()
{ char c; clrscr();
int lmic=0, lmare=0, lcif=0;
int altcar=0;
cout<<"Aștept car.: "; cin>>c;
while (c!='@'){
    if (c>='A' && c<='Z') {
        cout<<"Lit. mare!\n";
        lmare++; }
    else if (c>='a' && c<='z') {
        cout<<"Lit. mică!\n";
        lmic++; }
    else if (c>='0' && c<='9') {
        cout<<"Cifră!\n";
        lcif++; }
    else {
        cout<<"Alt car.!\n";
        altcar++; }
    cout<<"Aștept car.: ";cin>>c;
}
cout<<"Ați introdus \n";
cout<<lmare<<" litere mari, ";
cout<<lmic<<" litere mici\n";
cout<<lcif<<" cifre și \n";
cout<<altcar<<" alte caractere\n";
getch(); }
```

```
#include <iostream.h>
#include <conio.h>
void main()
{ char c;clrscr();
int lmic=0,lmare=0,lcif=0;int altcar=0;
cout<<"Aștept caract.: "; cin>>c;
for( ; c!='@'; ){
    //corp identic
}
cout<<"Ați introdus \n";
cout<<lmare<<" litere mari, ";
cout<<lmic<<" litere mici\n";
cout<<lcif<<" cifre și \n";
cout<<altcar<<" alte caractere\n";
getch(); }
```

Observații legate de implementare

Variabila `c` (tip `char`) memorează caracterul introdus la un moment dat, de la tastatură.

Variabilele întregi `lmic`, `lmare`, `lcif` și `altcar` sunt utilizate pe post de contor pentru litere mari, mici, cifre, respectiv alte caractere.

Acțiunea care se repetă cât timp caracterul citit este diferit de constanta caracter '@' constă din mai multe acțiuni simple: citirea unui caracter (cu afișarea în prealabil a mesajului "Aștept car.:"; testarea caracterului citit (operatorii relaționali pot fi aplicați datelor de tip `char`).

Ca urmare, acțiunea din corpul instrucțiunii `while` a fost implementată printr-o instrucțiune bloc.

Tot instrucțiuni bloc au fost utilizate pe fiecare ramură a instrucțiunii `if` (afișare mesaj referitor la caracter și incrementare contor).

Pentru implementarea aceluiași algoritm se poate utiliza instrucțiunea `for`. În cadrul acesteia, expresiile 1 și expresia 3 lipsesc, însă prezența instrucțiunilor `vide` este obligatorie.

O altă variantă de implementare poate fi următoarea, în care și inițializarea variabilelor contor se realizează în cadrul expresiei `expresie1`.

```
int lmic, lmare, lcif, altcar;
```

```

for(lmare=0, lmic=0, lcif=0, altcar=0; c!='@'; ){
    //corp identic
}

```

Varianta de implementare care utilizează instrucțiunea do-while:

```

int lmic=0, lmare=0, lcif=0;
int altcar=0;
cout<<"Aștept caract.: "; cin>>c;
do {
    //corp do-while
} while (c!='@');
cout<<"Ați introdus \n";
//...

```

2. Să se calculeze suma și produsul primelor n numere naturale, n fiind introdus de la tastatură. Se vor exemplifica modalitățile de implementare cu ajutorul instrucțiunilor

do-while, while, și for. (Se observă că: $S = \sum_{k=1}^n k$, $P = \prod_{k=1}^n k$).

<pre> cout<<"n="; int n; cin>>n; int S=0, P=1, k=1; while (k <= n){ S+=k; P*=k; k++; } cout<<"P="<<P<<"\tS="<<S<<"\n"; </pre>	<pre> cout<<"n="; int n; cin>>n; int S=0, P=1, k=1; do{ S+=k; P*=k; k++; } while (k <= n); cout<<"P="<<P<<"\tS="<<S<<"\n"; </pre>
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Pentru a ilustra multiplele posibilități oferite de instrucțiunea for, se prezintă variantele

<pre> // varianta1 int S=0, P=1, k; for (k=1; k<=n; k++){ S+=k; P*=k; } cout<<"P="<<P<<"\tS="; cout<<S<<"\n"; </pre>	<pre> // varianta2 int S=0, P=1; for (int k=1; k<=n; k++){ S+=k; P*=k; } cout<<"P="<<P<<"\tS="; cout<<S<<"\n"; </pre>
<pre> // varianta3 for (int S=0, P=1, k=1; k<=n; k++){ S+=k; P*=k; } cout<<"P="<<P<<"\tS="<<cout<<S<<"\n"; </pre>	
<pre> // varianta4 for (int S=0, P=1, k=1; k<=n; S+=k, P*=k, k++) ; cout<<"P="<<P<<"\tS="<<cout<<S<<"\n"; </pre>	

3. Să se citească un șir de numere reale, până la întâlnirea numărului 900. Să se afișeze maximul numerelor citite.

```
#include <iostream.h>
void main()
{double n;
cout<<"Introdu nr:"; cin>>n;
double max=n;
while (n!=900)
{    if (n>=max)
        max=n;
    cout<<"Introdu nr:";
    cin>>n;
}
cout<<"Max șir este:"<<max<<'\n';
}
```

Se presupune că primul element din șirul de numere are valoarea maximă. Se memorează valoarea sa în variabila max. Se parcurge apoi șirul, comparându-se valoarea fiecărui element cu valoarea variabilei max. În cazul în care se găsește un element cu o valoare mai mare decât a variabilei max, se reține noua valoare (max=n).

4. Să se afișeze literele mari ale alfabetului și codurile aferente acestora în ordine crescătoare, iar literele mici și codurile aferente în ordine descrescătoare. Afișarea se va face cu pauză după fiecare ecran.

```
#include <iostream.h>
#include <conio.h>
#define DIM_PAG 22 //dimensiunea paginii (numarul de randuri de pe o pagina)
void main()
{clrscr();
cout<<"LITERELE MARI:\n";int nr_lin=0; // nr_lin este contorul de linii de pe un ecran
for (char LitMare='A'; LitMare<='Z'; LitMare++){
    if (nr_lin==DIM_PAG){
        cout<<"Apasa o tasta...."; getch(); clrscr(); nr_lin=0;}
    cout<<"Litera "<<LitMare<<" cu codul ASCII "<<(int)LitMare<<'\n';
    // conversia explicita (int)LitMare permite afisarea codului ASCII al caracterului
    nr_lin++;
}
cout<<"LITERELE MICI:\n";
for (char LitMica='z'; LitMica>='a'; LitMica--){
    if (nr_lin==DIM_PAG){
        cout<<"Apasa o tasta...."; getch(); clrscr(); nr_lin=0;}
    cout<<"Litera "<<LitMica<<" cu codul ASCII "<<(int)LitMica<<'\n';
    nr_lin++;
}
}
```

5. Să se scrie un program care realizează conversia numărului N întreg, din baza 10 într-o altă bază de numerație, $b < 10$ (N și b citite de la tastatură). Conversia unui număr întreg din baza 10 în baza b se realizează prin împărțiri succesive la b și memorarea resturilor, în ordine inversă. De exemplu:

$547:8=68 \text{ rest } 3; 68:8=8 \text{ rest } 4; 8:8=1 \text{ rest } 0; 1:8=0 \text{ rest } 1 \quad 547_{10} = 1043_8$

```
#include <iostream.h>
void main()
{ int nrcif=0,N,b,rest,Nv,p=1;
long Nnou=0;
cout<<"\nIntroduceti baza<10, b=";cin>>b;
cout<<"Introduceti numarul in baza 10, nr=";cin>>N;
```

```

Nv=N;
while(N!=0)
{
    rest=N%b;    N/=b;    cout<<"nr="<<N<<"\n";    cout<<"rest="<<rest<<"\n";
    nrcif++;    Nnou+=rest*p;    p*=10;    cout<<"Nr. nou="<<Nnou<<"\n";
}
cout<<"Numarul de cifre este " <<nrcif<<"\n";    cout<<"Nr. in baza 10
"<<Nv;
cout<<" convertit in baza " <<b<<" este " <<Nnou<<"\n";    }

```

6. Să se calculeze seria următoare cu o eroare mai mică decât EPS (EPS introdus de la tastatură): $1 + \sum_{k=1}^{\infty} \frac{x^k}{k}$, $x \in [0,1]$, x citit de la tastatură. Vom aduna la sumă încă un termen cât timp diferența dintre suma calculată la pasul curent și cea calculată la pasul anterior este mai mare sau egală cu EPS.

```

#include <iostream.h>
#include <conio.h>
#include <math.h>
void main()
{ double T,S,S1; long k;k=1;T=1;S=T;double x; cout<<"x="; cin>>x;
// T= termenul general de la pasul curent; S=suma la pasul curent; S1=suma la pasul anterior
do {
    S1=S;k=k+1;T=pow(x,k)/k; //funcția pow(x, k), aflată în <math.h> calculează  $x^k$ 
    S=S+T; // cout<<k<<" "<<T<<" "<<S<<"\n";getch();
} while ((S-S1)>=EPS);
cout<<"Nr termeni="<<k<<"    T="<<T<<"    S="<<S<<"\n"; }

```

4. Facilități de întrerupere a unei secvențe

Pentru o mai mare flexibilitate (tratarea excepțiilor care pot apare în procesul de prelucrare), în limbajul C se utilizează instrucțiunile **break** și **continue**. Ambele instrucțiuni sunt utilizate în instrucțiunile ciclice. În plus, instrucțiunea **break** poate fi folosită în instrucțiunea **switch**.

- Instrucțiunea **break**. Așa cum se observă din figura 4, utilizată în cadrul instrucțiunilor ciclice, instrucțiunea **break** "forțează" ieșirea din acestea. Fără a se mai testa valoarea expresiei (condiția) care determină repetarea corpului instrucțiunii ciclice, se continuă execuția cu instrucțiunea care urmează instrucțiunii ciclice. Astfel, se întrerupe repetarea corpului instrucțiunii ciclice, indiferent de valoarea condiției de test. Utilizarea în cadrul instrucțiunii **switch**: În situația în care s-a ajuns la o valoare a unei expresiei constante egală cu cea a expresiei aritmetice, se execută instrucțiunea corespunzătoare acelei ramuri. Dacă se întâlnește instrucțiunea **break**, parcurgerea este întreruptă (nu se mai compară valoarea expresiei aritmetice cu următoarele constante), deci se va trece la execuția primei instrucțiuni de după **switch**. Dacă nu este întâlnit **break**, parcurgerea continuă. Instrucțiunea **break** cauzează deci, ieșirea imediată din **switch**.
- Instrucțiunea **continue**. Întâlnirea instrucțiunii **continue** (figura 4) determină ignorarea instrucțiunilor care o urmează în corpul instrucțiunii ciclice și reluarea execuției cu testarea valorii expresiei care determină repetarea sau nu a corpului ciclului.

Exemplu: Să revenim la programul realizat pentru problema 1, care folosește instrucțiunea `dowhile`. Dacă primul caracter citit este chiar caracterul `@`, se realizează testarea acestuia; ca urmare, se afișează mesajul "Alt car.!" și se incrementează valoarea contorului `altcar`. Dacă nu se dorește ca acest caracter să fie testat și numărat, în corpul instrucțiunii `do while` putem face un test suplimentar.

```
int lmic=0, lmare=0, lcif=0, altcar=0; cout<<"Aștept caract.:"; cin>>c;
do {
    if (c == '@')        break;        //ieșire din do while
    //corp do-while
} while (c!='@');
cout<<"Ați introdus \n";
//. . .
```

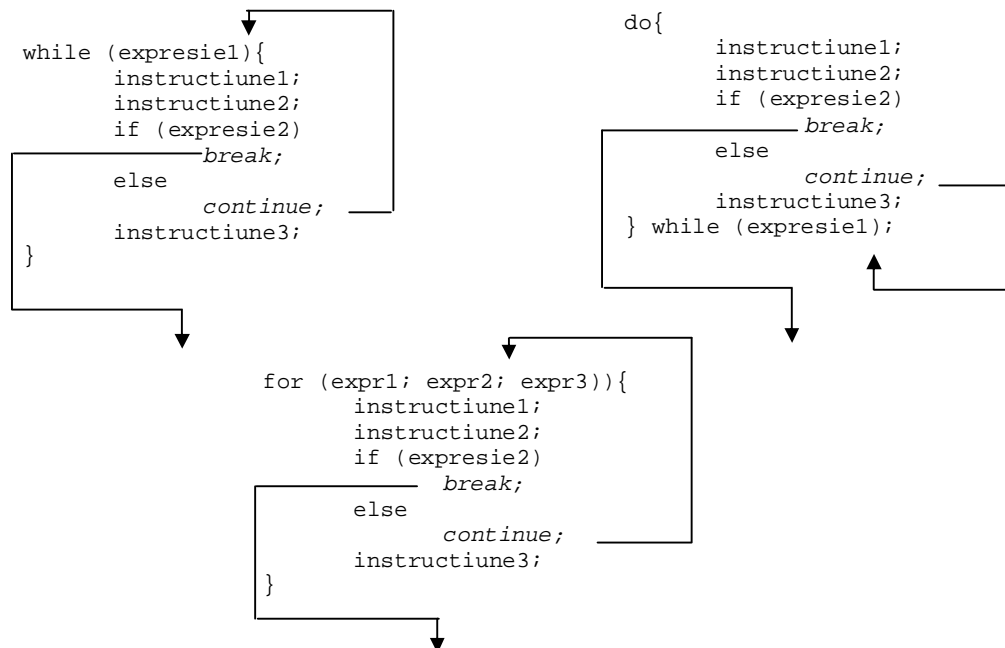


Figura 4. Modul de utilizare a instrucțiunilor `break` și `continue`.

Întrebări și exerciții

Întrebări teoretice

1. Care sunt instrucțiunile care implementează în limbajul C structura condițională?
2. Care sunt instrucțiunile care implementează în limbajul C structura secvențială?
3. Care sunt instrucțiunile care implementează în limbajul C structura repetitivă cu test inițial?
4. Care sunt instrucțiunile care implementează în limbajul C structura repetitivă cu test final?
5. Ce deosebiri sunt între instrucțiunea while și instrucțiunea do-while?
6. Pornind de la sintaxa instrucțiunii for, stabiliți echivalența între aceasta și instrucțiunile while și do-while?

Exerciții practice

1. Să se implementeze programele cu exemplele prezentate.
2. Să se scrie programele pentru exercițiile rezolvate care au fost prezentate.
3. Să se implementeze algoritmi proiectați pentru problemele 1-7 din **capitolul 1**.
4. Să se calculeze aria unui triunghi, cunoscându-se mărimea laturilor sale. Numerele care reprezintă mărimile laturilor vor fi introduse de utilizator. Se va testa mai întâi dacă cele 3 numere reprezentând mărimea laturilor pot forma un triunghi ($a \leq b+c$, $b \leq c+d$, $c \leq a+b$).
5. Să se rescrie următoarea secvență, folosind o singură instrucțiune if.

```
if (n<0)
    if (n>=90)
        if (x!=0)
            int b= n/x;
```
6. Să se citească un număr natural n. Să se scrie un program care afișează dacă numărul n citit reprezintă sau nu, un an bisect (anii bisești sunt multipli de 4, exceptând multiplii de 100, dar incluzând multiplii de 400).
7. Să se găsească toate numerele de două cifre care satisfac relația:
$$\overline{xy} = (x + y)^2$$
8. Să se citească un șir de numere reale, până la întâlnirea numărului 800 și să se afișeze valoarea minimă introdusă, suma și produsul elementelor șirului.
9. Scrieți un program care să verifice inegalitatea $1/(n+1) < \ln[(n+1)/n] < 1/n$, unde n este un număr natural pozitiv, introdus de la tastatură.
10. Fie funcția
$$f(x) = \begin{cases} e^{x-3} & , x \in [0, 1) \\ \sin x + \cos x & , x \in [1, 2) \\ 0,9 \ln(x+3) & , x \in [2, 100] \end{cases}$$

Să se calculeze f(x), x citit de la tastatură.
11. Să se scrie un program care calculează și afișează maximul a 3 numere reale (a, b și c) citite de la tastatură.
12. Să se scrie un program care calculează și afișează minimul a 3 numere reale (a, b și c) citite de la tastatură.
13. Să se citească 2 caractere care reprezintă 2 litere mari. Să se afișeze caracterele citite în ordine alfabetică.
14. Să se citească 3 caractere care reprezintă 3 litere mici. Să se afișeze caracterele citite

în ordine alfabetică.

15. Să se scrie un program care citește o cifră. În funcție de valoarea ei, să se facă următorul calcul: dacă cifra este 3, 5 sau 7 să se afișeze pătratul valorii numerice a cifrei; dacă cifra este 2, 4 sau 6 să se afișeze cubul valorii numerice a cifrei; dacă cifra este 0 sau 1 să se afișeze mesajul "Valori mici"; altfel., să se afișeze mesajul "Caz ignorat!".
16. Fie șirul lui Fibonacci, definit astfel:
 $f(0)=0, f(1)=1, f(n)=f(n-1)+f(n-2)$ în cazul în care $n>1$.
Să se scrie un program care implementează algoritmul de calcul al șirului Fibonacci.
17. Să se calculeze valoarea polinomului Cebîșev de ordin n într-un punct x dat, cunoscând relația:
 $T_0(x)=1, T_1(x)=x$ și $T_{k+1}(x) - 2xT_k(x) + T_{k-1}(x) = 0$
18. Să se citească câte 2 numere întregi, până la întâlnirea perechii (0, 0). Pentru fiecare pereche de numere, să se calculeze și să se afișeze cel mai mare divizor comun.
19. Se citesc câte 3 numere reale, până la întâlnirea numerelor 9, 9, 9. Pentru fiecare triplet de numere citit, să se afișeze maximum.
20. Se citește câte un caracter până la întâlnirea caracterului @. Să se afișeze numărul literelor mari, numărul literelor mici și numărul cifrelor citite; care este cea mai mare (lexicografic) literă mare, literă mică și cifră introdusă.
21. Se citesc câte 2 numere întregi, până la întâlnirea perechii de numere 9, 9. Pentru fiecare pereche de numere citite, să se afișeze cel mai mare divizor comun al acestora.
22. Să se calculeze suma seriei
$$1 + x^3/3 - x^5/5 + x^7/7 - \dots$$
cu o eroare mai mică decât epsilon (epsilon citit de la tastatură). Să se afișeze și numărul de termeni ai sumei.
23. Să se citească un număr întreg format din 4 cifre (abcd). Să se calculeze și să se afișeze valoarea expresiei reale: $4*a + b/20 - c + 1/d$.
24. Să se scrie un program care afișează literele mari ale alfabetului în ordine crescătoare, iar literele mici - în ordine descrescătoare.
25. Să se scrie un program care generează toate numerele perfecte până la o limită dată, LIM. Un număr perfect este egal cu suma divizorilor lui, inclusiv 1 (exemplu: $6=1+2+3$).
26. Să se calculeze valoarea sumei urmatoare, cu o eroare EPS mai mică de 0.0001:
 $S=1+(x+1)/2! + (x+2)/3! + (x+3)/4! + \dots$, unde $0 \leq x \leq 1$, x citit de la tastatură.
27. Să se genereze toate numerele naturale de 3 cifre pentru care cifra sutelor este egală cu suma cifrelor zecilor și unităților.
28. Să se citească câte un număr întreg, până la întâlnirea numărului 90. Pentru fiecare număr să se afișeze un mesaj care indică dacă numărul este pozitiv sau negativ. Să se afișeze cel mai mic număr din șir.
29. Să se genereze toate numerele naturale de 3 cifre pentru care cifra zecilor este egală cu diferența cifrelor sutelor și unităților.
30. Să se calculeze suma:
$$(1 + 2!) / (2 + 3!) - (2+3!) / (3+4!) + (3+4!) / (4+5!) - \dots$$

- | | |
|-----------------------------|----------------------------|
| 1. Declararea tablourilor | 3. Tablouri bidimensionale |
| 2. Tablouri unidimensionale | 4. Șiruri de caractere |
-

1. Declararea tablourilor

Se numește tablou o colecție (grup, mulțime ordonată) de date, de același tip, situate într-o zonă de memorie continuă (elementele tabloului se află la adrese succesive). Tablourile sunt *variabile compuse (structurate)*, deoarece grupează mai multe elemente. Variabilele tablou au nume, iar tipul tabloului este dat de tipul elementelor sale. Elementele tabloului pot fi referite prin numele tabloului și indicii (numere întregi) care reprezintă poziția elementului în cadrul tabloului.

În funcție de numărul indicilor utilizați pentru a referi elementele tabloului, putem întâlni tablouri *unidimensionale* (vectorii) sau *multidimensionale* (matricile sunt tablouri bidimensionale).

Ca și variabilele simple, variabilele tablou trebuie declarate înainte de utilizare.

Modul de declarare:

```
tip    nume_tablou[dim_1][dim_2]...[dim_n];
```

unde: *tip* reprezintă tipul elementelor tabloului; *dim_1, dim_2, ..., dim_n* sunt numere întregi sau expresii constante întregi (a căror valoare este evaluată la compilare) care reprezintă limitele superioare ale indicilor tabloului.

Exemple:

```
//1
int vect[20];          // declararea tabloului vect, de maximum 20 de elemente, de tipul int.
                        // Se rezervă 20*sizeof(int)=20 * 2 = 40 octeți

//2
double p,q,tab[10];
    // declararea variabilelor simple p, q și a vectorului tab, de maximum 10 elemente, tip double

//3
#define MAX 10
char tabc[MAX];        /*declararea tabloului tabc, de maximum MAX (10) elemente de tip char*/

//4
double matrice[2][3];   // declararea tabloului matrice (bidimensional),
                        // maximum 2 linii și maximum 3 coloane, tip double
```


2. Tablouri unidimensionale

Tablourile unidimensionale sunt tablouri cu un singur indice (vectori). Dacă tabloul conține `dim_1` elemente, indicii elementelor au valori întregi din intervalul $[0, \text{dim_1}-1]$.

La întâlnirea declarației unei variabile tablou, compilatorul alocă o zonă de memorie continuă (dată de produsul dintre dimensiunea maximă și numărul de octeți corespunzător tipului tabloului) pentru păstrarea valorilor elementelor sale. Numele tabloului poate fi utilizat în diferite expresii și valoarea lui este chiar adresa de început a zonei de memorie care i-a fost alocată. Un element al unui tablou poate fi utilizat ca orice altă variabilă (în exemplul următor, atribuirea de valori elementelor tabloului vector). Se pot efectua operații asupra fiecărui element al tabloului, nu asupra întregului tablou.

Exemplu:

```
// Declararea tabloului vector
int vector[6];
```

// Inițializarea elementelor tabloului

```
vector[0]=100;
vector[1]=101;
vector[2]=102;
vector[3]=103;
vector[4]=104;
vector[5]=105;
```

Exemplu:

```
double alpha[5], beta[5], gama[5];
int i=2;
alpha[2*i-1] = 5.78;
alpha[0]=2*beta[i]+3.5;
gama[i]=alpha[i]+beta[i]; //permis
gama=alpha+beta;          //nepermis
```

vector

100	vector[0]
101	vector[1]
102	vector[2]
103	vector[3]
104	vector[4]
105	vector[5]

Figura 1.

Variabilele tablou pot fi inițializate în momentul declarării:

declarație_tablou=listă_valori;

Valorile din lista de valori sunt separate prin virgulă, iar întreaga listă este inclusă între acolade:

Exemple:

```
//1
int vector[6]={100,101,102,103,104,105};
```

vector	100	101	102	103	104	105
	[0]					[5]

//2

```
double x=9.8;
double a[5]={1.2, 3.5, x, x-1, 7.5};
```

La declararea unui vector cu inițializarea elementelor sale, numărul maxim de elemente ale tabloului poate fi omis, caz în care compilatorul determină automat mărimea tabloului, în funcție de numărul elementelor inițializate.

Exemplu:

```
char tab[]={ 'A', 'C', 'D', 'C'};
```

tab	'A'	'B'	'C'	'D'
	[0]			[3]

```
float data[5]={ 1.2, 2.3, 3.4 };
```

data	1.2	2.3	3.4	?	?
	[0]				[4]

Adresa elementului de indice i dintr-un tablou unidimensional poate fi calculată astfel:

$$adresa_elementului_i = adresa_de_bază + i * lungime_element$$

Exerciții:

//1. Citirea elementelor unui vector:

```
double a[5];
int i;
for (i=0; i<5; i++)
{
    cout<<"a["<<i<<" ]="; //afișarea unui mesaj prealabil citirii fiecărui element
    cin>>a[i];             //citirea valorii elementului de indice i
}
```

//Sau:

```
double a[20];      int i, n;
cout<<"Dim. Max. ="; cin>>n;
for (i=0; i<n; i++)
{
    cout<<"a["<<i<<" ]=";
    cin>>a[i];
}
```

//2. Afișarea elementelor unui vector:

```
cout<<"Vectorul introdus este:\n";
for (i=0; i<n i++)
    cout<<a[i]<<' ';
```

//3. Afișarea elementelor unui vector în ordine inversă:

```
cout<<"Elementele vectorului în ordine inversă:\n";
for (i=n-1; i>=0 i++)
    cout<<a[i]<<' ';
```

//3. Vectorul sumă (c) a vectorilor a și b, cu același număr de elemente:

```
for (i=0; i<n i++)
    c[i]=a[i]+b[i];
```

//4. Vectorul diferență (c) a vectorilor a și b, cu același număr de elemente:

```
for (i=0; i<n i++)
    c[i]=a[i] - b[i];
```

//5. Produsul scalar (p) a vectorilor a și b, cu același număr de elemente:

$$p = \sum_{i=0}^{n-1} a_i * b_i$$

```
double p=0;
for (i=0; i<n i++)
    p += a[i] * b[i];
```

3. Tablouri bidimensionale

Din punct de vedere conceptual, elementele unui tablou bidimensional sunt plasate în spațiu pe două direcții. Matricea reprezintă o aplicație naturală a tablourilor bidimensionale.

În matematică:

$$Q = \begin{Bmatrix} q_{11} & q_{12} & q_{13} & \dots & q_{1n} \\ q_{21} & q_{22} & q_{23} & \dots & q_{2n} \\ \dots & \dots & \dots & \dots & \dots \\ q_{m1} & q_{m2} & q_{m3} & \dots & q_{mn} \end{Bmatrix} \quad Q_{m \times n}$$

În limbajele C/C++ (indicii de linie și de coloană pornesc de la 0):

$$Q = \begin{Bmatrix} q_{00} & q_{01} & q_{02} & \dots & q_{0,n-1} \\ q_{10} & q_{11} & q_{12} & \dots & q_{1,n-1} \\ \dots & \dots & \dots & \dots & \dots \\ q_{m-1,0} & q_{m-1,1} & q_{m-1,2} & \dots & q_{m-1,n-1} \end{Bmatrix} \quad Q_{m \times n}$$

Exemplu:

```
double q[3][2]; // declararea matricii q, cu maxim 3 linii și 2 coloane, tip double
```

În memorie, elementele unei matrici sunt memorate pe linii:

$q_{00} \quad q_{01} \quad q_{10} \quad q_{11} \quad q_{20} \quad q_{21} \quad \dots$

Dacă notăm cu k poziția în memorie a unui element, valoarea lui $k = i * m + j$ (unde m este numărul maxim de linii, i este indicele de linie, j este indicele de coloană).

$\boxed{q[0][0]} \boxed{q[0][1]} \boxed{q[0][2]} \dots \boxed{q[0][n-1]} \boxed{q[1][0]} \dots \boxed{q[m-1][0]} \dots \boxed{q[m-1][n-1]}$

Dacă se dorește *inițializarea elementelor unei matrici în momentul declarării acesteia*, se poate proceda astfel:

```
int mat[4][3] = {
    {10, -50, 3},
    {32, 20, 1},
    {-1, 1, -2},
    {7, -8, 19} };
```

Prin această construcție, elementele matricii `mat` se inițializează în modul următor:

```
mat[0][0]=10, mat[0][1]=-50, mat[0][2]=3
mat[1][0]=32, mat[1][1]=20, mat[1][2]=1
mat[2][0]=-1, mat[2][1]=1, mat[2][2]=-2
mat[3][0]=7, mat[3][1]=-8, mat[3][2]=19
```

La declararea unei matrici și inițializarea elementelor sale, se poate omite numărul maxim de linii, în schimb, datorită modului de memorare, trebuie specificat numărul maxim de coloane:

```
int mat[][3] = {
    {10, -5, 3},
    {32, 20, 1},
    {-1, 1, -2},
    {7, -8, 9} };
```

Construcția are același efect ca precedenta.

```
int mat[][3] = {
```

```
{1, 1},
{-1},
{3, 2, 1}};
```

mat reprezintă o matrice 3×3 , ale cărei elemente se inițializează astfel:

```
mat[0][0]=1, mat[0][1]=1, mat[1][0]=-1, mat[2][0]=3, mat[2][1]=2, mat[2][2]=1
```

Elementele `mat[0][2]`, `mat[1][1]`, `mat[1][2]` nu sunt inițializate. Ele au valoarea zero dacă tabloul este global și valori inițiale nedefinite dacă tabloul este automatic.

Construcțiile utilizate la inițializarea tablourilor bidimensionale se extind pentru tablouri multidimensionale, cu mai mult de doi indici.

Exemplu:

```
int a[2][2][3]={
    { {10, 20}, {1, -1}, {3, 4}},
    { {20, 30}, {50, -40}, {11, 12}}
};
```

Exercițiu: Să se citească de la tastatură elementele unei matrici de maxim 10 linii și 10 coloane. Să se afișeze matricea citită.

```
#include <iostream.h>
void main(void)
{int A[10][10]; int nr_lin, nr_col; cout<<"Nr. linii:"; cin>>nr_lin;
cout<<"Nr. coloane:"; cin>>nr_col;int i, j;
//citirea elementelor unei matrici
for (i=0; i<nr_lin; i++)
    for (j=0; j<nr_col; j++) {
        cout<<"A["<<i<<" "<<j<<"]="; //afișarea unui mesaj prealabil citirii
        cin>>A[i][j];
    }
//afișarea elementelor matricii
for (i=0; i<nr_lin; i++) {
    for (j=0; j<nr_col; j++)
        cout<<A[i][j]<<"\t";
    cout<<"\n"; // după afișarea elementelor unei linii, se trece pe linia următoare
}
}
```

4. Șiruri de caractere

Șirurile de caractere sunt *tablouri de caractere*, care au ca ultim element un terminator de șir, caracterul null (zero ASCII), `'\0'`.

Exemplu:

```
char tc[5] = {'a', 'b', 'c', 'd', 'e'}; // tablou de caractere
char sc[5] = {'a', 'b', 'c', 'd', '\0'}; // șirul de caractere cu elementele abcd
```

Limbajul C/C++ permite inițializarea unui tablou de caractere printr-o constantă șir (șir între ghilimele), care include automat caracterul null. Deci ultima inițializare este echivalentă cu:


```
char sc[5] = "abcd"; //sau cu
char sc[] = "abcd";
```

Exemplu:

```

char tc[5] = {'a', 'b', 'c', 'd', 'e'};
char sc[5] = {'a', 'b', 'c', 'd', '\0'};
char scl[5] = "abcd";
char s[10];
cout<<sc<<'\n';           //afișează abcd
cout<<tc<<'\n';
//eroare: tabloul de caractere nu conține terminatorul de șir, deci nu poate fi afișat ca șir
cout<<s<<'\n';             // eroare: tablou neinițializat
cout<<scl[2];               // afișează al treilea element din șirul scl
scl[1]='K';                 // elementului din șir de indice 1 i se atribuie valoarea 'K';

```

 **Funcții pentru operații cu șiruri de caractere.** Funcțiile pentru operații cu șiruri se găsesc în header-ul **<string.h>**.

strlen (nume_șir)

Returnează un număr întreg ce reprezintă lungimea unui șir de caractere, fără a număra terminatorul de șir.

strcmp (șir_1, șir_2)

Funcția compară cele două șiruri date ca argument și returnează o valoare întreagă egală diferența dintre codurile ASCII ale primelor caractere care nu coincid.

strcpy (șir_destinație, șir_sursă)

Funcția copie șirul sursă în șirul destinație. Pentru a fi posibilă copierea, lungimea șirului destinație trebuie să fie mai mare sau egală cu cea a șirului sursă, altfel pot apare erori grave.

strcat (șir_destinație, șir_sursă)

Funcția concatenează cele două șiruri: șirul sursă este adăugat la sfârșitul șirului destinație. Tabloul care conține șirul destinație trebuie să aibă suficiente elemente.

Exemplu:

```

#include <iostream.h>
#include <string.h>
void main()
{
char sir1[] = "abcd", sir2[] = "abcde", sir3 = "abcdef", sir4 = "de";
cout<<strcmp(sir1, sir2)<<'\n';           // afișare: -101
// 'e' = 101, 'a' = 97, 'd' = 100
// '0' - 'e' = -101
cout<<strcmp(sir2, sir1)<<'\n';           // afișare: 101
cout<<strcmp(sir1, "")<<'\n';             // compararea variabilei sir1 cu constanta șir vid
char str1[20]="hello";
char str2[20]="goodbye";
char str3[20];
int difer, lungime;
cout<<"str1="<<str1<<" str2="<<str2<<'\n';
difer=strcmp(str1, str2);
if (difer == 0)
    cout<<"Siruri echivalente!\n";
else if (difer>0)
    cout<<str1<<" mai mare (lexicografic) decât "<<str2<<'\n';
else
    cout<<str1<<" mai mic (lexicografic) decât "<<str2<<'\n';
cout<<"str1="<<str1<<'\n'; cout<<"str3="<<str3<<'\n';
strcpy (str3, str1); cout<<"str1="<<str1<<'\n';

```

```

cout<<"str3="<<str3<<'\\n';
strcat (str3, str1);
cout<<"str1="<<str1<<'\\n';
cout<<"str3="<<str3<<'\\n';
}

```

Exemplu: Să se citească elementele unui vector cu maxim 100 de elemente reale.

a) Să se interschimbe elementele vectorului în modul următor: primul cu ultimul, al doilea cu penultimul, etc.

b) Să se ordoneze crescător elementele vectorului.

```

// a)
#define FALSE    0
#define TRUE     1
#include <iostream.h>
void main()
{ double vect[100];int n;//n-numarul de elemente ale vectorului
cout<<"Nr. elemente"; cin>>n; double aux;
// de completat exemplul cu secventa de citire a elementelor vectorului
    for (int i=0; i<n/2; i++){
        aux=vect[i];
        vect[i]=vect[n-1-i];
        vect[n-1-i]=aux;
    }
// de completat exemplul cu secventa de afisare a vectorului
}

```

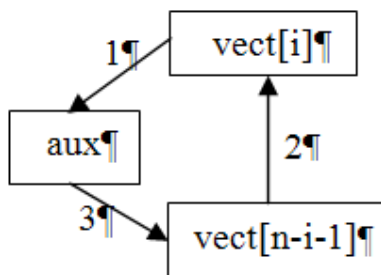


Figura 2. Interschimbarea a două variabile.

Pentru schimbarea elementelor vectorului s-a folosit variabila auxiliară aux (figura 2). Fără această variabilă, la atribuirea `vect[i]=vect[n-1-i]`, valoarea elementului `vect[i]` s-ar fi pierdut. Trebuie observat, de asemenea, că variabila contor `i` ia valori între 0 și `n/2` (de exemplu, dacă vectorul are 4 sau 5 elemente sunt necesare 2 interschimbări).

b) Pentru ordonarea elementelor vectorului, este prezentat un algoritm de sortare. **Metoda Bubble Sort** compară fiecare element al vectorului cu cel vecin, iar dacă este cazul, le schimbă între ele.

```

ALGORITHM Bubble_Sort
INCEPUT
gata ← false

```

```

CIT TIMP not gata REPETA
INCEPUT
    gata = true
    PENTRU i=0 LA n-2 REPETA
        INCEPUT
            DACA vect[i] > vect[i+1] ATUNCI
                INCEPUT
                    aux=vect[i]
                    vect[i]=vect[i+1]
                    vect[i+1]=aux
                    gata=fals
                SFARSIT
            SFARSIT
        SFARSIT
    SFARSIT

```

```

// implementarea metodei BubbleSort
int gata =FALSE;int i;
while (!gata){
    gata=TRUE;
    for (i=0; i<=n-2; i++)
        if (vect[i]>vect[i+1]){
            aux=vect[i];
            vect[i]=vect[i+1];
            vect[i+1]=aux;
        }
    gata=FALSE;
}

```

Exemplu: Să se citească elementele matricilor A(MXN), B(NXP) și C(MXN), unde $M \leq 10$, $N \leq 10$ și $P \leq 10$. Să se interschimbe liniile matricii A în modul următor: prima cu ultima, a doua cu penultima, etc. Să se calculeze și să se afișeze matricile: $AT = A^T$, $SUM = A + C$, $PROD = AXB$. Implementarea citirilor și afișărilor se va completa conform exemplului dat în *capitolul 4.2*.

```

#include <iostream.h>
void main()
{
    double a[10][10], b[10][10], c[10][10];
    int m,n,p,j;
    cout<<"m="; cin>>m; cout<<"n="; cin>>n; cout<<"p="; cin>>p;
    // de completat secvența de citire a elementelor matricii a, cu m linii și n coloane
    // de completat secvența de citire a elementelor matricii b, cu n linii și p coloane
    // de completat secvența de afișare a matricii a
    //interschimbarea liniilor matricii A:
    for (i=0; i<m/2; i++)
        for (j=0; j<n; j++){
            double aux=a[i][j];a[i][j]=a[m-1-i][j];a[m-1-i][j]=aux;
        }
    cout<<"Matricea A cu liniile interschimbate:\n";
    // de completat secvența de afișare a matricii a
    // calculul matricii AT = A^T
    double at[10][10]; // at este matricea transpusa

```

```

for (i=0; i<n; i++)
    for (j=0; j<m; j++)
        at[i][j]=a[j][i];
cout<<"A transpus=\n";
// de completat secvența de afișare a matricii at, cu n linii si m coloane
// de completat secvența de citire a elementelor matricii c, cu m linii și n coloane
// calculul matricii SUM=A+C, SUM(MxN):
double sum[10][10]; // sum este matricea suma dintre a si c
for (i=0; i<m; i++)
    for (j=0; j<n; j++)
        sum[i][j]=a[i][j]+ c[i][j];
cout<<"Matricea SUM=A+C este:\n";
// de completat secvența de afișare a matricii sum
double prod[10][10]; // prod este matricea produs dintre a si b
for (i=0; i<m; i++)
    for (j=0; j<p; j++){
        prod[i][j]=0;
        for (k=0; k<n; k++)
            prod[i][j]+=a[i][k]*b[k][j];
    }
cout<<"Matricea produs dintre A si B este:\n";
// de completat secvența de afișare a matricii prod, cu m linii si p coloane
}

```

Se observă că fiecare element din matricea produs $PROD=AXB$ ($A(M \times N)$, $B(N \times P)$),

$PROD(M \times P)$ este de forma: $prod_{i,j} = \sum_{k=0}^{n-1} a_{i,k} * b_{k,j}$, unde $i=\overline{0, m-1}$ și $j=\overline{0, n-1}$.


Întrebări și exerciții


Întrebări teoretice

1. Care este diferența dintre șirurile de caractere și vectorii de caractere?
2. Ce sunt tablourile?
3. De ce tablourile reprezintă date structurate?
4. Prin ce se referă elementele unui tablou?
5. Cine impune tipul unui tablou?

Exerciții aplicative

1. Să se implementeze programele cu exemplele prezentate.
2. Să se scrie programele pentru exercițiile rezolvate care au fost prezentate.
3. Se citesc de la tastatura elementele unei matrici de caractere (nr. linii=nr. coloane), $A(N \times N)$, $N \leq 10$.
 - a) Să se afișeze matricea A;
 - b) Să se formeze și să se afișeze cuvântul format din caracterele pe pe diagonala principală a matricii A;
 - c) Să se calculeze și să se afișeze numărul de litere mari, litere mici și cifre din matrice;
 - d) Să se afișeze cuvântul format din caracterele de pe diagonala secundară;
 - e) Să se afișeze procentul literelor mari, al literelor mici și al cifrelor de pe cele 2 diagonale;
 - f) Să se afișeze caracterele comune aflate pe liniile p și q ($p, q < N$, p și q citite de la tastatură);
 - g) Să se afișeze în ordine alfabetică, crescătoare, literele mari aflate pe coloanele impare.
4. Se citesc de la tastatură elementele unei matrici cu elemente reale, $B(N \times N)$, $N \leq 8$.
 - a) Să se afișeze matricea B;
 - b) Să se calculeze și să se afișeze produsul elementelor de pe coloanele impare;
 - c) Să se calculeze și să se afișeze matricea A, unde: $A = (B + B^T)^2$;
 - d) Să se formeze și să se afișeze vectorul V, ale cărui elemente sunt elementele pozitive din matricea A;
 - e) Să se calculeze și să se afișeze sumele și produsele elementelor matricii A, aflate în triunghiurile hașurate:


 - f) Să se calculeze procentul elementelor pozitive aflate pe diagonala secundară;
 - g) Să se calculeze și să se afișeze matricea C, unde: $C = 3 * B^T + B^2$;
 - h) Să se calculeze și să se afișeze matricea D, unde: $D = B + B^2 + B^3 + B^4$;
 - i) Să se interschimbe coloanele matricii A astfel: prima cu ultima, a doua cu antipenultima, etc.
5. Se citesc de la tastatură elementele unei matrici de numere întregi $C(N \times N)$, $N \leq 10$.
 - a) Să se afișeze matricea C;
 - b) Să se calculeze și să se afișeze procentul elementelor impare de pe liniile pare;
 - c) Să se calculeze și să se afișeze matricea B, unde: $B = C^2$;
 - d) Să se calculeze și să se afișeze matricea E, unde: $E = (C + C^T)^2 + I$, unde I este matricea unitate;
 - e) Să se afle elementul minim din matricea C;
 - f) Să se înlocuiască elementul maxim din matricea C cu valoarea val, introdusă de la tastatură;
 - g) Să se afișeze elementele matricii C care sunt numere prime;
 - h) Să se calculeze și să se afișeze sumele și produsele elementelor matricii A, aflate în triunghiurile hașurate:



1. Variabile pointer	3.1. Pointeri și șiruri de caractere
1.1. Declararea variabilelor pointer	3.2. Pointeri și tablouri bidimensionale
1.2. Inițializarea variabilelor pointer	4. Tablouri de pointeri
1.3. Pointeri generici	5. Pointeri la pointeri
2. Operații cu pointeri	6. Modificatorul const în declararea pointerilor
3. Pointeri și tablouri	

1. Variabile pointer

Pointerii sunt variabile care au ca valori sunt adresele altor variabile (obiecte). Variabila este un nume simbolic utilizat pentru un grup de locații de memorie. Valoarea memorată într-o variabilă pointer este o *adresă*.

Din punctul de vedere al conținutului zonei de memorie adresate, se disting următoarele categorii de pointeri:

- ❑ pointeri *de date (obiecte)* - conțin adresa unei variabile din memorie;
- ❑ pointeri *generici (numiți și pointeri void)* - conțin adresa unui obiect oarecare, de tip neprecizat;
- ❑ pointeri *de funcții* - conțin adresa codului executabil al unei funcții.



Figura 1. Variabile pointer.

În figura 1, variabila `x` este memorată la adresa 1024 și are valoarea 5. Variabila `ptrx` este memorată la adresa de memorie 1028 și are valoarea 1024 (adresa variabilei `x`). Vom spune că `ptrx` pointează către `x`, deoarece valoarea variabilei `ptrx` este chiar adresa de memorie a variabilei `x`.

1.1. Declararea variabilelor pointer. Sintaxa *declarației unui pointer de date* este:

```
tip *identificator_pointer;
```

Simbolul `*` precizează că `identificator_pointer` este numele unei variabile pointer de date, iar `tip` este tipul obiectelor a căror adresă o va conține.

Exemplu:

```
int u, v, *p, *q;           // *p, *q sunt pointeri de date (către int)
double a, b, *p1, *q1;      // *p1, *q1 sunt pointeri către date de tip double
```

Pentru **pointerii generici**, se folosește declarația:

```
void *identificator_pointer;
```

Exemplu:

```
void *m;
```

Aceasta permite declararea unui pointer generic, care nu are asociat un tip de date precis. Din acest motiv, în cazul unui pointer vid, dimensiunea zonei de memorie adresate și interpretarea informației nu sunt definite, iar proprietățile diferă de ale pointerilor de date.

1.2. Inițializarea variabilelor pointer. Există doi **operatori unari** care permit utilizarea variabilelor pointer:

- **&** - *operatorul adresă (de referențiere)* - pentru aflarea adresei din memorie a unei variabile;
- ***** - *operatorul de indirectare (de deferențiere)* - care furnizează valoarea din zona de memorie spre care pointează pointerul operand.

În exemplul prezentat în figura 1, pentru variabila întreagă `x`, expresia **&x** furnizează *adresa variabilei x*. Pentru variabila pointer de obiecte `int`, numită `ptr`, expresia ***ptr** înseamnă *conținutul locației de memorie a cărei adresă este memorată în variabila ptr*. Expresia `*ptr` poate fi folosită atât pentru aflarea valorii obiectului spre care pointează `ptr`, cât și pentru modificarea acesteia (printr-o operație de atribuire).

Exemplu:

```
int x, y, *ptr;
// ptr- variabilă pointer către un int; x,y-variabile predefinite, simple, de tip int
x=5; cout<<"Adresa variabilei x este:"<<&x<<'\\n';
cout<<"Valoarea lui x:"<<x<<'\\n';
ptr=&x;           // atribuire: variabila ptr conține adresa variabilei x
cout<<"Variabila pointer ptr are valoarea:"<<ptr;
cout<<" si adreseaza obiectul:"<< *ptr<<'\\n';
y=*ptr; cout<<"y="<<y<<'\\n';           // y=5
x=4; cout<<"x="<<x<<'\\n'; cout<<" *ptr="<<*ptr<<'\\n';
// x si *ptr reprezinta acelasi obiect, un intreg cu valoarea 4
x=70;           // echivalenta cu *ptr=70;
y=x+10;         // echivalenta cu y=*ptr+10
```

În exemplul anterior, atribuirea **ptr=&x** se execută astfel: operatorul **&** furnizează adresa lui `x`; operatorul **=** atribuie valoarea (care este o adresă) variabilei pointer `ptr`.

Atribuirea **y=*ptr** se realizează astfel: operatorul ***** accesează conținutul locației a cărei adresă este conținută în variabila **ptr**; operatorul **=** atribuie valoarea variabilei **y**.

Declarația **int *ptr;** poate fi, deci, interpretată în două moduri, ambele corecte:

- **ptr** este de tipul **int** (* ptr este de tip pointer spre int);
- ***ptr** este de tipul **int** (conținutul locației spre care pointează variabila **ptr** este de tipul int).

Construcția **tip *** este de tipul pointer către int.

Atribuirea **x=8;** este echivalentă cu **ptr=&x; *p=x;**

Variabilele pointer, alături de operatorii de referențiere și de deferențiere, pot apare în expresii.

Exemple:

```
int x, y, *q; q=&x;
*q=8;          // echivalentă cu x=8;
q=&5;           // invalidă - constantele nu au adresă
*x=9;          // invalidă - x nu este variabilă pointer
x=&y; //invalidă: x nu este variabilă pointer, deci nu poate fi folosită cu operatorul de indirectare
y=*q + 3;      // echivalentă cu y=x+3;
*q = 0;        // setează x pe 0
*q += 1;       // echivalentă cu (*q)++ sau cu x++
int *r; r = q;
/* copiază conținutul lui q (adresa lui x) în r, deci r va pointa tot către x (va conține tot adresa lui x)*/
double w, *r = &w, *r1, *r2; r1= &w; r2=r1;
cout<<"r1="<<r1<<'\\n';          //afișează valoarea pointerului r1 (adresa lui w)
cout<<"&r1="<<&r1<<'\\n';        // afișează adresa variabilei r1
cout<<"*r1= " <<*r1<<'\\n';
double z=*r1;          // echivalentă cu z=w
cout<<"z="<<z<<'\\n';
```

1.3. Pointeri generici. La declararea pointerilor generici (**void *nume;**) nu se specifică un tip, deci unui pointer void i se pot atribui adrese de memorie care pot conține date de diferite tipuri: int, float, char, etc. Acești pointeri pot fi folosiți cu mai multe tipuri de date, de aceea este necesară folosirea **conversiilor explicite** prin expresii de tip cast, pentru a preciza tipul datei spre care pointează la un moment dat pointerul generic.

Exemplu:

```
void *v1, *v2; int a, b, *q1, *q2;
q1 = &a; q2 = q1; v1 = q1;
q2 = v1; // eroare: unui pointer cu tip nu i se poate atribui un pointer generic
q2 = (int *) v1; double s, *ps = &s;
int c, *l; void *sv;
l = (int *) sv; ps = (double *) sv;
*(char *) sv = 'a'; /*Interpretare: adresa la care se găsește valoarea lui sv este interpretată ca fiind adresa zonei de memorie care conține o data de tip char.*/
```

Pe baza exemplului anterior, se pot face observațiile:

1. Conversia tipului pointer generic spre un tip concret înseamnă, de fapt, precizarea tipului de pointer pe care îl are valoarea pointerului la care se aplică conversia respectivă.
2. Conversia tipului pointer generic asigură o flexibilitate mai mare în utilizarea pointerilor.
3. Utilizarea în mod abuziv a pointerilor generici poate constitui o sursă de erori.

2. Operații cu pointeri

În afara operației de *atribuire* (prezentată anterior), asupra variabilelor pointer se pot realiza operații de *comparare*, *adunare* și *scădere* (inclusiv *incrementare* și *decrementare*).

- *Compararea valorilor variabilelor pointer.* Valorile a doi pointeri pot fi *comparate*, folosind *operatorii relaționali*, ca în exemplul următor:

```
int *p1, *p2;
if (p1<p2)
    cout<<"p1="<<p1<<"<"<<"p2="<<p2<<"\n";
else cout<<"p1="<<p1<<">="<<"p2="<<p2<<"\n";
```

O operație uzuală este *compararea unui pointer cu valoarea nulă*, pentru a verifica *dacă acesta adresează un obiect*. Compararea se face cu constanta simbolică NULL (definită în header-ul `stdio.h`) sau cu valoarea 0.

Exemplu:

```
if (!p1)                // sau if (p1 != NULL)
    . . . . .;         // pointer nul
else . . . . .;         // pointer nenul
```

- *Adunarea sau scăderea.* Sunt permise operații de *adunare* sau *scădere între un pointer de obiecte și un întreg*. Astfel, dacă *ptr* este un pointer către tipul *tip* (`tip *ptr`), iar *n* este un întreg, expresiile

$ptr + n$ și $ptr - n$

au ca valoare, valoarea lui *ptr* la care se adaugă, respectiv, se scade *n * sizeof(tip)*.

Un caz particular al adunării sau scăderii dintre un pointer de date și un întreg (*n=1*) îl reprezintă *incrementarea și decrementarea unui pointer de date*. În expresiile *ptr++*, respectiv *ptr--*, valoarea variabilei *ptr* devine *ptr+sizeof(tip)*, respectiv, *ptr-sizeof(tip)*.

Este permisă *scăderea a doi pointeri de obiecte de același tip*, rezultatul fiind o valoare întreagă care reprezintă diferența de adrese divizată prin dimensiunea tipului de bază.

Exemplu:

```
int a, *pa, *pb;
cout<<"&a="<<&a<<"\n"; pa=&a; cout<<"pa="<<pa<<"\n";
cout<<"pa+2"<<pa+2<<"\n"; pb=pa++; cout<<"pb="<<pb<<"\n";
int i=pa-pb; cout<<"i="<<i<<"\n";
```

3. Pointeri și tablouri

În limbajele C/C++ există o strânsă legătură între pointeri și tablouri, deoarece *numele unui tablou* este un *pointer* (constant!) care are ca valoare adresa primului element din tablou. Diferența dintre numele unui tablou și o variabilă pointer este aceea că unei variabile de tip pointer i se pot atribui valori la execuție, lucru imposibil pentru numele unui tablou. Acesta are tot timpul, ca valoare, adresa primului său element. De aceea, se spune că *numele unui tablou* este un *pointer constant* (valoarea lui nu poate fi schimbată). *Numele unui tablou* este considerat ca fiind un *rvalue* (right value-valoare dreapta), deci nu poate apare decât în partea dreaptă a unei expresii de atribuire. *Numele unui pointer* (în exemplul următor, `*ptr`) este considerat ca fiind un *lvalue* (left value-valoare stânga), deci poate fi folosit atât pentru a obține valoarea obiectului, cât și pentru a o modifica printr-o operație de atribuire.

Exemplu:

```
int a[10], *ptr; // a este definit ca &a[0]; a este pointer constant
a = a + 1;      // ilegal
ptr = a;        // legal: ptr are aceeași valoare ca și a, respectiv adresa elementului a[0]
                // ptr este variabilă pointer, a este constantă pointer.
int x = a[0];   // echivalent cu x = *ptr; se atribuie lui x valoarea lui a[0]
```

Deoarece numele tabloului `a` este sinonim pentru adresa elementului de indice zero din tablou, asignarea `ptr=&a[0]` poate fi înlocuită, ca în exemplul anterior, cu `ptr=a`.

3.1. Pointeri și șiruri de caractere. Așa cum s-a arătat în **capitolul 4**, un șir de caractere poate fi memorat într-un vector de caractere. Spre deosebire de celelalte constante, constantele șir de caractere nu au o lungime fixă, deci numărul de octeți alocați la compilare pentru memorarea șirului, variază. Deoarece valoarea variabilelor pointer poate fi schimbată în orice moment, cu multă ușurință, este preferabilă utilizarea acestora, în locul tablourilor de caractere (vezi exemplul următor).

Exemplu:

```
char sir[10];      char *psir;
sir = "hello";    // ilegal
psir = "hello";   // legal
```

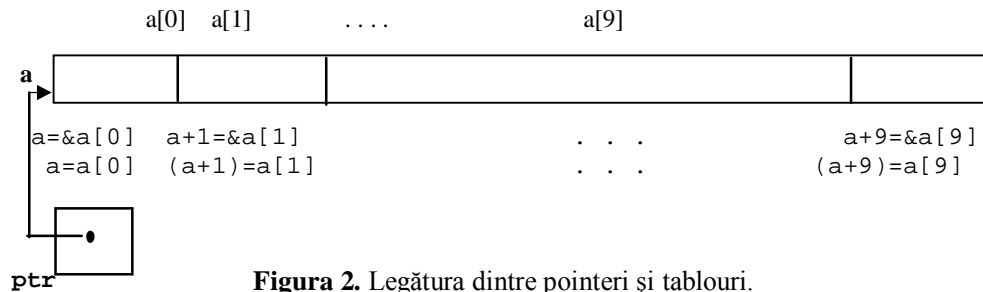
Operația de indexare a elementelor unui tablou poate fi realizată cu ajutorul variabilelor pointer.

Exemplu:

```
int a[10], *ptr; // a este pointer constant; ptr este variabilă pointer
ptr = a;        // ptr este adresa lui a[0]
ptr+i           înseamnă ptr+(i*sizeof(int)), deci: ptr + i ⇔ &a[i]
```

Deoarece numele unui tablou este un pointer (constant), putem concluziona (figura 2):

$$\begin{aligned} \mathbf{a+i} &\Leftrightarrow \mathbf{\& a[i]} \\ \mathbf{a[i]} &\Leftrightarrow \mathbf{* (a+i)} \end{aligned}$$



Exercițiu: Să se scrie următorul program (care ilustrează legătura dintre pointeri și vectori) și să se urmărească rezultatele execuției acestuia.

```
#include <iostream.h>
void main(void)
{int a[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }; int *pi1 = a;
int *pi2 = &a[0]; int *pi3;
cout<<"a="<<a<<"&a="<<&a<<"*a="<<*a<<"\n";
cout<<"a+1="<<(a+1)<<" &a[1]="<< &a[1]<<"\n";
cout<<"a[1]="<<a[1]<<" * (a+1)="<< * (a+1)<<"\n";
cout<<"pi1="<<pi1<<"pi2="<<pi2<<"\n"; int x=*pi1;
/* x primește valoarea locației a carei adresă se află în variabila pointer pi1, deci valoarea lui a[0] */
cout<<"x="<<x<<"\n"; x=*pi1++; // echivalent cu *(pi1++) x=1
cout<<"x="<<x<<"\n"; x=( *pi1)++;
/* x=0: întâi atribuirea, apoi incrementarea valorii spre care pointează pi1. În urma incrementării,
valoarea lui a[0] devine 1 */
cout<<"x="<<x<<"\n"; cout<<*pi1<<"\n";x=*++pi1; //echivalent cu
*(++pi1)
cout<<"x="<<x<<"\n"; x=++( *pi1); cout<<"x="<<x<<"\n"; pi1=a;
pi3=pi1+3;
cout<<"pi1="<<pi1<<"*pi1="<<*pi1<<"&pi1="<<&pi1<<"\n";
cout<<"pi3="<<pi3<<"*pi3="<<*pi3<<"&pi3="<<&pi3<<"\n";
cout<<"pi3-pi1="<<(pi3-pi1)<<"\n"; //pi3-pi1=3
}
```

Exercițiu: Să se scrie următorul program (legătura pointeri-șiruri de caractere) și să se urmărească rezultatele execuției acestuia.

```
#include <iostream.h>
void main(void)
{int a=-5, b=12, *pi=&a; double u=7.13, v=-2.24, *pd=&v;
char sir1[]="sirul 1", sir2[]="sirul 2", *psir=sir1;
cout<<"a="<<a<<" &a="<<&a<<" b="<<b<<" &b="<<&b<<"\n";
cout<<"*pi="<<*pi<<"pi="<<pi<<" &pi="<<&pi<<"\n";
cout<<"*pd="<<*pd<<"pd="<<pd<<" &pd="<<&pd<<"\n";
cout<<"*sir1="<<*sir1<<" sir1="<<sir1<<" &sir1="<<&sir1<<"\n";
// *sir1=s sir1=sirul 1 &sir1=0xffd6
cout<<"*sir2="<<*sir2<<" sir2="<<sir2<<" &sir2="<<&sir2<<"\n";
// *sir2=s sir2=sirul 2 &sir2=0xffce
cout<<"*psir="<<*psir<<" psir="<<psir<<" &psir="<<&psir<<"\n";
// *psir=s psir=sirul 1 &psir=0xffcc
cout<<"sir1+2="<<(sir1+2)<<" psir+2="<<(psir+2)<<"\n";
}
```

```

// sir1+2=rul 1 psir+2=rul 1
cout<<" * (sir1+2)="<< * (sir1+2)<<'\\n' ;
// *(sir1+2)=r valoarea elementului de indice 2
void *pv1, *pv2;
pv1=psir; pv2=sir1;
cout<<"pv1="<<pv1<<"&pv1="<<&pv1<<'\\n' ;
cout<<"pv2="<<pv2<<"&pv2="<<&pv2<<'\\n' ;
pi=&b; pd=&v; psir=sir2;
cout<<" * pi="<<*pi<<"pi="<<pi<<" &pi="<<&pi<<'\\n' ;
cout<<" * pd="<<*pd<<"pd="<<pd<<" &pd="<<&pd<<'\\n' ;
cout<<" * psir="<<*psir<<"psir="<<psir<<" &psir="<<&psir<<'\\n' ;
}

```

Exercițiu: Să se scrie un program care citește elementele unui vector de întregi, cu maxim 20 elemente și înlocuiește elementul maxim din vector cu o valoare introdusă de la tastatură. Se va folosi aritmetica pointerilor.

```

#include <iostream.h>
void main()
{ int a[20];
int n, max, indice; cout<<"Nr. elemente:"; cin>>n;
for (i=0; i<n; i++)
    { cout<<"a["<<i<<"]="; cin>>*(a+i);}
// citirea elementelor vectorului
max=*a; indice=0;
for (i=0; i<n; i++)
    if (max<=*(a+i))
        { max=*(a+i); indice=i;}
// aflarea valorii elementului maxim din vector și a poziției acestuia
int val;
cout<<"Valoare de înlocuire:"; cin >> val;
*(a+indice)=val;
// citirea valorii cu care se va înlocui elementul maxim
for (i=0; i<n; i++)
    cout<<*(a+i)<<'\\t' ;
cout<<'\\n' ;
// afișarea noului vector
/* în acest mod de implementare, în situația în care în vector există mai multe elemente a căror valoare
este egală cu valoarea elementului maxim, va fi înlocuit doar ultimul dintre acestea (cel de indice
maxim).*/
}

```

3.2. Pointeri și tablouri multidimensionale. Elementele unui tablou bidimensional sunt păstrate tot într-o zonă continuă de memorie, dar inconvenientul constă în faptul că ne gândim la aceste elemente în termeni de rânduri (linii) și coloane (figura 3). Un tablou bidimensional este tratat ca un tablou unidimensional ale cărui elemente sunt tablouri unidimensionale.

```
int M[4][3]={ {10, 5, -3}, {9, 18, 0}, {32, 20, 1}, {-1, 0, 8}
};
```

Compilerul tratează atât M, cât și M[0], ca *tablouri* de mărimi diferite. Astfel:

```

cout<<"Marime M:"<<sizeof(M)<<'\\n' ;           // 24 = 2octeți * 12elemente
cout<<"Marime M[0]"<<sizeof(M[0])<<'\\n' ;       // 6 = 2octeți * 3elemente
cout<<"Marime M[0][0]"<<sizeof(M[0][0])<<'\\n' ; // 4 octeți (sizeof(int))

```

Matricea M

Pagina	M[0]	10	5	-3
	M[1]	9	18	0
	M[2]	32	20	1
	M[3]	-1	0	8

Matricea M are 4 linii și 3 coloane.

Numele tabloului bidimensional, M, referă întregul tablou;

M[0] referă prima linie din tablou;

M[0][0] referă primul element al tabloului.

Așa cum compilatorul evaluează referința către un tablou unidimensional ca un pointer, un tablou bidimensional este referit într-o manieră similară. Numele tabloului bidimensional, *M*, reprezintă adresa (pointer) către primul element din tabloul bidimensional, acesta fiind prima linie, *M*[0] (tablou unidimensional). *M*[0] este adresa primului element (*M*[0][0]) din linie (tablou unidimensional), deci *M*[0] este un pointer către int: *M* = *M*[0] = &*M*[0][0]. Astfel, *M* și *M*[linie] sunt pointeri constanți.

Putem concluziona:

- ***M*** este un pointer către un tablou unidimensional (de întregi, în exemplul anterior).
- ****M*** este pointer către int (pentru că *M*[0] este pointer către int), și

$$*M = *(M + 0) \Leftrightarrow M[0].$$
- *****M*** este întreg; deoarece ***M*[0][0]** este

$$\text{int, } **M = * (*M) \Leftrightarrow * (M[0]) = * (M[0]+0) \Leftrightarrow M[0][0].$$

Exercițiu: Să se testeze programul următor, urmărind cu atenție rezultatele obținute.

```
#include <iostream.h>
#include <conio.h>
void main()
{int a[3][3]={ {5,6,7}, {55,66,77}, {555,666,777}};
clrscr();
cout<<"a="<<a<<" &a="<<&a<<" &a[0]="<<&a[0]<<"\n";
cout<<"Pointeri catre vectorii liniei\n";
for (int i=0; i<3; i++){
    cout<<" *(a+"<<i<<" )="<<*(a+i);
    cout<<" a["<<i<<" ]="<<a[i]<<"\n";
}
// afișarea matricii
for (i=0; i<3; i++){
    for (int j=0; j<3; j++)
        cout<<*(*(a+i)+j)<<"\t"; //sau:
        cout<<*(a[i]+j)<<"\t";
    cout<<"\n";
}
}
```

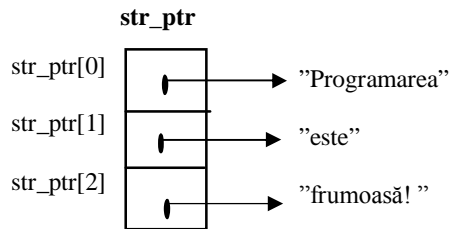
4. Tablouri de pointeri

Un tablou de pointeri este un tablou ale cărui elemente sunt pointeri. Modul general de declarare a unui tablou de pointeri:

```
tip *nume_tablou[dim];
```

Să considerăm exemplul în care se declară și se inițializează tabloul de pointeri **str_ptr** (figura 4):

```
char * str_ptr[3] = { "Programarea", "este", "frumoasă!" };
```



Deoarece operatorul de indexare [] are prioritate mai mare decât operatorul de diferențiere , declarația `char str_ptr[3]` este echivalentă cu `char (str_ptr[3])`, care precizează că `str_ptr` este un vector de trei elemente, fiecare element este pointer către caracter.

Figura 4. Tabloul de pointeri **str_ptr**.

În ceea ce privește declarația: `char* (str_ptr[3])`, se poate observa:

1. **str_ptr[3]** este de tipul **char *** (fiecare dintre cele trei elemente ale vectorului `str_ptr[3]` este de tipul pointer către `char`);
2. ***(str_ptr[3])** este de tip **char** (conținutul locației adresate de un element din `str_ptr[3]` este de tip `char`).

Fiecare element (pointer) din `str_ptr` este inițializat să poarte către un șir de caractere constant. Fiecare dintre aceste șiruri de caractere se găsesc în memorie la adresele memorate în elementele vectorului `str_ptr`: `str_ptr[0]`, `str_ptr[1]`, etc.

Să ne amintim de la pointeri către șiruri de caractere:

```
char *p="heLLo";
*( p+1) = 'e'      ⇔      p[1] = 'e';
```

În mod analog:

```
str_ptr[1] = "este";
*( str_ptr[1] + 1) = 's';      ⇔      str_ptr[1][1]='s';
```

Putem conculziona:

- **str_ptr** este un *pointer către un pointer de caractere*.
- ***str_ptr** este *pointer către char*. Este evident, deoarece `str_ptr[0]` este *pointer către char*, iar
`*str_ptr = *(str_ptr [0] + 0) ⇔ str_ptr[0]`.
- ****str_ptr** este un *pointer de tip char*. Este evident, deoarece `str_ptr[0][0]` este de tip `char`, iar
`**str_ptr=*(*str_ptr) ⇔ *(str_ptr[0])=*(str_ptr[0]+0) ⇔ str_ptr[0][0]`.

5. Pointeri la pointeri

Să revedem exemplul cu tabloul de pointeri **str_ptr**. Șirurile spre care pointează elementele tabloului pot fi accesate prin `str_ptr[index]`, însă deoarece `str_ptr` este un *pointer constant*, acestuia nu i se pot aplica operatorii de incrementare și decrementare. Este ilegală :

```
for (i=0;i<3;i++)
    cout<<str_ptr++;
```

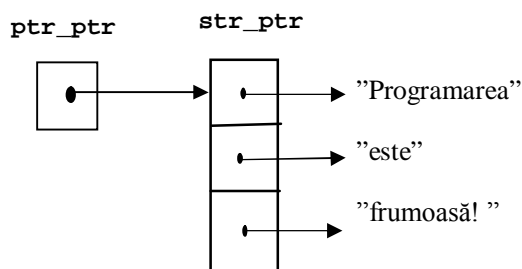
De aceea, putem declara o variabilă pointer **ptr_ptr**, care să poarte către primul element din `str_ptr`. Variabila **ptr_ptr** este *pointer către pointer* și se declară astfel:

```
char **ptr_ptr;
```

În exemplul următor este prezentat modul de utilizare a pointerului la pointer `ptr_ptr` (figura 5).

Exemplu:

```
char **ptr_ptr;
char * str_ptr[3] = { "Programarea", "este", "frumoasă!" };
char **ptr_ptr;
ptr_ptr = str_ptr;
```



După atribuire, și `str_ptr` și `ptr_ptr` pointează către aceeași locație de memorie (primul element al tabloului `str_ptr`). În timp ce fiecare element al lui `str_ptr` este un pointer, `ptr_ptr` este un *pointer către pointer*. Deoarece `ptr_ptr` este un pointer variabil, valoarea lui poate fi schimbată:

```
for (i=0;i<3;i++)
    cout<<ptr_ptr++;
```

Figura 5. Pointerul la pointer **ptr_ptr**.

Referitor la declarația `char **ptr_ptr`, putem concluziona:

- **ptr_ptr** este de tipul `char**` (*ptr_ptr este pointer la pointer către char*);
- ***ptr_ptr** este de tipul `char*` (*conținutul locației ptr_ptr este de tipul pointer către char*);
- ****ptr_ptr** este de tipul `char` (***ptr_ptr ⇔ *(*ptr_ptr); conținutul locației *ptr_ptr este de tipul char*).

6. Modificatorul *const* în declararea pointerilor

Modificatorul *const* se utilizează frecvent în declararea pointerilor, având următoarele roluri:

- Declararea unui *pointer* spre o *dată constantă*

```
const *tip nume_pointer=dată_constantă;
```

Exemplu:

```
const char *sirul="azi";
```

//variabila sirul este pointer spre un şir constant de caractere

Atribuirile de forma:

```
*sirul="coco";  
*(sirul+2)='A';
```

nu sunt acceptate, deoarece pointerul *sirul* pointează către o dată constantă (şir constant).

- Declararea unui *pointer constant* către o *dată care nu este constantă*

```
tip * const nume_pointer=dată_neconst;
```

Exemplu:

```
char * const psir="abcd"; const char *sir="un text";
```

```
sir="alt sir";
```

```
//incorect, sir pointează către dată constantă
```

```
psir=sir;
```

```
//incorect, deoarece psir este pointer constant
```

- Declararea unui *pointer constant* către o *dată constantă*

```
const tip * const nume_pointer=dată_constantă;
```

Exemplu:

```
const char * const psir1="mnP";
```

```
*(psir1+2)='Z'; // incorect, data spre care pointează psir1 este constantă
```

```
psir1++; // incorect, psir1 este pointer constant
```

Întrebări și exerciții

Întrebări teoretice

1. În ce constă operația de incrementare a pointerilor?
2. Tablouri de pointeri.
3. Ce sunt pointerii generici?
4. Ce operații se pot realiza asupra variabilelor pointer?
5. De ce numele unui pointer este lvalue?
6. Ce fel de variabile pot constitui operandul operatorului de deferențiere?
7. Operatorul de referențiere.
8. Unui pointer generic i se poate atribui valoarea unui pointer cu tip?
9. **Care este legătura între tablouri și pointeri?**
10. De ce numele unui tablou este rvalue?

Exerciții practice

1. Să se implementeze programele cu exemplele prezentate.
2. Să se scrie programele pentru exercițiile rezolvate care au fost prezentate.
3. Analizați următoarele secvențe de instrucțiuni. Identificați secvențele incorecte (acolo unde este cazul) și sursele erorilor:

```
❑ int a,b,*c; a=7; b=90; c=a;
❑ double y, z, *x=&z; z=&y;
❑ char x, **p, *q; x = 'A'; q = &x; p = &q; cout<<"x="<<x<<'\\n';
cout<<"**p="<<**p<<'\\n'; cout<<"*q="<<*q<<'\\n';
cout<<"p="<<p<<" q="<<q<<"*p="<<*p<<'\\n';
❑ char *p, x[3] = {'a', 'b', 'c'}; int i, *q, y[3] = {10, 20, 30};
p = &x[0];
for (i = 0; i < 3; i++)
{
cout<<"*p="<<*p<<" p="<<p<<'\\n';
p++;
}
q = &y[0];
for (i = 0; i < 3; i++)
{
cout<<"*q="<<*q<<"q="<<q<<'\\n';
q++;
}

const char *sirul="să programăm"; *(sirul)++;
double a, *s; s=&(a+89); cout<<"s="<<s<<'\\n';
double a1, *a2, *a3; a2=&a1; a2+=7.8; a3=a2; a3++;
int m[10], *p;p=m;
for (int i=0; i<10; i++)
cout<<*m++;
void *p1; int *p2; int x; p2=&x; p2=p1;
char c='A'; char *cc=&c; cout<<(*cc)+<<'\\n';
```

4. Rescrieți programele pentru problemele din **capitolul 4**, utilizând aritmetica pointerilor.

- | | |
|-------------------------------------------------|-------------------------------------------|
| 1. Structura unei funcții | 4. Tablouri ca parametri |
| 2. Apelul și prototipul unei funcții | 5. Funcții cu parametri implicați |
| 3. Transferul parametrilor unei funcții | 6. Funcții cu număr variabil de parametri |
| 3.1. Transferul parametrilor prin valoare | 7. Funcții predefinite |
| 3.2. Transferul prin pointeri | 8. Clase de memorare |
| 3.3. Transferul prin referință | 9. Moduri de alocare a memoriei |
| 3.4. Transferul parametrilor către funcția main | 10. Funcții recursive |
| | 11. Pointeri către funcții |

1. Structura unei funcții

Un program scris în limbajul C/C++ este *un ansamblu de funcții*, fiecare dintre acestea efectuând o activitate bine definită. Din punct de vedere conceptual, **funcția** reprezintă o aplicație definită pe o mulțime D (D =mulțimea, domeniul de definiție), cu valori în mulțimea C (C =mulțimea de valori, codomeniul), care îndeplinește condiția că oricărui element din D îi corespunde un unic element din C .

Funcțiile *comunică prin argumente*: ele primesc ca parametri (argumente) datele de intrare, efectuează prelucrările descrise în corpul funcției asupra acestora și pot returna o valoare (rezultatul, datele de ieșire). Execuția programului începe cu funcția principală, numită **main**. Funcțiile pot fi descrise în cadrul aceluiași fișier, sau în fișiere diferite, care sunt testate și compilate separat, asamblarea lor realizându-se cu ajutorul linkerului de legături. O funcție este formată din antet și corp:

```
antet_funcție
{ corpul_funcției }
```

Sau:

```
tip_val_return nume_func (lista_declaratiilor_param_formali)
{
    declarații_variabale_locale
    instrucțiuni
    return valoare
}
```

Prima linie reprezintă **antetul** funcției, în care se indică: tipul funcției, numele acesteia și lista declarațiilor parametrilor formali. La fel ca un operand sau o expresie, o funcție are un *tip*, care este dat de tipul valorii returnate de funcție în funcția apelantă. Dacă funcția nu întoarce nici o valoare, în locul *tip_vali_return* se specifică **void**. Dacă *tip_val_return* lipsește, se consideră, implicit, că acesta este **int**. *Nume_funcție* este un identificator.

Lista_declaratiilor_param_formali (încadrată între paranteze rotunde) constă într-o listă (enumerare) care conține tipul și identificatorul fiecărui parametru de intrare, despărțite prin virgulă. Tipul unui parametru poate fi oricare, chiar și tipul pointer. Dacă lista parametrilor formali este vidă, în antet, după numele funcției, apar doar parantezele (), sau (**void**).

Corpul funcției este un bloc, care implementează algoritmul de calcul folosit de către funcție. În corpul funcției apar (în orice ordine) declarații pentru variabilele locale și instrucțiuni. Dacă funcția întoarce o valoare, se folosește instrucțiunea **return valoare**. La execuție, la întâlnirea acestei instrucțiuni, se revine în funcția apelantă. În limbajul C/C++ se utilizează **declarații și definiții** de funcții.

Declarația conține antetul funcției și informează compilatorul asupra tipului, numelui funcției și a listei parametrilor formali (în care se poate indica doar tipul parametrilor formali, nu și numele acestora). Declarațiile de funcții se numesc **prototipuri**, și sunt constituite din antetul funcției, din care pot lipsi numele parametrilor formali.

Definiția conține antetul funcției și corpul acesteia. Nu este admisă definirea unei funcții în corpul altei funcții.

O formă învechită a antetului unei funcții este aceea de a specifica în lista parametrilor formali doar numele acestora, nu și tipul. Această libertate în omiterea tipului parametrilor constituie o sursă de erori.

```
tipul_valorii_returnate nume_funcție (lista_parametrilor_ formali)
declararea_parametrilor_formali
{
    declarații_variabale_locale
    instrucțiuni
    return valoare
}
```

2. Apelul și prototipul funcțiilor

O funcție poate fi **apelată** printr-o construcție urmată de punct și virgulă, numită **instrucțiune de apel**, de forma:

```
nume_funcție (lista_parametrilor_efectivi) ;
```

Parametri efectivi trebuie să corespundă cu cei formali ca ordine și tip. La apel, se atribuie parametrilor formali valorile parametrilor efectivi, după care se execută instrucțiunile din corpul funcției. La revenirea din funcție, controlul este redat funcției apelante, și execuția continuă cu instrucțiunea următoare instrucțiunii de apel, din funcția apelantă. O altă posibilitate de a apela o funcție este aceea în care apelul funcției constituie operandul unei expresii. Acest lucru este posibil doar în cazul în care funcția returnează o valoare, folosită în calculul expresiei.

Parametri declarați în antetul unei funcții sunt numiți **formali**, pentru a sublinia faptul că ei nu reprezintă valori concrete, ci numai țin locul acestora pentru a putea exprima procesul de calcul realizat prin funcție. Ei se concretizează la execuție prin apelurile funcției.

Parametri folosiți la apelul unei funcții sunt **parametri reali, efectivi, concreți**, iar valorile lor vor fi atribuite parametrilor formali, la execuție. Utilizarea parametrilor formali la implementarea funcțiilor și atribuirea de valori concrete pentru ei, la execuție, reprezintă un prim nivel de abstractizare în programare. Acest mod de programare se numește **programare procedurală** și realizează un proces de **abstractizare prin parametri**.

Variabilele declarate în interiorul unei funcții, cât și parametri formali ai acesteia nu pot fi accesați decât în interiorul acesteia. Aceste variabile sunt numite **variabile locale** și nu pot fi accesate din alte funcții. Domeniul de vizibilitate a unei variabile este

porțiunea de cod la a cărei execuție variabila respectivă este accesibilă. Deci, domeniul de vizibilitate a unei variabile locale este funcția în care ea a fost definită.

Exemplu:

```
int f1(void)
{ double a,b; int c;
  . . .
  return c; // a, b, c - variabile locale, vizibile doar în corpul funcției
}
void main()
{ . . . . . // variabile a și b nu sunt accesibile în main()
}
```

Dacă în interiorul unei funcții există instrucțiuni compuse (blocuri) care conțin declarații de variabile, aceste variabile nu sunt vizibile în afara blocului.

Exemplu:

```
void main()
{ int a=1, b=2;
  cout << "a="<<a<<" b="<<b<<" c="<<c'\n'; // a=1 b=2, c nedeclarat
  . . . . .
  { int a=5; b=6; int c=9;
    cout << "a="<<a<<" b="<<b<<' \n'; // a=5 b=6 c=9
    . . . . .
  }
  cout << "a="<<a<<" b="<<b<<" c="<<c'\n'; // a=1 b=6, c nedeclarat
  . . . . .
}
```

Exercițiu: Să se scrie următorul program (pentru înțelegerea modului de apel al unei funcții) și să se urmărească rezultatele execuției acestuia.

```
#include <iostream.h>
void f_afis(void)
{
  cout<<"Se execută instrucțiunile din corpul funcției\n";
  double a=3, b=9.4; cout<<a<<"*"<<b<<"="<<a*b<<' \n';
  cout<<"Ieșire din funcție!\n"; }

void main()
{
  cout<<"Intrare în funcția principală\n";
  f_afis( ); //apelul funcției f_afis, printr-o instrucțiune de apel
  cout<<"Terminat MAIN!\n"; }
```

Exercițiu: Să se scrie un program care citește două numere și afișează cele mai mare divizor comun al acestora, folosind o funcție care îl calculează.

```
#include <iostream.h>
int cmmdc(int x, int y)
{
  if (x==0 || y==1 || x==1 || y==0) return 1;
  if (x<0) x=-x;
  if (y<0) y=-y;
  while (x != 0){
    if ( y > x )
      {int z=x; x=y; y=z; }
    x-=y; // sau: x%=y;
  }
  return y;}
```



```

void main()
{
    int n1,n2; cout<<"n1=";cin>>n1; cout<<"n2=";cin>>n2;
    int diviz=cmmdc(n1,n2);
    cout<<"Cel mai mare divizor comun al nr-lor:"<<n1<<" și ";
    cout<<n2<<" este:"<<diviz<<'\\n';
    /* sau:
    cout<<"Cel mai mare divizor comun al nr-lor:"<<n1<<" și ";
    cout<<n2<<" este:"<<cmmdc(n1,n2)<<'\\n';*/ }

```

Exercițiu: Să se calculeze valoarea lui y , u și m fiind citite de la tastatură:

$z=2\omega(2\varphi(u)+1, m) + \omega(2u^2-3, m+1)$, unde:

$\omega(x, n) = \sum_{i=1}^n \sin(ix) \cos(2ix)$, $\varphi(x) = \sqrt{1+e^{-x^2}}$, $\omega: \mathbb{R} \times \mathbb{N} \rightarrow \mathbb{R}$, $\varphi: \mathbb{R} \rightarrow \mathbb{R}$

```

#include <iostream.h>
#include <math.h>
double omega(double x, int n)
{ double s=0; int i;
  for (i=1; i<=n; i++)    s+=sin(i*x)*cos(i*x);
  return s;
}

double psi( double x)
{ return sqrt( 1 + exp (- pow (x, 2)) );}

void main()
{double u, z; int m; cout<<"u="; cin>>u; cout<<"m="; cin>>m;
  z=2*omega(2* psi(u) + 1, m) + omega(2*pow(u,2) - 3, m+1);
  cout<<"z="<<z<<'\\n'; }

```

În exemplele anterioare, înainte de apelul funcțiilor folosite, acestea au fost definite (antet+corp). Există cazuri în care definirea unei funcții nu poate fi făcută înaintea apelului acesteia (cazul funcțiilor care se apelează unele pe altele). Să rezolvăm ultimul exercițiu, folosind declarațiile funcțiilor *omega* și *psi*, și nu definițiile lor.

Exercițiu:

```

#include <iostream.h>
#include <math.h>
double omega(double, int);
// prototipul funcției omega - antet din care lipsesc numele parametrilor formali
double psi(double); // prototipul funcției psi

void main()
{double u, z; int m; cout<<"u="; cin>>u; cout<<"m="; cin>>m;
  z=2*omega(2* psi(u) + 1, m) + omega(2*pow(u,2) - 3, m+1);
  cout<<"z="<<z<<'\\n'; }

double omega(double x, int i); // definiția funcției omega
{ double s=0; int i;
  for (i=1; i<=n; i++)    s +=  sin (i*x) * cos (i*x);
  return s; }

double psi( double x) // definiția funcției psi

```

```
{    return sqrt( 1 + exp ( - pow (x, 2))) ; }
```

Prototipurile funcțiilor din biblioteci (predefinite) se găsesc în headere. Utilizarea unei astfel de funcții impune doar includerea în program a headerului asociat, cu ajutorul directivei preprocesor **#include**.

Programatorul își poate crea propriile headere, care să conțină declarații de funcții, tipuri globale, macrodefiniții, etc.

Similar cu declarația de variabilă, domeniul de valabilitate (vizibilitate) a unei funcții este:

- fișierul sursă, dacă declarația funcției apare în afara oricărei funcții (la nivel global);
- funcția sau blocul în care apare declarația.

3. Transferul parametrilor unei funcții

Funcțiile comunică între ele prin argumente (parametri).

Există următoarele moduri de transfer (transmitere) a parametrilor către funcțiile apelate:

- Transfer prin valoare;
- Transfer prin pointeri;
- Transfer prin referință.

3.1. Transferul parametrilor prin valoare. În exemplele anterioare, parametri de la funcția apelantă la funcția apelată au fost transmiși **prin valoare**. De la programul apelant către funcția apelată, prin apel, se transmit valorile parametrilor efectivi, reali. Aceste valori vor fi atribuite, la apel, parametrilor formali. Deci procedeul de transmitere a parametrilor prin valoare constă în *încărcarea valorii parametrilor efectivi în zona de memorie a parametrilor formali (în stivă)*. La apelul unei funcții, parametri reali trebuie să corespundă - ca ordine și tip - cu cei formali.

Exercițiu: Să se scrie următorul program (care ilustrează legătura dintre pointeri și vectori) și să se urmărească rezultatele execuției acestuia.

```
void f1(float intr, int nr) // intr, nr - parametri formali
{
    for (int i=0; i<nr; i++) intr *= 2.0;
    cout<<"Val. Param. intr="<<intr<<"\n" // intr=12
}
void main()
{
    float data=1.5; f1(data,3);
    // apelul funcției f1; data, 3 sunt parametri efectivi
    cout<<"data="<<data<<"\n" ;
    // data=1.5 (nemodificat)
}
```

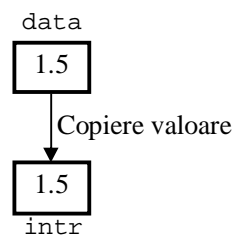


Figura 1. Transmiterea prin valoare.

Fiecare argument efectiv utilizat la apelul funcției este evaluat, iar valoarea este atribuită parametrului formal corespunzător. În interiorul funcției, o copie locală a acestei valori va fi memorată în parametrul formal. O modificare a valorii parametrului formal în interiorul funcției (printr-o operație din corpul funcției), nu va modifica valoarea parametrului efectiv, ci doar valoarea parametrului formal, deci a copiei locale a parametrului efectiv (figura 1). Faptul că variabila din programul apelant (parametrul efectiv) și parametrul formal sunt obiecte distincte, poate constitui un mijloc util de

protecție. Astfel, în funcția `f1`, valoarea parametrului formal *intr* este modificată (alterată) prin instrucțiunea ciclică *for*. În schimb, valoarea parametrului efectiv (*data*) din funcția apelantă, rămâne nemodificată.

În cazul transmiterii parametrilor prin valoare, parametri efectivi pot fi chiar expresii. Acestea sunt evaluate, iar valoarea lor va inițializa, la apel, parametri formali.

Exemplu:

```
double psi(int a, double b)
{
    if (a > 0) return a*b*2;
    else return -a+3*b; }
void main()
{ int x=4; double y=12.6, z; z=psi ( 3*x+9, y-5) +
28;
cout<<"z="<<z<<'\\n';    }
```

Transferul valorii este însoțit de eventuale conversii de tip. Aceste conversii sunt realizate automat de compilator, în urma verificării apelului de funcție, pe baza informațiilor despre funcție, sau sunt conversii explicite, specificate de programator, prin operatorul "cast".

Exemplu:

```
float f1(double, int);
void main()
{
    int a, b; float g=f1(a, b);    // conversie automată: int a -> double a
    float h=f1( (double) a, b);    // conversie explicită
}
```

Limbajul *C* este numit **limbajul apelului prin valoare**, deoarece, de fiecare dată când o funcție transmite argumente unei funcții apelate, este transmisă, de fapt, o copie a parametrilor efectivi. În acest mod, dacă valoarea parametrilor formali (inițializați cu valorile parametrilor efectivi) se modifică în interiorul funcției apelate, valorile parametrilor efectivi din funcția apelantă nu vor fi afectate.

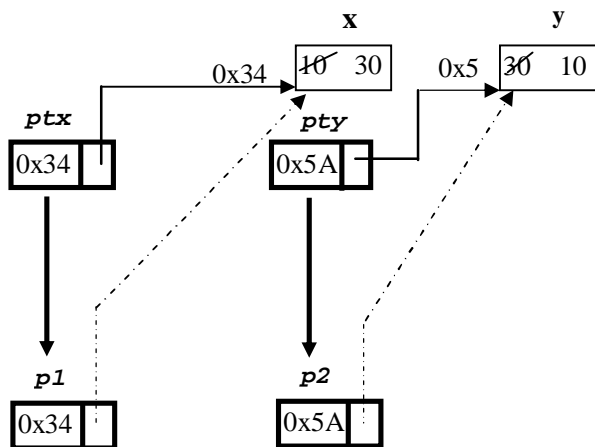
3.2. Transferul parametrilor prin pointeri. În unele cazuri, parametri transmiși unei funcții pot fi pointeri (variabile care conțin adrese). În aceste cazuri, parametri formali ai funcției apelate vor fi inițializați cu valorile parametrilor efectivi, deci cu valorile unor adrese. Astfel, *funcția apelată poate modifica conținutul locațiilor spre care pointează argumentele (pointerii).*

Exercițiu: Să se citească 2 valori întregi și să se interschimbe cele două valori. Se va folosi o funcție de interschimbare.

```
#include <iostream.h>
void schimbă(int *, int *);          //prototipul funcției schimba
void main()
{ int x, y, *ptx, *pty;    ptx=&x;    pty=&y;
cout<<"x=";cin>>x;cout<<"y=";cin>>y;cout<<"x="<<x;cout<<"y="<<y<<'\\n' ;
cout<<"Adr. lui x:"<<&x<<" Val lui x:"<<x<<'\\n';
cout<<"Adr.lui    y:"<<&y<<"    Val    y:"<<y<<'\\n';    cout<<"Val.    lui
ptx:"<<ptx;
cout<<" Cont. locației spre care pointează ptx:"<<*ptx<<'\\n';
cout<<"Val. lui pty:"<<pty;
cout<<"Cont. locației spre care pointează pty:"<<*pty;
schimbă(ptx, pty);
```

```
// SAU: schimba(&x, &y);
cout<<"Adr. lui x:"<<&x<<" %x Val lui x: %d\n", &x, x);
cout<<"Adr. y:"<<&y<<" Val lui y:"<<y<<"\n";cout<<"Val. lui
ptx:"<<ptx;
cout<<" Cont. locației spre care pointează ptx:"<<*ptx<<"\n";
cout<<"Val. lui pty:"<<pty;
cout<<" Cont. locației spre care pointează pty:"<<*pty<<"\n";
}
void schimbă( int *p1, int *p2)
{
cout<<"Val. lui p1:"<<p1;
cout<<" Cont. locației spre care pointează p1:"<<*p1<<"\n";
cout<<"Val. lui p2:"<<p2;
cout<<" Cont. locației spre care pointează p2:"<<*p2<<"\n";
int t = *p1; //int *t; t=p1;
*p2=*p1; *p1=t;
cout<<"Val. lui p1:"<<p1;
cout<<" Cont. locației spre care pointează p1:"<<*p1<<"\n";
cout<<"Val. lui p2:"<<p2;
cout<<" Cont. locației spre care pointează p2:"<<*p2<<"\n";
}
```

Dacă parametri funcției schimbă ar fi fost transmiși prin valoare, această funcție ar fi interschimbat copiile parametrilor formali, iar în funcția main modificările asupra parametrilor transmiși nu s-ar fi păstrat. În figura 2 sunt prezentate mecanismul de transmitere a parametrilor (prin pointeri) și modificările efectuate asupra lor de către funcția schimbă.



Parametri formali p1 și p2, la apelul funcției schimbă, primesc valorile parametrilor efectivi ptx și pty, care reprezintă adresele variabilelor x și y.

Astfel, variabilele pointer p1 și ptx, respectiv p2 și pty pointează către x și y. Modificările asupra valorilor variabilelor x și y realizate în corpul funcției schimbă, se păstrează și în funcția main.

Figura 2. Transmiterea parametrilor unei funcții prin pointeri.

Exercițiu: Să se scrie următorul program și să se urmărească rezultatele execuției acestuia.

```
#include <iostream.h>
double omega (long *k)
{
    cout<<"k=", k);
    // k conține adr. lui i
    cout<<"*k=";
    cout<<k<<'\\n';
    // *k = 35001
    double s=2+(*k)-3;
    // s = 35000
    cout<<"s="<<s<<'\\n';
    *k+=17; // *k = 35017
    cout<<"*k="<<*k;
    cout<<'\\n';
    return s;
}
```

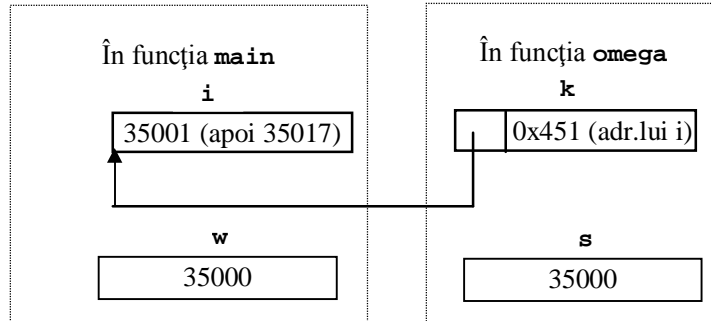


Figura 3. Transferul parametrilor prin pointeri.

```
void main()
{
    long i = 35001;    double w;
    cout<<"i="<<i;cout<<"Adr.lui i:"<<&i<<'\\n';
    w=omega(&i); cout<<"i="<<i<< " w="<<w<<'\\n';    // i = 350017 w = 35000
}
```

Funcții care returnează pointeri. Valoarea returnată de o funcție poate fi pointer, așa cum se observă în exemplul următor:

Exemplu:

```
#include <iostream.h>
double *f (double *w, int k)
{
    // w conține adr. de început a vectorului a
    cout<<"w="<<w<<" *w="<<*w<<'\\n';    // w= adr. lui a;*w = a[0]=10
    return w+=k;
    /*incrementeaza pointerul w cu 2(val. lui k); deci w pointează către elementul de indice 2 din vectorul a*/
}
void main()
{double a[10]={10,1,2,3,4,5,6,7,8,9}; int i=2;
    cout<<"Adr. lui a este:"<<a;
    double *pa=a;    // double *pa; pa=a;
    cout<<"pa="<<pa<<'\\n' // pointerul pa conține adresa de început a tabloului a
    // a[i] = * (a + i)
    // &a[i] = a + i
    pa=f(a,i); cout<<"pa="<<pa<<" *pa="<<*pa<<'\\n';
    // pa conține adr. lui a[2], adica adr. a + 2 * sizeof(double);
    *pa=2;
}
```

3.3. Transferul parametrilor prin referință. În acest mod de transmitere a parametrilor, unui parametru formal i se poate asocia (atribui) chiar obiectul parametrului efectiv. Astfel, parametrul efectiv poate fi modificat direct prin operațiile din corpul funcției apelate.

În exemplul următor definim variabila `br`, **variabilă referință** către variabila `b`. Variabilele `b` și `br` se găsesc, în memorie, la *aceeași* adresă și sunt variabile sinonime.

Exemplu:

```
#include <stdio.h>
#include <iostream.h>
void main()
{
    int b,c;
    int &br=b; //br referință la altă variabilă (b)
    br=7;
    cout<<"b="<<b<<"\n"; //b=7
    cout<<"br="<<br<<"\n"; //br=7
    cout<<"Adr. br este:"<<&br; //Adr. br este:0xffff4
    cout<<"Adr. b este:"<<&b<<"\n"; //Adr. b este:0xffff4
    b=12; cout<<"br="<<br<<"\n"; //br=12
    cout<<"b="<<b<<"\n"; //b=12
    c=br; cout<<"c="<<c<<"\n"; //c=12
}
```

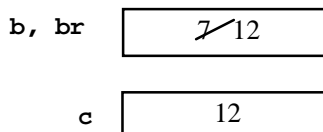


Figura 4. Variabilele referință `b`, `br`.

Exemplul devenit clasic pentru explicarea apelului prin referință este cel al funcției de permutare (interschimbare) a două variabile.

Fie funcția *schimb* definită astfel:

```
void schimb (double x, double y)
{ double t=x; x=y; y=t; }

void main()
{ double a=4.7, b=9.7;
  . . . . .
  schimb(a, b); // apel funcție
  . . . . . }
```

Parametri funcției *schimb* sunt transmiși prin valoare: parametrilor formali `x`, `y` li se atribuie (la apel) valorile parametrilor efectivi `a`, `b`. Funcția *schimb* permută valorile parametrilor formali `x` și `y`, dar permutarea nu are efect asupra parametrilor efectivi `a` și `b`.

Pentru ca funcția de interschimbare să poată permuta valorile parametrilor efectivi, în limbajul C/C++ parametri formali trebuie să fie *pointeri către valorile care trebuie interschimbate*:

```
void pschimb(double *x, double *y)
{ int t=*x; *x=*y; *y=t; }
void main()
{ double a=4.7, b=9.7;
  . . . . .
  pschimb(&a, &b); // apel funcție
  /* SAU:
  double *pa, *pb;
  pa=&a; pb=&b;
  pschimb(pa, pb);*/
  . . . . . }
```

Se atribuie pointerilor `x` și `y` valorile pointerilor `pa`, `pb`, deci adresele variabilelor `a` și `b`. Funcția *pschimb* permută valorile spre care pointează pointerii `x` și `y`, deci valorile lui `a` și `b` (figura 5).

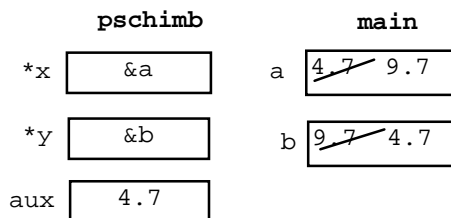


Figura 5. Transferul parametrilor prin pointeri.

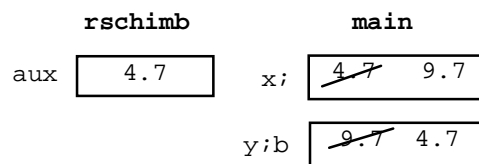


Figura 6. Transferul parametrilor prin referință.

În limbajul C++ aceeași funcție de permutare se poate defini cu parametri formali de *tip referință*.

```
void rschimb(double &x, double &y)
{ int t=x; x=y; y=t; }
void main()
{ double a=4.7, b=9.7;
  . . . . .
  rschimb(a, b);          // apel funcție
  . . . . . }
```

În acest caz, x și y sunt sinonime cu a și b (nume diferite pentru aceleași grupuri de locații de memorie). Interschimbarea valorilor variabilelor de x și y înseamnă interschimbarea valorilor variabilelor a și b (figura 6).

Comparând funcțiile pschimb și rschimb, se observă că *diferența dintre ele constă în modul de declarare a parametrilor formali*. În cazul funcției pschimb parametri formali sunt *pointeri* (de tip *double), în cazul funcției rschimb, parametri formali sunt *referințe* către date de tip double. În cazul transferului parametrilor prin referință, parametri formali ai funcției referă aceleași locații de memorie (sunt sinonime pentru) parametri efectivi.

Comparând cele trei moduri de transmitere a parametrilor către o funcție, se poate observa:

1. La apelul *prin valoare* transferul datelor este *unidirecțional*, adică valorile se transferă numai de la funcția apelantă către cea apelată. La apelul *prin referință* transferul datelor este *bidirecțional*, deoarece o modificare a parametrilor formali determină modificarea parametrilor efectivi, care sunt sinonime (au nume diferite, dar referă aceleași locații de memorie).
2. La transmiterea parametrilor *prin valoare*, ca parametri efectivi pot apare *expresii* sau *nume de variabile*. La transmiterea parametrilor *prin referință*, ca parametri efectivi nu pot apare expresii, ci *doar nume de variabile*. La transmiterea parametrilor *prin pointeri*, ca parametri efectivi pot apare expresii de pointeri.
3. Transmiterea parametrilor unei funcții prin referință este specifică limbajului C++.
4. Limbajul C este numit limbajul apelului prin valoare. Apelul poate deveni, însă, *apel prin referință* în cazul variabilelor simple, folosind pointeri, sau așa cum vom vedea în [paragraful 6.4.](#), în cazul în care parametru efectiv este un tablou.
5. În limbajul C++ se poate alege, pentru fiecare parametru, tipul de apel: prin valoare sau prin referință, așa cum ilustrează exemplele următoare:

Exercițiu: Să se scrie următorul program și să se urmărească rezultatele execuției acestuia.

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

double func(int a, double b, double *c, double &d)
{cout<<"***** func *****\n";
cout<<"a="<<a<<" b="<<b;           //a=7 (a=t prin val); b=21 (b=u prin val)
cout<<" c="<<c<<" *c="<<*c<<"\n";    // c=ffe(c=w=&u) *c=21
cout<<" d="<<d;           //d=17
cout<<"Adr d="<<&d<<"\n";           //Adr d=ffe6 (adr d=adr v)
a+=2; cout<<"a="<<a<<"\n";           //a=9
d=2*a+b; cout<<"d="<<d<<"\n";       //d=39
/*c=500;
cout<<" c="<<c<<" *c="<<*c<<"\n";    // c=ffe(c=w=&u) *c=21*/
cout<<"***** func *****\n";
return b+(*c);
}

void main()
{cout<<"\n\n \n MAIN MAIN";
int t=7; double u=12, v=17, *w, z;
cout<<"u="<<u<<"\n";           //u=12
w=&u; *w=21;
cout<<"t="<<t<<" u="<<u<<" v="<<v;   //t=7 u=12 v=17 *w=21
cout<<" *w="<<*w<<" u="<<u<<"\n";    //*w=21 u=21
printf("w=%x Adr. u=%x\n",w,&u);    //w=ffee Adr. u=ffee
printf("v=%f Adr v=%x\n",v,&v);     //v=17.000 Adr v=ffe6
z=func(t,u,w, v);
cout<<"t="<<t<<"u="<<u<<"v="<<v;     //t=7 u=21 (NESCHIMBATI) v=39 (v=d)
cout<<" *w="<<*w<<" z="<<z<<"\n";    //*w=21 w=ffee z=42
printf(" w=%x\n",w);
}
```

Exemplul ilustrează următoarele probleme. La apelul funcției `func`, parametri `t` și `u` sunt transmiși prin valoare, deci valorile lor vor fi atribuite parametrilor formali `a` și `b`. Orice modificare a parametrilor formali `a` și `b`, în funcția `func`, nu va avea efect asupra parametrilor efectivi `t` și `u`. Al treilea parametru formal al funcției `func` este transmis prin pointeri, deci `c` este de tip `double *` (pointer către un real), sau `*c` este de tip `double`. La apelul funcției, valoarea pointerului `w` (adresa lui `u` : `w=&u`) este atribuită pointerului `c`. Deci pointerii `w` și `c` conțin aceeași adresă, pointând către un real. Dacă s-ar modifica valoarea spre care pointează `c` în `func` (vezi instrucțiunile din comentariu `*c=500`), această modificare ar fi reflectată și în funcția apelantă, deoarece pointerul `w` are același conținut ca și pointerul `c`, deci pointează către aceeași locație de memorie. Parametrul formal `d` se transmite prin referință, deci, în momentul apelului, `d` și `v` devin similare (`d` și `v` sunt memorate la aceeași adresă). Modificarea valorii variabilei `d` în `func` se reflectă, deci, și asupra parametrului efectiv din funcția `main`.

Exercițiu: Să se scrie următorul program (care ilustrează legătura dintre pointeri și vectori) și să se urmărească rezultatele execuției acestuia.

```
#include <iostream.h>
#include <stdio.h>
double omega(long &k)
{printf("Adr k=%x Val k=%ld\n",&k,k);          //Adr k=fff2 Val k=200001
double s=2+k-3;cout<<"s="<<s<<'\\n';          //s=200000
k+=17;printf("Adr k=%x Val k=%ld\n",&k,k);      //Adr k=fff2 Val k=200018
return s;
}
void main()
{long a=200001;
printf("Adr a=%x Val a=%ld\n",&a,a);          //Adr a=fff2 Val a=200001
double w=omega(a); cout<<"w="<<w<<'\\n';      //s=200000
}
```

Modificatorii sunt cuvinte cheie utilizați în declarații sau definiții de variabile sau funcții. Modificatorul de acces **const** poate apare în:

- ❑ Declarația unei variabile (precede tipul variabilei) restricționând modificarea valorii datei;
- ❑ La declararea variabilelor pointeri definind pointeri constanți către date neconstante, pointeri neconstanți către date constante și pointeri constanți către date constante.
- ❑ În lista declarațiilor parametrilor formali ai unei funcții, conducând la imposibilitatea de a modifica valoarea parametrului respectiv în corpul funcției, ca în exemplul următor:

Exemplu:

```
#include <iostream.h>
#include <stdio.h>
int func(const int &a)
{printf("Adr a=%x Val a=%d\n",&a,a);int b=2*a+1;
//modificarea valorii parametrului a nu este permisă
cout<<"b="<<b<<'\\n';return b;}
void main()
{const int c=33;int u;printf("Adr c=%x Val c=%d\n",&c,c);
u=func(c);cout<<"u="<<u<<'\\n'; }
```

3.4. Transferul parametrilor către funcția main. În situațiile în care se dorește transmiterea a unor informații (opțiuni, date inițiale, etc) către un program, *la lansarea în execuție* a acestuia, este necesară definirea unor parametri către funcția main. Se utilizează trei parametri speciali: argc, argv și env. Trebuie inclus headerul **stdarg.h**.

Prototipul funcției main cu parametri în linia de comandă este:

```
main (int argc, char *argv[ ], char *env[ ])
```

Dacă **nu se lucrează cu un mediu de programare integrat**, argumentele transmise către funcția main trebuie editate (specificate) în linia de comandă prin care se lansează în execuție programul respectiv. Linia de comandă tastată la lansarea în execuție a programului este formată din grupuri de caractere delimitate de spațiu sau tab. Fiecare grup este memorat într-un șir de caractere. Dacă se lucrează cu un mediu integrat (de exemplu, BorlandC), selecția comenzii Arguments... din meniul Run

determină afișarea unei casete de dialog în care utilizatorul poate introduce argumentele funcției `main`.

- Adresele de început ale acestor șiruri sunt memorate în tabloul de pointeri `argv[]`, în ordinea în care apar în linia de comandă (`argv[0]` memorează adresa șirului care constituie numele programului, `argv[1]` - adresa primului argument, etc.).
- Parametrul întreg `argc` memorează numărul de elemente din tabloul `argv` (`argc>=1`).
- Parametrul `env[]` este un tablou de pointeri către șiruri de caractere care pot specifica parametri ai sistemului de operare.

Funcția `main` poate returna o valoare întreagă. În acest caz în antetul funcției se specifică la tipul valorii returnate `int`, sau nu se specifică nimic (implicit, tipul este `int`), iar în corpul funcției apare instrucțiunea `return valoare_intreagă`; Numărul returnat este transferat sistemului de operare (programul apelant) și poate fi tratat ca un cod de eroare sau de succes al încheierii execuției programului.

Exercițiu: Să se implementeze un program care afișează argumentele transmise către funcția `main`.

```
#include <iostream.h>
#include <stdarg.h>
void main(int argc, char *argv[], char *env[])
{cout<<"Nume program:"<<argv[0]<<'\n';//argv[0] contine numele programului
if(argc==1)
    cout<<"Lipsa argumente!\n";
else
    for (int i=1; i<argc; i++){
        cout<<"Argumentul "<<i<<": "<<argv[i]<<'\n';
    }
}
```

4. Tablouri ca parametri

În limbajul C, cazul parametrilor tablou constituie o excepție de la regula transferului parametrilor prin valoare. Numele unui tablou reprezintă, de fapt, adresa tabloului, deci a primului element din tablou.

Exercițiu: Să se afle elementul minim dintr-un vector de maxim 10 elemente. Se vor scrie două funcții: de citire a elementelor vectorului și de aflare a elementului minim:

```
#include <iostream.h>
int min_tab(int a[], int nr_elem)
{int elm=a[0];
for (int ind=0; ind<nr_elem; ind++)
    if (elm>=a[ind]) elm=a[ind];
return elm;
}
void citireVector(int b[], int nr_el)
{ for (int ind=0; ind<nr_el; ind++){
    cout<<"Elem "<<ind+1<<"="; cin>>b[ind];}
}
void main()
{
int a[10]; int i,j,n; cout<<"n="; cin>>n;
citireVector(a,n);
int min=min_tab(a,n); cout<<"Elem. min:"<<min<<'\n'; }
}
```

Aceleași problemă poate fi implementată folosind aritmetica pointerilor:

```
#include <iostream.h>
void citireVector(int *b, int nr_el)
{ for (int ind=0; ind<nr_el; ind++){
    cout<<"Elem "<<ind+1<<"="; cin>>*b(ind+ind);}
}
int min_tab(int *a, int nr_elem)
{int elm=*a;
for (int ind=0; ind<nr_elem; ind++)
    if ( elm>=*(a+ind) )    elm=*(a+ind);
return elm;
}
void main()
{
int a[10]; int i,j,n; cout<<"n="; cin>>n;
citireVector(a, n);
int min=min_tab(a,n);
cout<<"Elem. min:"<<min<<'\n';
}
```

Din exemplele anterioare se poate observa:

1. Prototipul funcției `min_tab` poate fi unul dintre:

```
int min_tab(int a[], int nr_elem);
int min_tab(int *a, int nr_elem);
```
2. Echivalențe:

```
int *a    ⇔    int a[]
a[i]      ⇔    *(a+i)
```
3. Apelul funcțiilor:

```
citireVector(a,n);
int min=min_tab(a,n);
```

Pentru tablourile unidimensionale, la apel, nu trebuie specificat numărul de elemente. Dimensiunea tabloului trebuie să fie cunoscută în funcția care îl primește ca parametru. De obicei, dimensiunea tabloului se transferă ca parametru separat (`nr_elem`).

Exercițiu: Să se scrie următorul program și să se urmărească rezultatele execuției acestuia.

```
#include <iostream.h>
#include <stdio.h>
double omega(int j, double x, double t[], double *w)
{double s; cout<<"În funcția omega:";
cout<<"j="<<j<<" t[j]="<<t[j]<<" t[j+1]="<<t[j+1]<<'\n';
//j=2 (=i din main)
//t[j]=-3.21 t[j+1]=7.44
cout<<"j="<<j<<" w[j]="<<w[j]<<" w[j+1]="<<w[j+1]<<'\n';
//j=2 (=i din main)
//w[j]=-21.16 w[j+1]=92.2
t[j]=100; *(t+j+1)=200; w[j]=300; *(w+j+1)=400;
cout<<"După atribuire:\n";
cout<<"j="<<j<<" t[j]="<<t[j]<<" t[j+1]="<<t[j+1]<<'\n';
//După atribuire:
//j=2
//t[j]=100 t[j+1]=200
//w[j]=300 w[j+1]=400
cout<<"j="<<j<<" w[j]="<<w[j]<<" w[j+1]="<<w[j+1]<<'\n';
int i=2*j+1; x=x+2.29*i; s=x+2*t[0]-w[1];
```

```

cout<<"i="<<i<<" x="<<x<<" s="<<s<<"\n";
//i=5 x=1.123+2.29+5 s=x+2*1.32-(-15.34)
return s;
}
void switch1(double *x, double *y)
{double t=*x; *x=*y; *y=t;}
void switch2(double &x, double &y)
{double t; t=x;x=y;y=t;}
void main()
{double a=123, b=456, u=1.123;
int i=2;
double r[]={1.32, 2.15, -3.21, 7.44, -15.8};
double q[]={12.26, -15.34, -21.16, 92.2, 71.6};
cout<<"i="<<i<<" u="<<u<<"\n";
double y=omega(i,u,r,q);
cout<<"i="<<i<<" u="<<u<<"\n";
//i=2 u=...
cout<<"omega(i,u,r,q)=y="<<y<<"\n";
cout<<"r[i]="<<r[i]<<" r[i+1]="<<r[i+1]<<";
cout<<" q[i]="<<q[i]<<" q[i+1]="<<q[i+1]<<"\n";
//r[i]=100 r[i+1]=200 q[i]=300 q[i+1]=400
cout<<"a="<<a<<" b="<<b<<"\n"; //a=123 b=456
switch1(&a,&b);
cout<<"Rez. intersch. a="<<a<<" b="<<b<<"\n"; //a=456 b=123
switch2(a,b);
cout<<"Rez. intersch. a="<<a<<" b="<<b<<"\n"; //a=123 b=456
cout<<"r[i]="<<r[i]<<" r[i+1]="<<r[i+1]<<"\n";
switch1(r+i,r+i+1);
cout<<"Rez. intersch. r[i]="<<r[i]<<" r[i+1]="<<r[i+1]<<"\n";
switch2(r[i],r[i+1]);
//switch2(*(r+i),*(r+i+1));
cout<<"Rez. intersch. r[i]="<<r[i]<<" r[i+1]="<<r[i+1]<<"\n";
}

```

În exemplul anterior, parametri formali *i* și *x* din funcția *omega* sunt transmiși prin valoare; parametri *t* și *w* sunt parametri tablou, transmiși prin referință (referință și pointeri). În funcția *switch1* parametri sunt transmiși prin pointeri. În funcția *switch2* parametri sunt transmiși prin referință.

Pentru **tablourile multidimensionale**, pentru ca elementele tabloului să poată fi referite în funcție, compilatorul trebuie informat asupra modului de organizare a tabloului.

Pentru **tablourile bidimensionale** (vectori de vectori), poate fi omisă doar precizarea numărului de linii, deoarece pentru a adresa elementul *a[i][j]*, compilatorul utilizează relația: **&mat[i][j]=&mat+(i* N+j)*sizeof(tip)**, în care *N* reprezintă numărul de coloane, iar *tip* reprezintă tipul tabloului.

Exercițiu: Fie o matrice de maxim 10 linii și 10 coloane, ale cărei elemente se introduc de la tastatură. Să se implementeze două funcții care afișează matricea și calculează elementul minim din matrice.

```

#include <iostream.h>
int min_tab(int a[][10], int nr_lin, int nr_col)
{int elm=a[0][0];
for (int il=0; il<nr_lin; il++)
    for (int ic=0; ic<nr_col; ic++)
        if (elm>a[il][ic]) elm=a[il][ic];
return elm;
}

```

```

}
void afisare(int a[][10], int nr_lin, int nr_col)
{
    for (int i=0; i<nr_lin; i++)
        {for (int j=0; j<nr_col; j++) cout<<a[i][j]<<'\\t';
          cout<<'\\n';
         }
    }
void main()
{
    int mat[10][10];int i, j, M, N;cout<<"Nr. linii:"; cin>>M;
    cout<<"Nr. coloane:"; cin>>N;
    for (i=0; i<M; i++)
        for (j=0; j<N; j++)
            { cout<<"mat["<<i<<" "<<j<<"]="; cin>>mat[i][j];}
    afisare(mat, M, N);
    int min=min_tab(mat, M, N);
    cout<<"Elem. min:"<<min<<'\\n';
}

```

Valoarea returnată de o funcție poate să fie transmisă și prin referință, cum ilustrează exemplul următor:

Exemplu:

```

#include <iostream.h>
#include <stdio.h>
double &func(double &a, double b)
{ printf("În funcție:\\n");
  printf("Val a=%f Adr a=%x\\n", a, &a);      //Val a=1.20 Adr a=fffe
  cout<<"b="<<b<<'\\n';                    //b=2.2
  a=2*b+1; printf(" După atrib: val a=%f Adr a=%x\\n", a, &a);
  //Val a=5.40 Adr a=fffe
  return a;
}
void main()
{double c=1.2;cout<<"*****MAIN*****\\n";
 printf("Val c=%f Adr c=%x\\n",c, &c);      //Val c=1.20 Adr c=fffe
 double d; printf("Adr. d=%x\\n", &d);     //Adr. d=ffe6
 d=func(c,2.2);
 printf("Val d=%f Adr d=%x\\n", d, &d);    //Val d=5.40 Adr d=ffe6
}

```

5. Funcții cu parametri implicați

Spre deosebire de limbajul C, în limbajul C++ se pot face inițializări ale parametrilor formali. Parametri formali inițializați se numesc **parametri implicați**. De exemplu, antetul funcției `cmmdc` (care calculează și returnează cel mai mare divizor comun al numerelor întregi primite ca argumente) poate avea această formă:

```
int cmmdc(int x, int y=1);
```

Parametrul formal `y` este inițializat cu valoarea 1 și este *parametru implicit*. La apelul funcțiilor cu parametri implicați, unui parametru implicit, *poate să-i corespundă sau nu*, un parametru efectiv. Dacă la apel nu îi corespunde un parametru efectiv, atunci parametrul formal va primi valoarea prin care a fost inițializat (valoarea implicită). Dacă la apel îi corespunde un parametru efectiv, parametrul formal va fi inițializat cu valoarea acestuia, neglijându-se astfel valoarea implicită a parametrului formal. În exemplul

anterior, la apelul: `int div=cmmdc(9);` `x` va lua valoarea 9, iar `y` va lua valoarea 1 (implicită).

Dacă în lista de parametri formali ai unei funcții există și parametri implicați și parametri *neinițializați*, parametri implicați trebuie să ocupe ultimele poziții în listă, nefiind permisă intercalarea acestora printre parametri neinițializați.

6. Funcții cu număr variabil de parametri

În limbajele C și C++ se pot defini funcții cu un număr variabil de parametri. Parametri care trebuie să fie prezenți la orice apel al funcției se numesc **parametri fiși**, ceilalți se numesc **parametri variabili**. Parametri fiși preced parametri variabili. Prezența parametrilor variabili se indică în antetul funcției prin *trei puncte* care se scriu după ultimul parametru fix al funcției.

De exemplu, fie antetul funcției numite `vârf`:

```
void vârf (int n, double a, . . . )
```

Funcția `vârf` are *doi parametri fiși* (`n` și `a`) și **parametri variabili**, pentru care nu se precizează în prealabil numărul și tipul; numărul și tipul parametrilor variabili diferă de la un apel la altul.

Funcțiile cu un număr variabil de parametri sunt, de obicei, funcții de bibliotecă (ex: `printf`, `scanf`) și se definesc folosind niște macroui speciale care permit accesul la parametri variabili și se găsesc în headerul `stdarg.h`.

7. Funcții predefinite

Orice mediu de programare este prevăzut cu una sau mai multe biblioteci de funcții predefinite. Orice bibliotecă este formată din:

- ❑ fișierele header (conține prototipurile funcțiilor, declarațiile de variabile);
- ❑ biblioteca (arhiva) propriu-zisă (conține definiții de funcții).

Pentru ca funcțiile predefinite să poată fi utilizate, fișierele header în care se găsesc prototipurile acestora trebuie inclus în funcția (programul) apelant printr-o directivă preprocesor (exemplu `#include <stdio.h>`). De asemenea, utilizatorul își poate crea propriile headere proprii. Pentru a putea utiliza funcțiile proprii, el trebuie să includă aceste headere în programul apelant (exemplu `#include "my_header.h"`).

Pentru funcțiile predefinite, au fost create fișiere header orientate pe anumite tipuri de aplicații. De exemplu, funcțiile matematice se găsesc în headerul `<math.h>`. Headerul `<stdlib.h>` care conține funcții standard. Headerul `<values.h>` definește o serie de constante simbolice (exemplu `MAXINT`, `MAXLONG`) care reprezintă, în principal, valorile maxime și minime ale diferitelor tipuri de date.

7.1. Funcții matematice (headerul <math.h>).

➤ Funcții aritmetice

Valori absolute

int abs(int x);

Returnează un întreg care reprezintă valoarea absolută a argumentului.

long int labs(long int x);

Analog cu funcția abs, cu deosebirea că argumentul și valoarea returnată sunt de tip long int.

double fabs(double x);

Returnează un real care reprezintă valoarea absolută a argumentului real.

➤ Funcții de rotunjire

double floor(double x);

Returnează un real care reprezintă cel mai apropiat număr, fără zecimale, mai mic sau egal cu x (rotunjire prin lipsă).

double ceil(double x);

Returnează un real care reprezintă cel mai apropiat număr, fără zecimale, mai mare sau egal cu x (rotunjire prin adaos).

➤ Funcții trigonometrice

double sin(double x);

Returnează valoarea lui sin(x), unde x este dat în radiani. Numărul real returnat se află în intervalul [-1, 1].

double cos(double x);

Returnează valoarea lui cos(x), unde x este dat în radiani. Numărul real returnat se află în intervalul [-1, 1].

double tan(double x);

Returnează valoarea lui tg(x), unde x este dat în radiani.

➤ Funcții trigonometrice inverse

double asin(double x);

Returnează valoarea lui arcsin(x), unde x se află în intervalul [-1, 1]. Numărul real returnat (în radiani) se află în intervalul [-pi/2, pi/2].

double acos(double x);

Returnează valoarea lui arccos(x), unde x se află în intervalul [-1, 1]. Numărul real returnat se află în intervalul [0, pi].

double atan(double x);

Returnează valoarea lui arctg(x), unde x este dat în radiani. Numărul real returnat se află în intervalul [0, pi].

double atan2(double y, double x);

Returnează valoarea lui tg(y/x), cu excepția faptului ca semnele argumentelor x și y permit stabilirea cadranelor și x poate fi zero. Valoarea returnată se află în intervalul [-pi, pi]. Dacă x și y sunt coordonatele unui punct în plan, funcția returnează valoarea unghiului format de dreapta care unește originea axelor carteziane cu punctul, față de axa absciselor. Funcția folosește, de asemenea, la transformarea coordonatelor carteziane în coordonate polare.

➤ Funcții exponențiale și logaritmice

double exp(double x);

long double exp(long double x);

Returnează valoarea e^x .

double log(double x);

Returnează logaritmul natural al argumentului ($\ln(x)$).

double log10(double x);

Returnează logaritmul zecimal al argumentului ($\lg(x)$).

double pow(double baza, double exponent);

Returnează un real care reprezintă rezultatul ridicării bazei la exponent ($baza^{\text{exponent}}$).

double sqrt(double x);

Returnează rădăcina pătrată a argumentului \sqrt{x} .

double hypot(double x, double y);

Funcția distanței euclidiene - returnează $\sqrt{x^2 + y^2}$, deci lungimea ipotenuzei unui triunghi dreptunghic, sau distanța punctului P(x, y) față de origine.

➤ **Funcții de generare a numerelor aleatoare**

int rand(void) **<stdlib.h>**

Generează un număr aleator în intervalul [0, RAND_MAX].

7.2. Funcții de clasificare (testare) a caracterelor. Au prototipul în headerul <ctype.h>. Toate aceste funcții primesc ca argument un caracter și returnează un număr întreg care este pozitiv dacă argumentul îndeplinește o anumită condiție, sau valoarea zero dacă argumentul nu îndeplinește condiția.

int isalnum(int c);

Returnează valoare întreagă pozitivă dacă argumentul este literă sau cifră. Echivalentă

cu: `isalpha(c) || isdigit(c)`

int isalpha(int c);

Testează dacă argumentul este literă mare sau mică. Echivalentă cu

`isupper(c) || islower(c)`.

int iscntrl(int c);

Testează dacă argumentul este caracter de control (neimprimabil).

int isdigit(int c);

Testează dacă argumentul este cifră.

int isxdigit(int c);

Testează dacă argumentul este cifră hexagesimală (0-9, a-f, A-F).

int islower(int c);

Testează dacă argumentul este literă mică.

int isupper(int c);

Testează dacă argumentul este literă mare.

int ispunct(int c);

Testează dacă argumentul este caracter de punctuație (caracter imprimabil, dar nu literă sau spațiu).

int isspace(int c);

Testează dacă argumentul este spațiu alb (' ', '\n', '\t', '\v', '\r')

int isprint(int c);

Testează dacă argumentul este caracter imprimabil, inclusiv blancul.

7.3. Funcții de conversie a caracterelor (prototip în <ctype.h>).

int tolower(int c);

Funcția schimbă caracterul primit ca argument din literă mare, în literă mică și returnează codul ASCII al literei mici. Dacă argumentul nu este literă mare, codul returnat este chiar codul argumentului.

int toupper(int c);

Funcția schimbă caracterul primit ca argument din literă mică, în literă mare și returnează codul acesteia. Dacă argumentul nu este literă mică, codul returnat este chiar codul argumentului.

7.4. Funcții de conversie din șir în număr (de citire a unui număr dintr-un șir).

(prototip în <stdlib.h>)

long int atol(const char *npr);

Funcția convertește șirul transmis ca argument (spre care pointează npr) într-un număr cu semn, care este returnat ca o valoare de tipul long int. Șirul poate conține caracterele '+' sau '-'. Se consideră că numărul este în baza 10 și funcția nu semnalizează eventualele erori de depășire care pot apare la conversia din șir în număr.

int atoi(const char *sir);

Converteste șirul spre care pointeaza sir într-un număr întreg.

double atof(const char *sir);

Funcția convertește șirul transmis ca argument într-un număr real cu semn (returnează valoare de tipul double). În secvența de cifre din șir poate apare litera 'e' sau 'E' (exponentul), urmată de caracterul '+' sau '-' și o altă secvență de cifre. Funcția nu semnalează eventualele erori de depășire care pot apare.

7.5. Funcții de terminare a unui proces (program).

(prototip în <process.h>)

void exit(int status);

Termină execuția unui program. Codul returnat de terminarea corectă este memorat în constanta simbolică EXIT_SUCCES, iar codul de eroare - în EXIT_FAILURE.

void abort();

Termină forțat execuția unui program.

int system(const char *comanda); prototip în <system.h>

Permite execuția unei comenzi DOS, specificate prin șirul de caractere transmis ca parametru.

7.6. Funcții de intrare/ieșire (prototip în <stdio.h>). Streamurile (fluxurile de date) implicite sunt: stdin (fișierul, dispozitivul standard de intrare), stdout (fișierul, dispozitivul standard de ieșire), stderr (fișier standard pentru erori), stdprn (fișier standard pentru imprimantă) și stdaux (dispozitivul auxiliar standard). De câte ori este executat un program, streamurile implicite sunt deschise automat de către sistem. În headerul <stdio.h> sunt definite și constantele NULL (definită ca 0) și EOF (sfârșit de fișier, definită ca -1, CTRL/Z).

int getchar(void);

Citește un caracter (cu ecou) din fișierul standard de intrare (tastatură).

```
int putchar(int c);
```

Afișează caracterul primit ca argument în fișierul standard de ieșire (monitor).

```
char *gets(char *sir);
```

Citește un șir de caractere din fișierul standard de intrare (până la primul blank întâlnit sau linie nouă). Returnează pointerul către șirul citit.

```
int puts(const char *sir);
```

Afișează șirul argument în fișierul standard de ieșire și adaugă terminatorul de șir. Returnează codul ultimului caracter al șirului (caracterul care precede NULL) sau -1 în caz de eroare.

```
int printf(const char *format, ... );
```

Funcția permite scrierea în fișierul standard de ieșire (pe monitor) a datelor, într-un anumit format. Funcția returnează numărul de octeți (caractere) afișați, sau -1 în cazul unei erori.

1. Parametrul fix al funcției conține:

- Succesiuni de caractere afișate ca atare

Exemplu:

```
printf("\n Buna ziua!\n\n"); // afișare: Buna ziua!
```

- Specificatori de format care definesc conversiile care vor fi realizate asupra datelor de ieșire, din formatul intern, în cel extern (de afișare).

2. Parametri variabili ai funcției sunt expresii. Valorile obținute în urma evaluării acestora sunt afișate corespunzător specificatorilor de format care apar în parametrul fix. De obicei, parametrul fix conține atât specificatori de format, cât și alte caractere. Numărul și tipul parametrilor variabili trebuie să corespundă specificatorului de format.

Un *specificator de format* care apare în parametrul fix poate avea următoarea formă:

```
%[-|c|][sir_cifre_eventual_punct_zecimal]
```

una_sau_doua_litere

- Implicit, datele se cadrează (aliniază) la dreapta câmpului în care se scriu. Prezența caracterului - determină cadrarea la stânga.

Șirul de cifre definește dimensiunea câmpului în care se scrie data. Dacă scrierea datei necesită un câmp de lungime mai mare, lungimea indicată în specificator este ignorată. Dacă scrierea datei necesită un câmp de lungime mai mică, data se va scrie în câmp, cadrată la dreapta sau la stânga (dacă apare semnul -), completându-se restul câmpului cu caracterele ne semnificative implicite, adică spații. Șirul de cifre aflate după punct definește precizia (numărul de zecimale cu care este afișat un număr real - implicit sunt afișate 6 zecimale).

Literele definesc tipul conversiei aplicat datei afișate:

- **c** – Afișează un caracter;
- **s** – Afișează un șir de caractere;
- **d** – Afișează date întregi; cele negative sunt precedate de semnul -;
- – Afișează date de tip int sau unsigned int în octal;
- **x sau X** – Afișează date de tip int sau unsigned int în hexazecimal;
- **f** – Afișează date de tip float sau double în forma:
parte_întreagă.parte_fract;
- **e sau E** – Afișează date de tip float sau double în forma:

parte_întreagă.parte_fractionară exponent

Exponentul începe cu e sau E și definește o putere a lui zece care înmulțită cu restul numărului dă valoarea reală a acestuia;

- **g sau G**—Afișează o dată reală fie ca în cazul specificatorului terminat cu f, fie ca în cazul specificatorului terminat cu e. Criteriul de afisare se alege automat, astfel încât afișarea să ocupe un număr minim de poziții în câmpul de afișare;
- **l** – Precede una din literele d, o, x, X, u. La afișare se fac conversii din tipul *long* sau *unsigned long*;
- **L** – Precede una din literele f, e, E, g, G. La afișare se fac conversii din tipul *long double*.

```
int scanf(const char *format, ... );
```

Funcția citește din fișierul standard de intrare valorile unor variabile și le depune în memorie, la adresele specificate. Funcția returnează numărul câmpurilor citite corect.

1. *Parametrul fix* al funcției conține:

Specificatorii de format care definesc conversiile aplicate datelor de intrare, din formatul extern, în cel intern (în care sunt memorate). Specificatorii de format sunt asemanători celor folosiți de funcția *printf*: **c, s, d, o, x sau X, u, f, l, L**.

2. *Parametri variabili* reprezintă o listă de adrese ale variabilelor care vor fi citite, deci în această listă, numele unei variabile simple va fi precedată de operatorul adresă **&**.

```
int sprintf(char *sir_cu_format, const char *format, ... );
```

Funcția permite scrierea unor date în șirul transmis ca prim argument, într-un anumit format. Valoarea returnată reprezintă numărul de octeți (caractere) scrise în șir, sau -1 în cazul unei erori.

```
int sscanf(char *sir_cu_format, const char *format, ... );
```

Funcția citește valorile unor variabile din șirul transmis ca prim argument și le depune în memorie, la adresele specificate. Returnează numărul câmpurilor citite corect.

Exemplu: Să se scrie următorul program (care ilustrează modalitățile de folosire a funcțiilor predefinite) și să se urmărească rezultatele execuției acestuia.

```
#include <iostream.h>
#include <stdlib.h>
#include <math.h>
#include <ctype.h>
#include <stdio.h>

void main()
{ int x=-34;          int a=abs(x);          cout<<"a="<<a<<"\n";
  long int y=-566666;
  cout<<"labs(y)="<<labs(y)<<"fabs(-45.67)="<<fabs(-45.67)<<"\n";
  cout<<"fabs(45.67)="<<fabs(45.67)<<"\n";
  cout<<floor(78.99)<<"\n";          //78
  cout<<floor(78.45)<<"\n";          //78
  cout<<floor(-78.45)<<"\n";         //-79
  cout<<ceil(78.99)<<"\n";           //79
  cout<<ceil(78.45)<<"\n";           //79
  cout<<ceil(-78.45)<<"\n";          //-78
  cout<<isalpha('8')<<"\n";         //0
  cout<<isalpha('f')<<"\n";         //val diferita de zero
  cout<<isalpha('%')<<"\n";         //0
  cout<<tolower('D')<<"\n";         //100 (codul caracterului 'd')
  cout<<toupper('a')<<"\n";         //65 (codul caracterului 'A')
  char s1[]="-56.234 h mk";          cout<<atol(s1)<<"\n";          //-56
  cout<<atoi(s1)<<"\n";           //-56
```

```

cout<<atof(s1)<<'\\n';           //-56.234
cout<<atof("45E+3  n")<<'\\n'; //45000
cout<<"EXECUTIA COMENZII DOS DIR\\n";      int cod_ret=system("dir");
cout<<"Val. cod retur="<<cod_ret<<'\\n';
int c;cout<<"Astept car:"; c=getchar();    //Presupunem caracter introdus: e
cout<<"Caracterul citit este:"<<putchar(c);//Caracterul citit este: 101
// 101=codul carcterului e
cout<<'\\n';puts(s1);cout<<'\\n';      printf("Afisarea unui mesaj\\n");
int intreg=-45;
printf("VALOAREA VARIABILEI INTREG ESTE:%d\\n", intreg);
printf("VALOAREA VARIABILEI INTREG ESTE:%10d\\n", intreg);
printf("VALOAREA VARIABILEI INTREG ESTE:%-10d\\n", intreg);
double real=2.45;
printf("VALOAREA VARIABILEI real ESTE:%f\\n", real);
printf("VALOAREA VARIABILEI real ESTE:%10.3f\\n", real);
printf("VALOAREA VARIABILEI real ESTE:%10.5f\\n", real);
printf("VALOAREA VARIABILEI real ESTE:%e\\n", real);
printf("VAL VAR real:%f si\\neste mem. la adr.%x\\n",real,&real );
printf("astept sir:");scanf("%s",s1);
printf("Sirul citit este: %s \\n", s1);
char sir_f[100];
sprintf(sir_f,"Codul caracterului %c este:%d",c, (int)c);
puts(sir_f);
}

```

8. Clase de memorare

Definiții:

Variabilele declarate în afara oricărei funcții sunt **variabilele globale**.

Variabilele declarate în interiorul unui bloc sunt **variabilele locale**.

Porțiunea de cod în care o variabilă este accesibilă reprezintă **scopul (domeniul de vizibilitate) al variabilei** respective.

Parametri formali ai unei funcții sunt **variabile locale** ale funcției respective.

Domeniul de vizibilitate al unei variabile locale este blocul în care variabila respectivă este definită.

În situația în care numele unei variabile globale coincide cu numele unei variabile locale, *variabila locală o "maschează" pe cea globală*, ca în exemplul următor: în interiorul blocului din funcția main s-a redefinit variabila a, care este *variabilă locală* în interiorul blocului. Variabila locală a maschează variabila globală numită tot a.

Exemplu:

```

#include <stdio.h>
void main()
{ int a,b; a=1; b=2;
printf("În afara blocului a=%d b=%d\\n", a, b);
    {int a=5;          b=6;
      printf("În interiorul blocului a=%d b=%d\\n",a,b);
    }
printf("În afara blocului a=%d b=%d\\n", a, b);
}

```

În cazul variabilelor locale, compilatorul alocă memorie în momentul execuției blocului sau funcției în care acestea sunt definite. Când execuția funcției sau blocului se

termină, se eliberează memoria pentru acestea și valorile pentru variabilele locale se pierd.

Definiții:

Timpul de viață a unei variabile locale este durata de execuție a blocului (sau a funcției) în care aceasta este definită.

Timpul de viață a unei variabile globale este durata de execuție a programului.

În exemplul următor, variabila întregă `x` este vizibilă atât în funcția `main`, cât și în funcția `func1` (`x` este variabila globală, fiind definită în exteriorul oricărei funcții). Variabilele `a` și `b` sunt variabile locale în funcția `main` (vizibile doar în `main`). Variabilele `c` și `d` sunt variabile locale în funcția `func1` (vizibile doar în `func1`). Variabila `y` este variabilă externă și este vizibilă din punctul în care a fost definită, până la sfârșitul fișierului sursă (în acest caz, în funcția `func1`).

Exemplu:

```
int x;
void main()
{int a,b;
//-----
}
int y;
void func1(void)
{int c,d;
//-----
}
```

Clase de memorare. O variabilă se caracterizează prin: nume, tip, valoare și clasă de memorare.

Clasa de memorare se specifică la declararea variabilei, prin unul din următoarele cuvinte cheie:

- **auto;**
- **register;**
- **extern;**
- **static.**

Clasa de memorare determină timpul de viață și domeniul de vizibilitate (scopul) unei variabile (tabelul 1).

Exemplu:

```
auto int a;
static int x;
extern double y;
register char c;
```

- **Clasa de memorare auto.** Dacă o *variabilă locală* este declarată fără a se indica în mod explicit o clasă de memorare, clasa de memorare considerată implicit este `auto`. Pentru acea variabilă se alocă memorie automat, la intrarea în blocul sau în funcția în care ea este declarată. Deci domeniul de vizibilitate al variabilei este blocul sau funcția în care aceasta a fost definită. Timpul de viață este durata de execuție a blocului sau a funcției.
- **Clasa de memorare register.** Variabilele din clasa `register` au același domeniu de

vizibilitate și timp de viață ca și cele din clasa auto. Deosebirea față de variabilele din clasa auto constă în faptul că pentru memorarea variabilelor register, compilatorul utilizează regiștrii interni (ceea ce conduce la creșterea eficienței). Unei variabile pentru care se specifică drept clasă de memorare register, *nu i se poate aplica operatorul de referențiere*.

- **Clasa de memorare extern.** O *variabilă globală* declarată fără specificarea unei clase de memorare, este considerată ca având clasa de memorare extern. Domeniul de vizibilitate este din momentul declarării până la sfârșitul fișierului sursă. Timpul de viață este durata execuției fișierului. O variabilă din clasa extern este inițializată automat cu valoarea 0.
- **Clasa de memorare static.** Clasa de memorare static are două utilizări distincte:
 - Variabilele *locale statice* au ca domeniu de vizibilitate blocul sau funcția în care sunt definite, iar ca timp de viață - durata de execuție a programului. Se inițializează automat cu 0.
 - Variabilele *globale statice* au ca domeniu de vizibilitate punctul în care au fost definite până la sfârșitul fișierului sursă, iar ca timp de viață - durata execuției programului.

Tabelul 1.

Clasa de memorare	Variabila	Domeniu vizibilitate	Timp de viață
auto (register)	locală (internă)	Blocul sau funcția	Durara de execuție a blocului sau a funcției
extern	globală	Din punctul definirii, până la sfârșitul fișierului (ROF) Alte fișiere	Durara de execuție a blocului sau a programului
static	globală	ROF	“-”-
	locală	Bloc sau funcție	“-”-
nespecificat	globală	Vezi extern	Vezi extern
	locală	Vezi auto	Vezi auto

9. Moduri de alocare a memoriei

Alocarea memoriei se poate realiza în următoarele moduri:

- ***alocare statică;***
- ***alocare dinamică;***
- ***alocare pe stivă.***
- *Se alocă static memorie* în următoarele cazuri:
 - pentru instrucțiunile de control propriu-zise;
 - pentru variabilele globale și variabilele locale declarate în mod explicit static.
- *Se alocă memorie pe stivă* pentru variabilele locale.
- *Se alocă dinamic memorie* în mod explicit, cu ajutorul funcțiilor de alocare dinamica, aflate în headerul **<alloc.h>**.

Exemplu:

```
int a,b; double x;
double f1(int c, double v)
{int b;
static double z;
}
```

```
double w;
int fl(int w)
{double a;
}
void main()
{double b, c; int k;
b=fl(k,c);
}
```

📖 Alocarea memoriei în mod dinamic. Pentru toate tipurile de date (simple sau structurate), la declararea acestora, compilatorul alocă automat un număr de locații de memorie (corespunzător tipului datei). Dimensiunea zonei de memorie necesară pentru păstrarea valorilor datelor este fixată înaintea lansării în execuție a programului. În cazul declarării unui tablou de întregi cu maximum 100 de elemente vor fi alocați `100*sizeof(int)` locații de memorie succesive. În situația în care la un moment dat tabloul are doar 20 de elemente, pentru a aloca doar atâta memorie cât este necesară în momentul respectiv, se va aloca memorie în mod dinamic.

Este de dorit ca în cazul datelor a căror dimensiune nu este cunoscută a priori sau variază în limite largi, să se utilizeze o altă abordare: alocarea memoriei în mod dinamic. În mod dinamic, memoria nu mai este alocată în momentul compilării, ci **în momentul execuției**. Alocarea dinamică elimină necesitatea definirii complete a tuturor cerințelor de memorie în momentul compilării. În *limbajul C*, alocarea memoriei în mod dinamic se face cu ajutorul funcțiilor **malloc**, **calloc**, **realloc**; eliberarea zonei de memorie se face cu ajutorul funcției **free**. Funcțiile de alocare/dezallocare a memoriei au prototipurile în header-ele **<stdlib.h>** și **<alloc.h>**:

```
void *malloc(size_t nr_octei_de_alocat);
```

Funcția **malloc** necesită un singur argument (numărul de octeți care vor fi alocați) și returnează un pointer generic către zona de memorie alocată (pointerul conține adresa primului octet al zonei de memorie rezervate).

```
void *calloc(size_t nr_elemente, size_t mărimea_în_octeți_
a_unui_elem);
```

Funcția **calloc** lucrează în mod similar cu **malloc**; alocă memorie pentru un tablou de `nr_elemente`, numărul de octeți pe care este memorat un element este `mărimea_în_octeți_a_unui_elem` și returnează un pointer către zona de memorie alocată.

```
void *realloc(void *ptr, size_t mărime);
```

Funcția **realloc** permite modificarea zonei de memorie alocată dinamic cu ajutorul funcțiilor **malloc** sau **calloc**.

Observație: În cazul în care nu se reușește alocarea dinamică a memoriei (memorie insuficientă), funcțiile **malloc**, **calloc** și **realloc** returnează un pointer null. Deoarece funcțiile **malloc**, **calloc**, **realloc** returnează un pointer generic, rezultatul poate fi atribuit oricărui tip de pointer. La atribuire, este indicat să se utilizeze operatorul de conversie explicită (vezi exemplu).

Eliberarea memoriei (alocate dinamic cu una dintre funcțiile **malloc**, **calloc** sau **realloc**) se realizează cu ajutorul funcției **free**.

```
void free(void *ptr);
```

Exemplu: Să se aloce dinamic memorie pentru 20 de valori întregi.

```
int *p;
p=(int*)malloc(20*sizeof(int));
```

```
//p=(int*)calloc(20, sizeof(int));
```

Exercițiu: Să se scrie un program care implementează funcția numită `introd_val`. Funcția trebuie să permită introducerea unui număr de valori reale, pentru care se alocă memorie dinamic. Valorile citite cu ajutorul funcției `introd_val` sunt prelucrate în funcția `main`, apoi memoria este eliberată.

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
float *introd_val()
/* pentru a putea realiza eliberarea memoriei în funcția main, funcția introd_val trebuie să returneze
adresa de început a zonei de memorie alocate dinamic */
{double *p; int nr;printf("Număr valori:"); scanf("%d", nr);
if (!(p=(float*)malloc(nr*sizeof(float))) ){
    printf("Memorie insuficientă!\n");return NULL;
}
for (int i=0; i<nr; i++){
    printf("Val %d=", i+1); scanf("%lf", p+i); return p;}
}
void main()
{float *pt; pt=introd_val();
// prelucrare tablou
free(pt);
}
```

Exercițiu: Să se scrie un program care citește numele angajaților unei întreprinderi. Numărul angajaților este transmis ca argument către funcția `main`. Alocarea memoriei pentru cei `nr_ang` angajați, cât și pentru numele fiecăruia dintre aceștia se va face în mod dinamic.

```
#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>
void main(int argc, char *argv[])
{char **ang_ptr;
char *nume;
int nr_ang, i;
if (argc==2){
    nr_ang=atoi(argv[1]);/* numărul angajaților este transmis ca argument către funcția
main. El este convertit din șir de caractere în număr */
    ang_ptr=(char**)calloc(nr_ang, sizeof(char*));
    if ((ang_ptr==0)){
        printf("Memorie insuficientă!\n");exit(1);}
    nume=(char*)calloc(30, sizeof(char));
    for (i=0; i<nr_ang; ++i){
        printf("Nume angajat:");
        scanf("%s",nume);
        ang_ptr[i]=(char*)calloc(strlen(nume)+1, sizeof(char));
        strcpy(ang_ptr[i], nume);
    }
    free(nume);
    printf("\n");
    for (i=0; i<nr_ang; i++)
        printf("Angajat nr %d: %s\n", i+1, ang_ptr[i]);
}
else
    printf("Lansare în execuție: %s număr_de_angajați\n", argv[0]);
```



```
}
```

În limbajul C++ alocarea dinamică a memoriei și eliberarea ei se pot realiza cu operatorii **new** și **delete**. Folosirea acestor operatori reprezintă o metodă superioară, adaptată programării orientate obiect.

Operatorul **new** este un operator unar care returnează un pointer la zona de memorie alocată dinamic. În situația în care nu există suficientă memorie și alocarea nu reușește, operatorul **new** returnează pointerul **NULL**. Operatorul **delete** eliberează zona de memorie spre care pointează argumentul său.

Sintaxa:

```
tipdata_pointer = new tipdata;
tipdata_pointer = new tipdata(val_inițializare);
//pentru inițializarea datei pentru care se alocă memorie dinamic
tipdata_pointer = new tipdata[nr_elem]; //alocarea memoriei pentru un tablou

delete tipdata_pointer;
delete [nr_elem] tipdata_pointer;           //eliberarea memoriei pentru tablouri
```

Tipdata reprezintă tipul datei (predefinit sau obiect) pentru care se alocă dinamic memorie, iar **tipdata_pointer** este o variabilă pointer către tipul **tipdata**.

Pentru a putea afla memoria RAM disponibilă la un moment dat, se poate utiliza funcția **coreleft**:

unsigned coreleft(void);

Exercițiu: Să se alocă dinamic memorie pentru o dată de tip întreg:

```
int *pint;
pint=new int;
//Sau:
int &i=*new int;
i=100;           //i permite referirea la întregul păstrat în zona de memorie alocată dinamic
```

Exercițiu: Să se alocă dinamic memorie pentru o dată reală, dublă precizie, inițializând-o cu valoarea -7.2.

```
double *p;
p=new double(-7.2);
//Sau:
double &p=* new double(-7.2);
```

Exercițiu: Să se alocă dinamic memorie pentru un vector de m elemente reale.

double *vector; vector=new double[m];

Exemplu: Să se urmărească rezultatele execuției următorului program, care utilizează funcția **coreleft**.

```
#include <iostream.h>
#include <alloc.h>
#include <conio.h>
void main()
{ int *a,*b; clrscr();
cout<<"Mem. libera inainte de alocare:"<<coreleft()<<"\n";
cout<<"Adr. pointerilor a si b:"<<&a<<"    "<<&b<<"\n";
cout<<"Valorile pointeri a si b inainte de alocare:"<<a<<"
"<<b<<"\n";
a=new int;  b=new int[10];
cout<<"Mem. libera dupa alocare:"<<coreleft()<<"\n";
cout<<"Valorile pointerilor a si b dupa alocare:"<<a<<"    "<<b<<"\n";
```

```

cout<<"Continutul memoriei alocate:\n"<<"*a="<<*a<<"\n*b="<<*b<<"\n";
for (int k=0;k<10;k++) cout<<"\nb[ "<<k<<" ]="<<b[k]; cout<<"\n";
getch();
*a=1732;
for (int u=0;u<10;u++) b[u]=2*u+1;
cout<<"Cont. zonelor alocate dupa atribuire:"<<"\n*a="<<*a<<"\nb=";
for (u=0;u<10;u++) cout<<"\nb[ "<<u<<" ]="<<b[u];
delete a; delete b;
cout<<"Mem. libera dupa eliberare:"<<coreleft()<<"\n";
cout<<"Valorile pointerilor a si b dupa eliberare:"<<a<<"
"<<b<<"\n";
cout<<"Continutul                                     memoriei
eliberate:\n"<<"*a="<<*a<<"\n*b="<<*b<<"\n";
for      (k=0;k<10;k++)      cout<<"\nb[ "<<k<<" ]="<<b[k];      cout<<"\n";
cout<<b[3];
getch();
}

```

10. Funcții recursive

O funcție este numită **funcție recursivă** dacă ea se autoapelează, fie *direct* (în definiția ei se face apel la ea însăși), fie *indirect* (prin apelul altor funcții). Limbajele C/C++ dispun de mecanisme speciale care permit suspendarea execuției unei funcții, salvarea datelor și reactivarea execuției la momentul potrivit. Pentru fiecare apel al funcției, parametri și variabilele automate se memorează pe stivă, având valori distincte. Variabilele statice ocupă tot timpul aceeași zonă de memorie (figurează într-un singur exemplar) și își păstrează valoarea de la un apel la altul. Orice apel al unei funcții conduce la o revenire în funcția respectivă, în punctul următor instrucțiunii de apel. La revenirea dintr-o funcție, stiva este curățată (stiva revine la starea dinaintea apelului).

Un exemplu de funcție recursivă este funcția de calcul a factorialului, definită astfel:

```

fact(n)=1, dacă n=0;
fact(n)=n*fact(n-1), dacă n>0;

```

Exemplu: Să se implementeze recursiv funcția care calculează $n!$, unde n este introdus de la tastatură:

```
#include <iostream.h>
int fact(int n)
{if (n<0){
    cout<<"Argument negativ!\n";
    exit(2);
}
else if (n==0)    return 1;
else             return n*fact(n-1);
}
void main()
{int nr, f; cout<<"nr="; cin>>nr;
f=fact(nr); cout<<nr<<"!="<<f<<"\n";
}
```

Se observă că în corpul funcției `fact` se apelează însăși funcția `fact`. Presupunem că $nr=4$ (inițial, funcția `fact` este apelată pentru a calcula $4!$). Să urmărim diagramele din figurile 7 și 8. La apelul funcției `fact`, valoarea parametrului de apel `nr` ($nr=4$) inițializează parametrul formal `n`. Pe *stivă* se memorează adresa de revenire în funcția apelantă (`adr1`) și valoarea lui `n` ($n=4$) (figura 7.a.). Deoarece $n>0$, se execută instrucțiunea de pe ramura `else` (`return n*fact(n-1)`). Funcția `fact` se autoapelează direct. Se memorează pe *stivă* noua adresă de revenire și noua valoare a parametrului `n` ($n=3$) (figura 7.b.).

La noul reapel al funcției `fact`, se execută din nou instrucțiunea de pe ramura `else` (`return n*fact(n-1)`). Se memorează pe *stivă* adresa de revenire și noua valoare a parametrului `n` ($n=2$) (figura 7.c.). La noul reapel al funcției `fact`, se execută din nou instrucțiunea de pe ramura `else` (`return n*fact(n-1)`). Se memorează pe *stivă* adresa de revenire și noua valoare a parametrului `n` ($n=1$) (figura 7.d.). La noul reapel al funcției `fact`, se execută din nou instrucțiunea de pe ramura `else` (`return n*fact(n-1)`). Se memorează pe *stivă* adresa de revenire și noua valoare a parametrului `n` ($n=0$) (figura 7.e.).

În acest moment $n=0$ și se revine din funcție cu valoarea 1 ($1*fact(0)=1*1$), la configurația stivei din figura 7.d) (se curăță stiva și se ajunge la configurația din figura 7.d). În acest moment $n=1$ și se revine cu valoarea $2*fact(1)=2*1=2$, se curăță stiva și se ajunge la configurația stivei din figura 7.c). În acest moment $n=2$ și se revine cu valoarea $3*fact(2)=3*2=6$, se curăță stiva și se ajunge la configurația stivei din figura 7.b). Se curăță stiva și se ajunge la configurația stivei din figura 7.a). În acest moment $n=3$ și se revine cu valoarea $4*fact(3)=4*6=24$.

O funcție recursivă poate fi realizată și *iterativ*. Modul de implementare trebuie ales în funcție de problemă. Deși implementarea recursivă a unui algoritm permite o descriere clară și compactă, recursivitatea nu conduce la economie de memorie și nici la execuția mai rapidă a programelor. În general, se recomandă utilizarea funcțiilor recursive în anumite tehnici de programare, cum ar fi unele metode de căutare (*backtracking*).

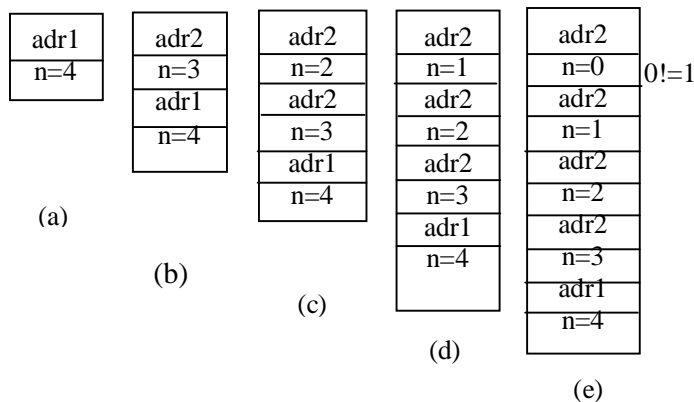


Figura 7. Configurația stivei.

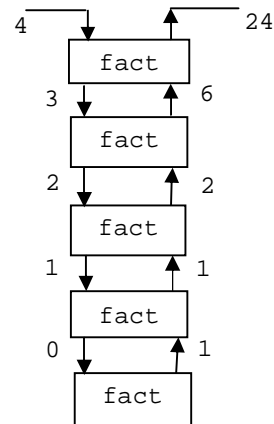


Figura 8. Parametri funcției fact.

Exercițiu: Fie șirul lui Fibonacci, definit astfel: $f(0)=0$, $f(1)=1$, $f(n)=f(n-1)+f(n-2)$, dacă $n>1$. Să se scrie un program care implementează algoritmul de calcul al șirului Fibonacci atât recursiv, cât și iterativ. Să se compare timpul de execuție în cele două situații.

```
#include <iostream.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>
#include <stdio.h>
```

```
long int iterativ_fib(long int n) //varianta de implementare iterativă
{
    if (n==0) return 0;
    if (n==1) return 1;
    int i; long int a, b, c; a=0; b=1;
    for (i=2; i<=n; i++){
        c=b; b+=a; a=c;
    }
    return b;
}
```

```
long int recursiv_fib(long int n) //varianta de implementare recursivă
{
    if (n==0) return 0;
    if (n==1) return 1;
    long int i1=recursiv_fib(n-1);
    long int i2=recursiv_fib(n-2);
    return i1+i2;
}
```

```
void main()
{
    int n; clrscr();
    cout<<MAXLONG<<'\\n';
    for (n=10; n<=40; n++) {
        clock_t t1, t2, t3;
        cout<<CLK_TCK<<'\\n';
        t1=clock(); long int f1=iterativ_fib(n);
        t2=clock(); long int f2=recursiv_fib(n); t3=clock();
        double timp1=(double)(t2-t1)/CLK_TCK;
        double timp2=(double)(t3-t2)/CLK_TCK;
    }
}
```

```
printf("ITERATIV: %10ld t=%20.10lf\n",f1,timp1);
printf("RECURSIV: %10ld t=%20.10lf\n",f2,timp2);
cout<<"Apasa o tasta...\n"; getch();
} }
```

În exemplul anterior, pentru măsurarea timpului de execuție s-a utilizat funcția `clock`, al cărei prototip se află în header-ul `time.h`. Variabilele `t1`, `t2` și `t3` sunt de tipul `clock_t`, tip definit în același header. Constanta simbolică `CLK_TCK` definește numărul de bătăi ale ceasului, pe secundă.

În general, orice algoritm care poate fi implementat iterativ, poate fi implementat și recursiv. Timpul de execuție a unei recursii este semnificativ mai mare decât cel necesar execuției iterației echivalente.

Exercițiu: Să se implementeze și să se testeze un program care:

- Generează aleator și afișează elementele unui vector;
 - Sortează aceste elemente, crescător, aplicând metodele de sortare `BubbleSort`, `InsertSort`, și `QuickSort`;
 - Să se compare viteza de sortare pentru vectori de diverse dimensiuni (10,30,50,100 elemente).
- Metoda **BubbleSort** a fost prezentată în **capitolul 4**.
 - Metoda **QuickSort** reprezintă o altă metodă de sortare a elementelor unui vector. Algoritmul este recursiv: se împarte vectorul în două partiții, față de un element pivot (de obicei, elementul din "mijlocul vectorului"). Partiția stângă începe de la indexul i (la primul apel $i=0$), iar partiția dreaptă se termină cu indexul j (la primul apel $j=n-1$) (figura 9).

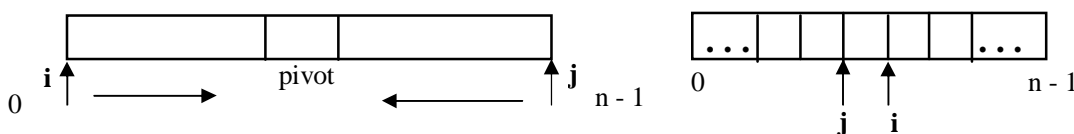


Figura 9. Sortare prin metoda QuickSort.

Partiția stângă este extinsă la dreapta (i incrementat) până când se găsește un element mai mare decât pivotul; partiția dreaptă este extinsă la stânga (j decrementat) până când se găsește un element mai mic decât pivotul. Cele două elemente găsite, `vect[i]` și `vect[j]`, sunt interschimbate.

Se reia ciclic extinderea partițiilor până când i și j se "încrucișează" (i devine mai mare ca j). În final, partiția stângă va conține elementele mai mici decât pivotul, iar partiția dreaptă - elementele mai mari decât pivotul, dar nesortate.

Algoritmul este reluat prin recursie pentru partiția stângă (cu limitele între 0 și j), apoi pentru partiția dreaptă (cu limitele între i și $n-1$). Recursia pentru partea stângă se oprește atunci când j atinge limita stângă (devine 0), iar recursia pentru partiția dreaptă se oprește când i atinge limita dreaptă (devine $n-1$).

```

SUBALGORITM QuickSort (vect[ ], stg, drt)    //la primul apel stg = 0 si drt = n - 1
ÎNCEPUT SUBALGORITM
i ← stg
j ← drt
DACĂ i < j ATUNCI
ÎNCEPUT
    pivot=vect[(stg+drt)/2]
    CÂT TIMP i <= j REPETĂ
        //extinderea partițiilor stânga și dreapta până când i se încrucișează cu j
    ÎNCEPUT
        CÂT TIMP i<drt si vect[i]<pivot REPETĂ
            i = i + 1
        CÂT TIMP j<stg si vect[j]>pivot REPETĂ
            j = j - 1
        DACĂ i<=j ATUNCI
            ÎNCEPUT                //interschimbă elementele vect[i] și vect[j]
                aux←vect[i]
                vect[i]←vect[j]
                vect[j]←aux
                i←i+1
                j←j-1
            SFÂRȘIT
        SFÂRȘIT
    DACĂ j > stg ATUNCI // partiția stângă s-a extins la maxim, apel qiuckSort pentru ea
        CHEAMĂ QuickSort(vect, stg, j)
    DACĂ i < drt ATUNCI // partiția dreaptă s-a extins la maxim, apel qiuckSort pentru ea
        CHEAMĂ QuickSort(vect, i, drt)
SFÂRȘIT
SFÂRȘIT SUBALGORITM

```

- Metoda **InsertSort** (metoda inserției). Metoda identifică cel mai mic element al vectorului și îl schimbă cu primul element. Se reia procedura pentru vectorul inițial, fără primul element și se caută minimumul în acest nou vector, etc.

```

SUBALGORITM InsertSort (vect[ ], nr_elem)
ÎNCEPUT SUBALGORITM
    CÂT TIMP i< nr_elem REPETĂ
        ÎNCEPUT
            pozMin←cautMinim(vect, i)    // se apelează algoritmul cautMinim
            aux←vect[i]
            vect[i]←vect[pozMin]
            vect[pozMin]←aux
            i←i+1
        SFÂRȘIT
    SFÂRȘIT SUBALGORITM

```

Funcția **cautMin(vect[], indexIni, nr_elem)** caută elementul minim al unui vector, începând de la poziția **indexIni** și returnează poziția minimumului găsit.

Mod de implementare (Se va completa programul cu instrucțiunile care obțin și afișează timpului necesar ordonării prin fiecare metodă. Se vor compara rezultatele pentru un vector de 10, 30, 50, 100 elemente):

```
#include <stdlib.h>
```

```

#include <stdio.h>
#include <time.h>
#define TRUE      1
#define FALSE     0
void gener(double v[], int n)
//functia de generare aleatoare a elementelor vectorului v, cu n elemente
{for (int i=0; i<n; i++)
    v[i]=1.0*rand()/100000;
}
void afis(double v[], int n)
//functia de afisare a vectorului
{for (int i=0; i<n; i++)
    printf("%10.2f",v[i]);
printf("\n");
}
void copie_vect(double v1[], double v[], int n)
//functie de "duplicare" a unui vector; copie vectorul v in vectorul v1
{for (int i=0; i<n; i++)
    v1[i]=v[i];
}
void bubbleSort(double v[], int n)
{int gata; gata=FALSE;
while (!gata){
    gata=TRUE;
    for (int i=0; i<n-1; i++)
        if (v[i]>=v[i+1]){
            double aux=v[i];
            v[i]=v[i+1];
            v[i+1]=aux;
//            printf("Interschimbare element %d cu %d",i,i+1);
//            afis(v,n);
            gata=FALSE;}
}
}
int cautMin(double v[], int indexIni, int n)
// cauta elementul minim, incepând de la pozitia indexIni, inclusiv
{ double min=v[indexIni];
int pozMin=indexIni;
for (int i=indexIni; i<n; i++)
    if (v[i]<=min){
        min=v[i]; pozMin=i;
    }
return pozMin;
}
void insertSort(double v[], int n)
{ int i;
for (i=0; i<n; i++){
    int poz=cautMin(v, i, n);
    double aux=v[i];
    v[i]=v[poz];
    v[poz]=aux;
}
}
void quickSort(double v[], int stg, int drt)
{int i,j; i=stg; j=drt; double pivot, aux;
if (i<j){
    pivot=v[(stg+drt)/2];
    while (i<=j){ //extindere partitie st si dr pana i se incruca cu j
        while (i<drt && v[i]<pivot) i++;

```

```

        while (j>stg && v[j]>pivot)    j--;
        if (i<=j){
            aux=v[i];v[i]=v[j];v[j]=aux; //interschimbare elemente
            i++; j--;
        }
    }
    if (j>stg)    quickSort(v, stg, j);
    if (i<drt)    quickSort(v, i, drt);
}
}
void main()
{
    clock_t ti,tf; int n;    //n=nr elemente vector
    printf("Nr componente vector:"); scanf("%d", &n);
    double v[200], v1[200], v2[200], v3[200];
    gener(v, n);
    copie_vect(v1,v,n);
    printf("\nInainte de ordonare: v1="); afis(v1, n); ti=clock();
    bubbleSort(v1,n); tf=clock(); printf("\nDupa ordonare :
    v1=");afis(v1, n);
    printf("%10.7f", dif_b);
    printf("\n\n***** INSERT SORT *****\n");
    copie_vect(v2,v,n);
    printf("\nInainte de ordonare INSERT: v2="); afis(v2, n);
    insertSort(v2,n); printf("\nDupa ordonare          INSERT: v2=");afis(v2,
    n);
    int st=0; int dr=n-1; copie_vect(v3, v, n);
    printf("\n\n***** QUICK SORT *****\n");
    printf("\nInainte ordonare QUICK: v3="); afis(v3, n);
    quickSort(v3, st, dr); printf("\nDupa ordonare      QUICK: v3=");
    afis(v3, n);
}

```

11. Pointeri către funcții

Așa cum s-a evidențiat în precedent, există trei categorii de variabilele pointer:

- Pointeri cu tip;
- Pointeri generici (void);
- Pointeri *către funcții*.

Pointerii către funcții sunt variabile pointer care conțin adresa de început a codului executabil al unei funcții. Pointerii către funcții permit:

- Transferul ca parametru al adresei unei funcții;
- Apelul funcției cu ajutorul pointerului.

Declarația unui pointer către funcție are următoarea formă:

```
tip_val_intoarse (*nume_point)(lista_declar_param_formali);
```

unde: `nume_point` este un pointer de tipul “funcție cu rezultatul tipul valorii întoarse”. În declarația anterioară trebuie remarcat rolul parantezelor, pentru a putea face distincție între declarația unei funcții care întoarce un pointer și declarația unui pointer de funcție:

```
tip_val_intoarse * nume_point (lista_declar_param_formali);
tip_val_intoarse (* nume_point)(lista_declar_param_formali);
```

Exemplu:

```
int f(double u, int v);           //prototipul funcției f
int (*pf)(double, int);          //pointer către funcția f
int i, j; double d;
```



```
pf=f; //atribuie adresa codului executabil al funcției f pointerului pf
j=*pf(d, i); //apelul funcției f, folosind pf
```

Exercițiu: Să se implementeze un program care calculează o funcție a cărei valoare este integrala altei funcții. Pentru calculul integralei se va folosi metoda trapezelor.

Relația pentru calculul integralei prin metoda

trapezelor pentru $\int_a^b f(x)dx$ este:

$$I = (f(a)+f(b))/2 + \sum_{k=1}^{n-1} f(a+k \cdot h)$$

Să se calculeze $\int_a^b \frac{\sqrt{0.2 + e^{\frac{|x|}{2}}}}{1 + \sqrt{0.3 + \ln(1 + x^4)}} dx$,

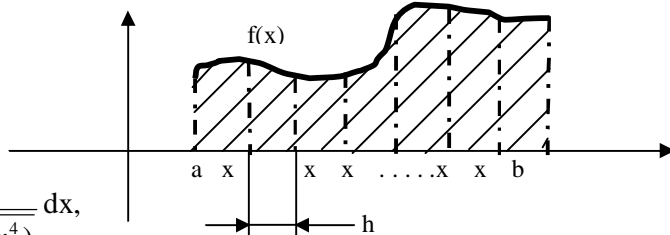


Figura 10. Calculul integralei prin metoda trapezelor.

cu o eroare mai mică decât eps (valoarea erorii introdusă de la tastatură).

```
#include <conio.h>
#include <math.h>
#include <iostream.h>
double functie(double x)
{return sqrt(0.1+exp(0.5*fabs(x)))/(1+sqrt(0.3+log(1+pow(x,4))))};
```

```
double intrap(double a, double b, long int n, double (*f)(double))
{double h,s=0; long k;
if (a>=b) return 0;
if (n<=0) n=1;
h=(b-a)/n;
for (k=1; k<n; k++) s+=f(a+k*h);
return ((f(a)+f(b))/2+s)*h;
}
```

```
void main()
{long int j; double p,q; double eps, d2;double dif;
cout<<"Marg. inf:"<<cin>>p; cout<<"Marg. sup:"<<cin>>q;
cout<<"Eroare:"<<cin>>eps; j=1;
double d1=intrap(p, q, j, functie);
do{
j*=2;
if (j>MAXLONG || j<0) break;
d2=intrap(p, q, j, functie);
dif=fabs(d1-d2); d1=d2;
cout<<"Nr.intervale "<<j<<" Val.integralei "<<d2<<"\n";
}while (dif>eps);
cout<<"\n\n-----\n";
cout<<"Val. integralei: "<<d2<<" cu eroare de:"<<eps<<"\n";
}
```

Întrebări și exerciții

Întrebări teoretice

1. Asemănări între transferul parametrilor unei funcții prin pointeri și prin referință.
2. Caracteristicile modului de transfer a parametrilor unei funcții prin pointeri.
3. Caracteristicile variabilelor globale.
4. Caracteristicile variabilelor locale.
5. Care este diferența între antetul unei funcții și prototipul acesteia?
6. Care sunt modurile de alocare a memoriei?
7. Care sunt modurile de transfer a parametrilor unei funcții?
8. Care sunt operatorii din C++ care permit alocarea/dezallocarea dinamică a memoriei?
9. Ce clase de memorare cunoașteți?
10. Ce este domeniul de vizibilitate a unei variabile?
11. Ce este prototipul unei funcții?
12. Ce este timpul de viață a unei variabile?
13. Ce loc ocupă declarațiile variabilelor locale în corpul unei funcții?
14. Ce reprezintă antetul unei funcții?
15. Ce rol are declararea funcțiilor?
16. Ce se indică în specificatorul de format al funcției printf ?
17. Ce sunt funcțiile cu număr variabil de parametri? Exemple.
18. Ce sunt funcțiile cu parametri implicați?
19. Ce sunt pointerii către funcții?
20. Ce sunt variabilele referință?
21. Cine determină timpul de viață și domeniul de vizibilitate ale unei variabile?
22. Comparatie între declararea și definirea funcțiilor.
23. Diferențe între modurile de transfer a parametrilor prin valoare și prin referință.
24. Diferențe între modurile de transfer a parametrilor unei funcții prin pointeri și prin referință.
25. Din apelul funcției printf se poate omite specificatorul de format?
26. Din ce este formată o funcție?
27. În ce zonă de memorie se rezervă spațiu pentru variabilele globale?
28. O funcție poate fi declarată în corpul altei funcții?
29. O funcție poate fi definită în corpul unei alte funcții?
30. Parametri formali ai unei funcții sunt variabile locale sau globale?
31. Transferul parametrilor prin valoare.
32. Ce rol au parametri formali ai unei funcții?

Exerciții practice

1. Să se implementeze programele cu exemplele prezentate.
2. Să se scrie programele pentru exercițiile rezolvate care au fost prezentate.
3. Să se modularizeze programele din **capitolul 4** (3.a.-3.g., 4.a.-4.i, 5.a.-5.h.), prin implementarea unor funcții (funcții pentru: citirea elementelor unui vector, afișarea vectorului, calculul sumei a doi vectori, calculul produsului scalar a doi vectori, aflarea elementului minim din vector, citire a unei matrici, afișare a matricii, calculul transpusei unei matrici, calculul sumei a două matrici, calculul produsului a două matrici, calculul produsului elementelor din triunghiul hașurat, etc.).
4. Să se rescrie programele care rezolvă exercițiile din **capitolul 3**, folosind funcții (pentru calculul factorialului, aflarea celui mai mare divizor comun, ordonarea

lexicografică a caracterelor, etc). Utilizați funcțiile de intrare/ieșire `printf` și `scanf`.

5. Să se scrie un program care citește câte două numere, până la întâlnirea perechii de numere 0, 0 și afișează, de fiecare dată, cel mai mare divizor comun al acestora, folosind o funcție care îl calculează.
6. Se introduce de la tastatura un număr întreg. Să se afișeze toți divizorii numărului introdus. Se va folosi o funcție de calcul a celui mai mare divizor comun a 2 numere.
7. Secvențele următoare sunt corecte din punct de vedere sintactic? Dacă nu, identificați sursele erorilor.


```

❑ void a(int x, y) {cout<<"x="<<x<<" y="<<y<<"\n"; }
void main( ) { int b=9; a(6, 7); }
❑ void main( ) { int x=8; double y=f(x); cout<<"y="<<y<<"\n"; }
int f(int z) {return z+z*z;}

```
8. Scrieți o funcție `găsește_cifra` care returnează valoarea cifrei aflate pe poziția `k` în cadrul numărului `n`, începând de la dreapta (`n` și `k` vor fi argumentele funcției).
9. Implementați propriile versiuni ale funcțiilor de lucru cu șiruri de caractere (din **paragraful 4.4**).
10. Să se calculeze valoarea funcției g , cu o eroare EPS (a , b , EPS citite de la tastatură):

$$g(x) = \int_a^b \sqrt{(x^2 + x + 1)} * \ln|x + a| dx + \int_a^b x * \arctg(b/(b + x)) dx$$

11. Implementați funcții iterative și recursive pentru calculul valorilor polinoamelor Hermite $H_n(y)$, știind că: $H_0(y) = 1$, $H_1(y) = 2y$, $H_n(x) = 2yH_{n-1}(y) - 2H_{n-2}(y)$ dacă $n > 1$. Comparați timpul de execuție al celor două funcții.
12. Să se scrie un program care generează toate numerele palindrom, mai mici decât o valoare dată, LIM. Un număr palindrom are cifrele simetrice egale (prima cu ultima, a doua cu penultima, etc). Se va folosi o funcție care testează dacă un număr este palindrom.
13. Fie matricea C ($N \times N$), $N \leq 10$, ale cărei elemente sunt date de relația:

$$C_{i,j} = \begin{cases} j! + \sum_{k=0}^j \sin(kx), & \text{dacă } i < j \\ x^i, & \text{dacă } i = j \\ i! + i \sum_{k=0}^i \cos(kx), & \text{dacă } i > j \end{cases}, \text{ unde } x \in [0, 1], x \text{ introdus de la tastatură}$$

- a) Să se implementeze următoarele funcții: de calcul a elementelor matricii; de afișare a matricii; de calcul și de afișare a procentului elementelor negative de pe coloanele impare (de indice 1, 3, etc).
- b) Să se calculeze și să se afișeze matricea B , unde: $B = C - C^2 + C^3 - C^4 + C^5$.
14. Să se creeze o bibliotecă de funcții pentru lucrul cu matrici, care să conțină funcțiile utilizate frecvent (citirea elementelor, afisarea matricii, adunare a două matrici, etc). Să se folosească această bibliotecă.
15. Să se creeze o bibliotecă de funcții pentru lucrul cu vectori, care să conțină funcțiile utilizate frecvent. Să se folosească această bibliotecă.

Sumar:

- | | |
|----------------------------------|-----------------------|
| 1. Tipuri definite de utilizator | 5. Declarații typedef |
| 2. Structuri | 4. Uniuni |
| 3. Câmpuri de biți | 6. Enumerări |
-

1. Tipuri definite de utilizator

Limbajele de programare de nivel înalt oferă utilizatorului facilități de a prelucra atât datele singulare (izolate), cât și pe cele grupate. Un exemplu de grupare a datelor - de același tip - îl constituie tablourile. Datele predefinite și tablourile (prezentate în capitolele anterioare) nu sunt însă suficiente. Informația prelucrată în programe este organizată, în general în ansambluri de date, de diferite tipuri. Pentru a putea descrie aceste ansambluri (structuri) de date, limbajele de programare de nivel înalt permit programatorului să-și definească *propriile tipuri de date*.

Limbajul C oferă posibilități de definire a unor tipurilor de date, cu ajutorul:

- ❑ **structurilor** - permit gruparea unor obiecte (date) de tipuri diferite, referite printr-un nume comun;
- ❑ **câmpurilor de biți** - membri ai unei structuri pentru care se alocă un grup de biți, în interiorul unui cuvânt de memorie;
- ❑ **uniunilor** - permit utilizarea în comun a unei zone de memorie de către mai multe obiecte de diferite tipuri;
- ❑ declarațiilor **typedef** - asociază nume tipurilor noi de date;
- ❑ **enumerărilor** - sunt liste de identificatori cu valori constante, întregi.

2. Structuri

Structurile grupează date de tipuri diferite, constituind definiții ale unor noi tipuri de date. Componentele unei structuri se numesc **membrii (câmpurile)** structurii. La declararea unei structuri se pot preciza tipurile, identificatorii elementelor componente și numele structurii.

Forma generală de declarare a unei structuri:

```
struct identificador_tip_structura
{
    lista_de_declaratii_membrii;
} lista_identificatori_variabile;
```

în care:

struct este un cuvânt cheie (obligatoriu)

identificador_tip_structura reprezintă numele noului tip (poate lipsi)

lista_de_declaratii_membri este o listă în care apar tipurile și identificatorii membrilor structurii

lista_identificatori_variabile este o listă cu identificatorii variabilelor de tipul declarat.

Membrii unei structuri pot fi de orice tip, cu excepția tipului structură care se declară. Se admit însă, pointeri către tipul structură.

Identificator_tip_structura poate lipsi din declarație, însă în acest caz, în lista_identificatori_variabile trebuie să fie prezent cel puțin un identificator_varabila.

Lista_identificatori_variabile poate lipsi, însă, în acest caz, este obligatorie prezența unui identificator_tip_structura.

Exemplu: Se definește noul tip de date numit data, cu membrii zi, luna, an. Identificatorii variabilelor de tipul data sunt data_nașterii, data_angajării.

```
struct data {
    int zi;
    char luna[11];
    int an;
} data_nașterii, data_angajării;
```

Declarația de mai sus poate apare sub forma:

```
struct data {
    int zi;
    char luna[11];
    int an;
};
struct data data_nașterii, data_angajării;
/*Variabilele data_nașterii și data_angajării sunt date de tipul data */
```

Se poate omite numele noului tip de date:

```
struct {
    int zi;
    char luna[11];
    int an;
} data_nașterii, data_angajării;
```

Inițializarea variabilelor de tip nou, definit prin structură, se poate realiza prin enumerarea valorilor membrilor, în ordinea în care aceștia apar în declarația structurii. Referirea unui membru al structurii se realizează cu ajutorul unui operator de bază, numit **operator de selecție**, simbolizat prin „.”. Operatorul are prioritate maximă. Membrul stâng al operatorului de selecție precizează numele variabilei de tipul introdus prin structură, iar membrul drept-numele membrului structurii, ca în exemplul următor:

Exemplu:

```
struct angajat{
    char nume[20], prenume[20];
    int nr_copii;
    double salariu;
    char loc_nastere[20];
};
struct angajat a1= {"Popescu", "Vlad", 2, 2900200, "Galati"};
a1.nr_copii = 3;
strcpy(a1.nume, "Popesco");
```

Variabilele de același tip pot apare ca operanzi ai operatorului de atribuire. În acest caz atribuirile se fac membru cu membru. În exemplul anterior am declarat și inițializat variabila a1, de tip angajat. Declarăm și variabila a2, de același tip. Dacă dorim ca membrii variabilei a2 să conțină aceleași valori ca membrii variabilei a1 (a1 și a2 de tip angajat), putem folosi operatorul de atribuire, ca în exemplul următor:

```
struct angajat a2;
a2=a1;
```

Așa cum s-a observat din exemplul anterior, structurile pot avea ca membri tablouri (structura angajat are ca membrii tablourile de caractere loc_naștere[20], nume[20], prenume[20]). De asemenea, variabilele de tip definit prin structură pot fi grupate în tablouri.

Exemplu:

```
struct persoana{
    char nume[20], prenume[20];
    int nr_copii;
    double salariu;
    char loc_nastere[20];
}angajati[100];

/* S-au declarat noul tip numit persoana și variabila numită angajati, care este un vector (cu maxim
100 de elemente), ale cărui elemente sunt de tipul persoana */
//Inițializarea elementelor vectorului angajați[100]
for (int i=0; i<100; i++){
    cout<<"Introduceți datele pentru angajatul "<<i+1<<'\n';
    cout<<"Numele :"; cin>>angajati[i].nume;
    cout<<"Prenumele :"; cin>>angajati[i].prenume;
    cout<<"Nr. copii:"; cin>> angajati[i].nr_copii;
    cout<<"Locul nașterii:"; cin>> angajati[i].loc_naștere;
}
}
```

Limbajul C permite definirea de structuri ale căror membri sunt tot structuri:

Exemplu:

```
struct data{
    int zi;
    char luna[11];
    int an;
};

struct persoana{
    char nume[20], prenume[20];
    int nr_copii;
    double salariu;
    char loc_naștere[20];
    struct data data_nașterii;
};
```

```
Struct persoana
p1={"Popescu", "Vasile", 1, 4000000, "Galati", {22, "Mai", 1978}};
//Modificarea membrului data_nașterii pentru variabila p1 de tip persoana:
p1.data_nașteri.zi=23;
strcpy(p1.data_nașteri.luna, "Februarie");
p1.data_nasteri.an=1980;
```

Dacă se dorește transmiterea ca parametri ai unor funcții a datelor de tip definit de utilizator prin structuri, acest lucru se realizează **numai** cu ajutorul pointerilor spre noul tip.

De exemplu, este necesar ca variabila p1, de tip persoana, să fie prelucrată în funcția f. În acest caz, funcția va primi ca parametru un pointer spre tipul persoana. Funcția va avea prototipul:

```
void f(struct persoana *q);
```

Apelul funcției se realizează astfel: f(&p1);

În corpul funcției f, accesul la membrii variabilei q, de tip persoana, se realizează astfel:

```

(*q).nume;
(*q).prenume;
(*q).data_nasterii.an;           , etc.

```

Pentru a simplifica construcțiile anterioare, se folosește **operatorul de selecție indirectă** (\rightarrow):

```

q->nume;
q->prenume;
q->data_nasterii.an           , etc.

```

Structurile sunt utilizate în mod frecvent la definirea unor tipuri de date recursive (în implementarea listelor, arborilor, etc.). Un tip de date este direct recursiv dacă are cel puțin un membru care este de tip pointer spre el însuși.

Exemplu:

```

struct nod{
    char nume[100];
    int an;
    struct nod *urmator;
};

```

Exercițiu: Să se citească informațiile despre angajații unei întreprinderi, folosind o funcție de citire. Să se afișeze apoi informațiile despre angajați.

```

#include <stdio.h>
#include <conio.h>
struct persoana{
    char nume[20];int varsta;int salariu;
};

void cit_pers(struct persoana *ptr_pers)
{printf("Nume angajat:"); scanf("%s",ptr_pers->nume);
printf("Varsta angajat:"); scanf("%d", &ptr_pers->varsta);
printf("Salariu angajat:"); scanf("%d", &ptr_pers->salariu);
}

void main()
{struct persoana *p;    //pointer catre date de tip persoana
int nr_ang; clrscr();
printf("Nr. angajati:");scanf("%d", &nr_ang);
p=new persoana[nr_ang]; //alocare dinamica a memoriei pentru cei nr_ang angajati
for (int i=0; i<nr_ang; i++)
    cit_pers(&p[i]);
printf("\n\n Datele despre angajati:\n\n");
for (i=0; i<nr_ang; i++){
    printf("Angajatul %d\n NUME:  %s\n VARSTA:  %d\n \ //continuare sir
        SALARIUL:  %.d\n", i+1,p[i].nume,p[i].varsta, p[i].salariu);
    printf("\n\n Apasa o tasta....\n"); getch();
}
}

```

Așa cum se observă din exemplu, funcția `cit_pers` primește ca parametru pointerul `ptr_pers`, către tipul `persoana`. Pentru a acesa membri structurii, în corpul funcției, se folosește operatorul de selecție indirectă (\rightarrow). În funcția `main`, se alocă memorie dinamic (cu ajutorul operatorului `new`). La afișare, în funcția `printf`, șirul specificator de format se continuă pe rândul următor (folosirea caracterului `\` pentru continuare).

3. Câmpuri de biți

Limbajul C oferă posibilitatea de prelucrare a datelor la nivel de bit. De multe ori se utilizează date care pot avea doar 2 valori (0 sau 1), cum ar fi datele pentru controlul unor dispozitive periferice, sau datele de valori mici. Declarând aceste date de tip *int* sau *short int*, în memorie se rezervă 16 biți. Alocarea unui număr atât de mare de locații de memorie nu este justificată, de aceea, limbajul C oferă posibilitatea declarării unor date pentru care să se aloce un număr specificat de biți (alocare pe biți).

Definiție: Un șir de biți adiacenți formează un **câmp de biți**.

Câmpurile de biți se pot declara ca membri ai unei structuri, astfel:

```
struct identificator_tip_struct {
    tip_elem_1 identificator_elem_1:lungime1;
    tip_elem_2 identificator_elem_2:lungime2;
    .
    .
    tip_elem_3 identificator_elem_3:lungime3;
} lista_identif_var_struct;
```

Lungime1, lungime2, etc. reprezintă lungimea fiecărui câmp de biți, rezervat pentru memorarea membrilor. Câmpurile se alocă de la biții de ordin inferior ai unui cuvânt (2 octeți), către cei de ordin superior (figura 1).

Exemplu:

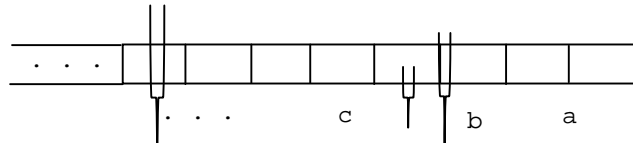


Figura 1. Câmpurile de biți a, b, c .

```
struct {
    int          a:    2;
    unsigned int b:    1;
    int          c:    3;
} x, y;
```

Câmpurile se referă ca orice membru al unei structuri, prin nume calificate:

Exemplu:

```
x.a = -1; x.b = 3; x.c = 4;
```

Utilizarea câmpurilor de biți impune următoarele restricții:

- ❑ Tipul membrilor poate fi *int* sau *unsigned int*.
- ❑ Lungime este o constantă întreagă din intervalul [0, 31];
- ❑ Un câmp de biți nu poate fi operandul unui operator de referențiere.
- ❑ Nu se pot organiza tablouri de câmpuri de biți.

Datorită restricțiilor pe care le impune folosirea câmpurilor de biți, cât și datorită faptului că aplicațiile care folosesc astfel de structuri de date au o portabilitate extrem de redusă (organizarea memoriei depinzând de sistemul de calcul), se recomandă folosirea câmpurilor de biți cu precauție, doar în situațiile în care se face o economie substanțială de memorie.

4. Declarații de tip

Limbajul C permite atribuirea unui nume pentru un tip (predefinit sau utilizator) de date. Pentru aceasta se folosesc declarațiile de tip. Forma generală a acestora este:

```
typedef tip nume_tip;
```

Nume_tip poate fi folosit la declararea datelor în mod similar cuvintelor cheie pentru tipurile predefinite.

Exemplu:

```
//1
typedef int INTREG;
INTREG x, y;
INTREG z=4;
//2
typedef struct{
    double parte_reală;
    double parte_imaginară;
} COMPLEX;
COMPLEX x, y;
```

5. Uniuni

Aceeași zonă de memorie poate fi utilizată pentru păstrarea unor obiecte (date) de diferite tipuri, prin declararea uniunilor. Uniunile sunt similare cu structurile, singura diferență constând în modul de memorare. Declararea uniunilor:

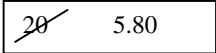
```
union identificator_tip_uniune {
    lista de declaratii_membrii;
} lista_identificatori_variabale;
```

Spațiul de memorie alocat corespunde tipului membrului de dimensiune maximă. Tipul uniune folosește aceeași zonă de memorie, care va conține informații organizate în mai multe moduri, corespunzător tipurilor membrilor.

Exemplu:

```
union numeric{
    int i;
    float f;
    double d;
} num;
num.i = 20;
num.f = 5.80;
cout<<sizeof(num)<<'\n'; //8
```

num.i
num.f
num.d



num

Figura 2. Modul de alocare a memoriei pentru variabila num (uniune) - 8 octeți.

Pentru variabile num se rezervă 8 octeți de memorie, dimensiunea maximă a zonei de memorie alocate membrilor (pentru int s-ar fi rezervat 2 octeți, pentru float 4, iar pentru double 8). În exemplul anterior, în aceeași zonă de memorie se păstrează fie o valoare întreagă (num.i=20), fie o valoare reală, dublă precizie (num.f=5.80).

Dacă pentru definirea tipului numeric s-ar fi folosit o structură, modul de alocare a memoriei ar fi fost cel din figura 3.

```

struct numeric{
    int i;
    float f;
    double d;
} num;
num.i = 20;
num.f = 5.80;
cout<<sizeof(num)<<'\n'; //14

```

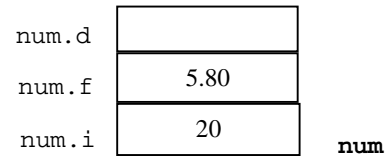


Figura 3. Modul de alocare a memoriei pentru variabila num (structură) - 14 octeți.

6. Enumerări

Tipul enumerare asociază fiecărui identificator o constantă întreagă. Sintaxa declarației:

```

enum identificador_tip_enumerare {
    identif_eleml = const1, . . .
} lista_identif_variabile;

```

Din declarație pot lipsi fie `identificador_tip_enumerare`, fie `lista_identif_variabile`. Pentru fiecare element al enumerării, constanta poate fi asociată în mod explicit (ca în declarația anterioară), fie implicit. În modul implicit nu se specifică nici o constantă, iar valoarea implicită este 0 pentru primul element, iar pentru restul elementelor, valoarea precedentă incrementată cu 1. Enumerările se folosesc în situațiile în care variabilele pot avea un număr mic de valori întregi, asociind un nume sugestiv pentru fiecare valoare.

Exemplu:

```

//1
enum boolean {FALSE, TRUE}; //definirea tipului boolean cu elementele FALSE si TRUE
//declaratie echivalenta cu enum boolean {FALSE=0, TRUE=1};
cout<<"FALSE este "<<FALSE<<'\n'; //FALSE este 0
//2
typedef enum temperatura {mica=-10, medie=10, mare=80};
//tipul enumerare temperatura, cu elementele mica (de valoare -10), medie (valoare 10), mare (valoare 80)
temperatura t1, t2; //declararea variabilelor t1, t2 de tip enumerare temperatura
t1=medie;
cout<<"t1="<<t1<<'\n'; //t1=10

```

Exercițiu: Să se citească (cu ajutorul unei funcții de citire) următoarele informații despre elevii participanți la un concurs de admitere: nume, numărul de înscriere și cele trei note obținute. Să se afișeze, printr-o funcție, informațiile citite. Să se afișeze o listă cu elevii participanți la concurs, ordonați alfabetic, notele și media obținută (funcție de ordonare, funcție de calculare a mediei). Să se afișeze lista elevilor înscriși la concurs, în ordinea descrescătoare a mediilor.

Sunt prezentate câteva modalități de implementare. În aceste variante apar doar funcția `cit_elev` (de citire) și `main`. S-a definit tipul `elev`. Se lucrează cu vectori de tip `elev`. În funcția `cit_elev` se validează fiecare notă. Se va observa modul de acces la membri structurii în funcția `cit_elev`. Dezavantajul principal al acestui mod de implementare îl constituie risipa de memorie, deoarece în funcția `main` se rezervă o

zonă de memorie continuă, pentru 100 de elemente de tip elev (100*sizeof(elev)).

```
#include <iostream.h>
#include <conio.h>
typedef struct elev{
    char nume[20];int nr_matr;int note[3];
}; //definirea tipului elev
void cit_elevi(elev a[], int n)
{for (int i=0; i<n; i++){
    cout<<"Nume elev:"; cin>>a[i].nume; //citirea numelui unui elev
    cout<<"Nr. inscriere:"; cin>>a[i].nr_matr;
    for (int j=0; j<3; j++){ // citirea notelor obtinute
        do{
            cout<<"Nota :"<<j+1<<" ="; cin>>a[i].note[j];
            if (a[i].note[j]<0 || a[i].note[j]>10) //validarea notei
                cout<<"Nota incorecta!....Repeta!\n";
            }while (a[i].note[j]<0 || a[i].note[j]>10);
        }
    }
}
void main()
{ int nr_elevi; clrscr();
  cout<<"Nr. elevi:";cin>>nr_elevi;
  elev p[100]; //declararea tabloului p, de tip elev
  cit_elevi(p, nr_elevi); //apel functie
}
```

În varianta următoare, se lucrează cu pointeri către tipul elev, iar memoria este alocată dinamic.

```
typedef struct elev{
    char nume[20];int nr_matr;int note[3];
}; //definirea tipului elev
void cit_elevi(elev *a, int n)
{
  for (int i=0; i<n; i++){
    cout<<"Nume elev:"; cin>>(a+i)->nume; //sau cin>>(*(a+i)).nume;
    cout<<"Nr. inscriere:"; cin>>(a+i)->nr_matr;
    for (int j=0; j<3; j++){
        do{
            cout<<"Nota :"<<j+1<<" =";
            cin>>(a+i)->note[j];
            if ((a+i)->note[j]<0 || (a+i)->note[j]>10)
                cout<<"Nota incorecta!....Repeta!\n";
            }while ((a+i)->note[j]<0 || (a+i)->note[j]>10);
        }
    }
}
void main()
{ int nr_elevi; clrscr();
  cout<<"Nr. elevi:";cin>>nr_elevi;
  elev *p; //declararea pointerului p, către tipul elev
  p=new elev[nr_elevi];
  //alocarea dinamică a memoriei, pentru un tablou cu nr_elevi elemente
  cit_elevi(p, nr_elevi); //apel functie
}
```

Implementarea tuturor funcțiilor:

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

```

#define DIM_PAG    24    //dimensiunea paginii de afisare
#define FALSE 0
#define TRUE 1
void ord_medii(elev *a, int n)
{
    int gata =FALSE;int i;double med1, med2;elev aux;
    while (!gata){
        gata=TRUE;
        for (i=0; i<=n-2; i++){
            med1=0;med2=0;
            for (int j=0; j<3; j++){
                med1+=(a+i)->note[j]; med2+=(a+i+1)->note[j];
                //calculul mediilor pentru elementele vecine
            }
            med1/=3; med2/=3;
            if (med1<med2){
                aux=(a+i); *(a+i)=*(a+i+1);*(a+i+1)=aux;
            }
            gata=FALSE; }
        }
    }
}

void ord_alf(elev *a, int n)
{
    int gata =FALSE;int i;double med1, med2;elev aux;
    while (!gata){
        gata=TRUE;
        for (i=0; i<=n-2; i++){
            if (strcmp( (a+i)->nume,(a+i+1)->nume) >0){
                aux=(a+i); *(a+i)=*(a+i+1);*(a+i+1)=aux;
                gata=FALSE;}
        }
    }
}

void cit_elevi(elev *a, int n);
// functie implementata anterior
void antet_afis(const char *s)
{printf("%s\n", s);
}

void afis_elev(elev *a, int n, char c)
{clrscr();
if (c=='A')
    antet_afis("          LISTA INSCRISILOR \n");
if (c=='O')
    antet_afis("          LISTA ALFABETICA \n");
if (c=='R')
    antet_afis("          LISTA MEDII          \n");
printf("Nr.crt.|Nr. Matricol|          NUME          |Nota1|Nota2|Nota3|
MEDIA\    |\n");
printf("-----\n");
int lin=3;
for (int i=0; i<n; i++){
    printf("%7d|%12d|%-20s|",i+1,(a+i)->nr_matr,(a+i)->nume);
    double med=0;
    for (int j=0; j<3; j++){
        printf("%-5d|", (a+i)->note[j]);
        med+=(a+i)->note[j];
    }
    med/=3;printf("%-9.2f|\n", med);lin++;
}
}

```

```

        if (lin==(DIM_PAG-1)){
            printf(" Apasa o tasta...."); getch();
            clrscr();
            if (c=='A') antet_afis("        LISTA INSCRISILOR \n");
            if (c=='O') antet_afis("        LISTA ALFABETICA  \n");
            if (c=='R') antet_afis("        LISTA MEDII      \n");
            printf("Nr.crt.|    NUME    |Nota1|Nota2|Nota3| MEDIA\  |\n");
            printf("-----\n");
            int lin=3;
        }
    }
    printf(" Apasa o tasta...."); getch();
}

void main()
{ int nr_elevi; clrscr();
  cout<<"Nr. elevi:";cin>>nr_elevi;
  elev *p; p=new elev[nr_elevi];
  cit_elevi(p, nr_elevi);
  afis_elev(p, nr_elevi, 'A');//afisarea inscrisilor
  ord_medii(p, nr_elevi);
  afis_elev(p, nr_elevi, 'R');//afisarea in ordinea descrescatoare a mediilor
  ord_alf(p, nr_elevi); //ordonare alfabetica
  afis_elev(p, nr_elevi, 'O');//afisarea in ordinea descrescatoare a mediilor
}

```

S-au implementat următoarele funcții:

- `cit_elevi` - citește informațiile despre elevii înscriși.
- `afis_elevi` - afișează informațiile despre elevi. Această funcție este folosită pentru cele trei afișări (lista înscrișilor, lista alfabetică și clasamentul în ordinea descrescătoare a mediilor). Afișarea se realizează cu ajutorul funcției `printf`, care permite formatarea datelor afișate. Afișarea se realizează ecran cu ecran (se folosește variabila `lin` care contorizează numărul de linii afișate), cu pauză după fiecare ecran. La începutul fiecărei pagini se afișează titlul listei - corespunzător caracterului transmis ca parametru funcției - și capul de tabel. De asemenea, pentru fiecare elev înscris se calculează media obținută (variabila `med`).
- `ord_medii` - ordonează vectorul de elevi (transmis ca parametru, pointer la tipul `elev`), descrescător, după medii. Se aplică metoda BubbleSort, comparându-se mediile elementelor vecine (`med1` reprezintă media elementului de indice `i`, iar `med2` - a celui de indice `i+1`) ale vectorului.
- `ord_alf` - ordonează vectorul de elevi (transmis ca parametru, pointer la tipul `elev`), crescător, după informația conținută de membrul `nume`. Pentru compararea numelor se folosește funcția `strcmp`.

Deoarece este foarte probabil ca vectorul înscrișilor să aibă multe elemente, pentru ordonări, ar fi fost mai eficientă metoda QuickSort; s-a folosit BubbleSort pentru a nu complica prea mult problema.

Întrebări și exerciții

Întrebări teoretice

1. Variabilele tablou și variabilele de tip definit de utilizator sunt exemple de variabile compuse (reprezintă date structurate). Care este, totuși, deosebirea dintre ele?
2. Ce posibilități de definire a unor noi tipuri de date vă oferă limbajul C/C++?

3. În ce constă diferența dintre structuri și uniuni?
4. Cum se numesc componentele unei structuri?
5. Ce restricții impune folosirea câmpurilor de biți?
6. Există vreo restricție referitoare la tipul membrilor unei structuri? Dacă da, care este aceasta?


Exerciții practice

1. Să se implementeze programele cu exemplele prezentate.
2. Să se scrie programele pentru exercițiile rezolvate care au fost prezentate.
3. Realizați următoarele modificări la exercițiul prezentat la sfârșitul **capitolului**:
 - a) Completați cu o funcție de calcul și afișare a mediei notelor tuturor candidaților pentru fiecare probă (media tuturor elevilor la proba1, media la proba2, etc).
 - b) Modificați lista alfabetică, astfel încât la elevii cu medie peste 5, să apară (alături de medie) mesajul "Promovat", iar la ceilalți, mesajul "Nepromovat".
 - c) Considerând că rezultatelor obținute sunt utilizate la un concurs de admitere, la care există N locuri (N introdus de la tastatură), și de faptul că pentru a fi admis media trebuie să fie cel puțin 5, să se afișeze lista admișilor și lista respinșilor, în ordinea descrescătoare a mediilor, în limita locurilor disponibile.
4. Să se scrie un program care să permită memorarea datelor privitoare la angajații unei firme mici: nume angajat, adresă, număr copii, sex, data nașterii, data angajării, calificare, salariul brut. Se vor implementa următoarele funcții:
 - a) Citirea informațiilor despre cei N angajați (N introdus de la tastatură);
 - b) Căutarea - după nume - a unui angajat și afișarea informațiilor despre acesta;
 - c) Modificarea informațiilor despre un anumit angajat;
 - d) Lista alfabetică a angajaților, în care vor apare: nume, adresă, data angajării, calificare, salariu;
 - e) Lista angajaților în ordone descrescătoare a vechimii;
 - f) Lista angajaților cu un anumit număr de copii, C, introdus de la tastatură;
 - g) Lista angajaților cu vârsta mai mare decât V (V introdus de la tastatură);
 - h) Salariul minim, salariul mediu și cel maxim din firmă;
 - i) Lista de salarii, în care vor apare: numele, calificarea, salariul brut și salariul net. La sfârșitul listei vor apare totalurile pentru salariile brute, impozite, salarii nete. Pentru calculul salariului net se aplică următoarele reguli de impozitare:
 - i.1) $I=15\%$ pentru salariul brut (SB)<600000
 - i.2) $I=50000+20\%$ pentru $600000 \leq SB < 1500000$ (20% din ceea ce depășește 600000)
 - i.3) $I=100000+30\%$ pentru $1500000 \leq SB < 3000000$
 - i.4) $I=250000+40\%$ pentru $3000000 \leq SB < 15000000$
 - i.5) $I=45\%$ pentru $SB \geq 15000000$

1. Caracteristicile generale ale fișierelor	4.3. Prelucrarea la nivel de șir de caractere
2. Deschiderea unui fișier	4.4. Intrări/ieșiri formate
3. Închiderea unui fișier	5. Intrări/ieșiri binare
4. Prelucrarea fișierelor text	6. Poziționarea într-un fișier
4.1. Prelucrarea la nivel de caracter	7. Funcții utilitare pentru lucrul cu fișiere
4.2. Prelucrarea la nivel de cuvânt	8. Alte operații cu fișiere

1. Caracteristicile generale ale fișierelor

Noțiunea de **fișier** desemnează o colecție de informații memorată pe un suport permanent (de obicei discuri magnetice), percepută ca un ansamblu, căreia i se asociază un nume (în vederea conservării și regăsirii ulterioare).

 **Caracteristicile unui fișier** (sub sistem de operare MS-DOS) sunt :

- **Dispozitivul logic de memorare (discul);**
- **Calea** (în structura de directoare) unde este memorat fișierul;
- **Numele și extensia;**
- **Atributele** care determină operațiile care pot fi efectuate asupra fișierului (de exemplu: **R**-read-only - citire; **W**-write-only scriere; **RW**-read-write citire/scriere; **H**-hidden - nu se permite nici măcar vizualizarea; **S**-system - fișiere sistem asupra cărora numai sistemul de operare poate realiza operații operații, etc.).

 **Lucrul cu fișiere în programare** oferă următoarele avantaje:

- Prelucrarea de unei cantități mari de informație obținută din diverse surse cum ar fi execuția prealabilă a unui alt program;
- Stocarea temporară pe suport permanent a informației în timpul execuției unui program pentru a evita supraîncărcarea memoriei de lucru;
- Prelucrarea aceleiași colecții de informații prin mai multe programe.

În limbajul C, operațiile asupra fișierelor se realizează cu ajutorul unor funcții din biblioteca standard (`stdio.h`). Transferurile cu dispozitivele periferice (tastatură, monitor, disc, imprimantă, etc.) se fac prin intermediul unor dispozitive logice identice numite *stream-uri* (fluxuri) și prin intermediul sistemului de operare. Un *flux de date* este un fișier sau un dispozitiv fizic tratat printr-un pointer la o structură de tip *FILE* (din header-ul `stdio.h`). Când un program este executat, în mod automat, se deschid următoarele fluxuri de date predefinite, dispozitive logice (în `stdio.h`):

- **`stdin`** (standard input device) - dispozitivul standard de intrare (tastatura) - ANSI C;
- **`stdout`** (standard output device) - dispozitivul standard de ieșire (monitorul) - ANSI C;
- **`stderr`** (standard error output device) - dispozitivul standard de eroare (de obicei un fișier care conține mesajele de eroare rezultate din execuția unor funcții) - ANSI C;
- **`stdaux`** (standard auxiliary device) - dispozitivul standard auxiliar (de obicei interfața

serială auxiliară) - specifice MS-DOS;

- **stdprn** (standard printer) - dispozitivul de imprimare - specifice MS-DOS.

În abordarea limbajului C (impusă de **stdio.h**), toate elementele care pot comunica informații cu un program sunt percepute - în mod unitar - ca fluxuri de date. Datele introduse de la tastatură formează un *fișier de intrare* (*fișierul standard de intrare*). Datele afișate pe monitor formează un *fișier de ieșire* (*fișierul standard de ieșire*). Sfârșitul oricărui fișier este indicat printr-un marcaj de *sfârșit de fișier* (end of file). În cazul fișierului standard de intrare, sfârșitul de fișier se generează prin **Ctrl+Z** (^Z) (sub MS-DOS) (sau **Ctrl+D** sub Linux). Acest caracter poate fi detectat prin folosirea constantei simbolice **EOF** (definită în fișierul *stdio.h*), care are valoarea **-1**. Această valoare nu rămâne valabilă pentru fișierele binare, care pot conține pe o poziție oarecare caracterul '\x1A'.

De obicei, schimbul de informații dintre programe și periferice se realizează folosind zone tampon. O zonă tampon păstrează una sau mai multe înregistrări. Prin *operația de citire*, înregistrarea curentă este transferată de pe suportul extern în zona tampon care îi corespunde, programul având apoi acces la elementele înregistrării din zona tampon. În cazul *operației de scriere*, înregistrarea se construiește în zona tampon, prin program, fiind apoi transferată pe suportul extern al fișierului. În cazul monitoarelor, înregistrarea se compune din caracterele unui rând. De obicei, o zonă tampon are lungimea multiplu de 512 octeți. Orice fișier trebuie *deschis* înainte de a fi prelucrat, iar la terminarea prelucrării lui, trebuie *închis*.

Fluxurile pot fi de tip **text** sau de tip **binar**. Fluxurile de tip text împart fișierele în linii separate prin caracterul '\n' (newline=linie nouă), putând fi citite ca orice fișier text. Fluxurile de tip binar transferă blocuri de octeți (fără nici o structură), neputând fi citite direct, ca fișierele text.

Prelucrarea fișierelor se poate face la *două niveluri*:

- Nivelul *superior* de prelucrare a fișierelor în care se utilizează funcțiile specializate în prelucrarea fișierelor.
- Nivelul *inferior* de prelucrare a fișierelor în care se utilizează direct facilitățile oferite de sistemul de operare, deoarece, în final, sarcina manipulării fișierelor revine sistemului de operare. Pentru a avea acces la informațiile despre fișierele cu care lucrează, sistemul de operare folosește câte un descriptor (bloc de control) pentru fiecare fișier.

Ca urmare, există două abordări în privința lucrului cu fișiere:

- abordarea implementată în **stdio.h**, asociază referinței la un fișier un **stream** (flux de date), un pointer către o structură **FILE**.
- abordarea definită în header-ul **io.h** (input/output header) asociază referinței la un fișier un așa-numit **handle** (în cele ce urmează acesta va fi tradus prin indicator de fișier) care din punct de vedere al tipului de date este **in;**

Scopul lucrului cu fișiere este acela de a prelucra informația conținută. Pentru a putea accesa un fișier va trebui să-l asociem cu unul din cele două modalități de manipulare. Acest tip de operație se mai numește deschidere de fișier. Înainte de a citi sau scrie într-un fișier (neconectat automat programului), fișierul trebuie deschis cu ajutorul funcției **fopen** din biblioteca standard. Funcția primește ca argument numele extern al fișierului, negociază cu sistemul de operare și returnează un nume (identificator) intern care va fi utilizat ulterior la prelucrarea fișierului. Acest identificator intern este un pointer la o structură care conține informații despre fișier (poziția curentă în buffer, dacă se citește sau se scrie în fișier, etc.). Utilizatorii nu

trebuie să cunoască detaliile, singura declarație necesară fiind cea pentru pointerul de fișier.

Exemplu: **FILE *fp;**

Operațiile care pot fi realizate asupra fișierelor sunt:

- ❑ *deschiderea unui fișier;*
- ❑ *scrierea într-un fișier;*
- ❑ *citirea dintr-un fișier;*
- ❑ *poziționarea într-un fișier;*
- ❑ *închiderea unui fișier.*

2. Deschiderea unui fișier

- ❑ **Funcția `fopen`.** Crează un flux de date între fișierul specificat prin numele extern (`nume_fișier`) și programul C. Parametrul `mod` specifică sensul fluxului de date și modul de interpretare a acestora. Funcția returnează un pointer spre tipul `FILE`, iar în caz de eroare - pointerul `NULL` (prototip în `stdio.h`).

FILE *fopen(const char *nume_fișier, const char *mod);

Parametrul `mod` este o constantă șir de caractere, care poate conține caracterele cu semnificațiile:

- **r** : flux de date de intrare; deschidere pentru citire;
- **w** : flux de date de ieșire; deschidere pentru scriere (crează un fișier nou sau suprascrie conținutul anterior al fișierului existent);
- **a** : flux de date de ieșire cu scriere la sfârșitul fișierului, adăugare, sau crearea fișierului în cazul în care acesta nu există;
- **+** : extinde un flux de intrare sau ieșire la unul de intrare/ieșire; operații de scriere și citire asupra unui fișier deschis în condițiile `r`, `w` sau `a`.
- **b** : date binare;
- **t** : date text (modul implicit).

Exemple:

"r+" – deschidere pentru modificare (citire și scriere);
"w+" – deschidere pentru modificare (citire și scriere);
"rb" – citire binară;
"wb" – scriere binară;
"r+b" – citire/scriere binară.

- ❑ **Funcția `freopen` (`stdio.h`).** Asociază un nou fișier unui flux de date deja existent, închizând legătura cu vechiul fișier și încercând să deschidă una nouă, cu fișierul specificat. Funcția returnează pointerul către fluxul de date specificat, sau `NULL` în caz de eșec (prototip în `stdio.h`).

**FILE*freopen(const char*fiș,const char*mod,FILE
*flux_date);**

- ❑ **Funcția `open`.** Deschide fișierul specificat conform cu restricțiile de acces precizate în apel. Returnează un întreg care este un indicator de fișier sau -1 (în caz de eșec) (prototip în `io.h`).

int open(const char *nume_fișier, int acces [,int mod]);

Restricțiile de acces se precizează prin aplicarea operatorului `|` (disjuncție logică la

nivel de bit) între anumite constante simbolice, definite în **fcntl.h**, cum sunt :

O_RDONLY	- citire;
O_WRONLY	- scriere;
O_RDWR	- citire și scriere;
O_CREAT	- creare;
O_APPEND	- adăugare la sfârșitul fișierului;
O_TEXT	- interpretare CR-LF;
O_BINARY	- nici o interpretare.

Restricțiile de mod de creare se realizează cu ajutorul constantelor:

S_IREAD - permisiune de citire din fișier;

S_IWRITE - permisiune de scriere din fișier, eventual legate prin operatorul “|”.

- ❑ **Funcția creat.** Crează un fișier nou sau îl suprascrie în cazul în care deja există. Returnează indicatorul de fișier sau -1 (în caz de eșec). Parametrul **un_mod** este obținut în mod analog celui de la funcția de deschidere (prototip în **io.h**).

int creat(const char *nume_fișier, int un_mod);

- ❑ **Funcția creatnew.** Crează un fișier nou, conform modului specificat. Returnează indicatorul fișierului nou creat sau rezultat de eroare (-1), dacă fișierul deja există (prototip în **io.h**).

int creatnew(const char *nume_fișier, int mod);

După cum se observă, informația furnizată pentru deschiderea unui fișier este aceeași în ambele abordări, diferența constând în *tipul de date* al entității asociate fișierului. Implementarea din **io.h** oferă un alt tip de control la nivelul comunicării cu echipamentele periferice (furnizat de funcția **ioctl**), asupra căruia nu vom insista, deoarece desfășurarea acestui tip de control este mai greoaie, dar mai profundă.

3. Închiderea unui fișier

- ❑ **Funcția fclose**

int fclose(FILE *pf);

Funcția închide un fișier deschis cu **fopen** și eliberează memoria alocată (zona tampon și structura **FILE**). Returnează valoarea 0 la închiderea cu succes a fișierului și -1 în caz de eroare (prototip în **stdio.h**).

- ❑ **Funcția fcloseall**

int fcloseall(void);

Închide toate fluxurile de date și returnează numărul fluxurilor de date închise (prototip în **stdio.h**).

- ❑ **Funcția close**

int close(int indicator);

Închide un indicator de fișier și returnează 0 (în caz de succes) sau -1 în caz de eroare (prototip în **io.h**).

4. Prelucrarea fișierelor text

După deschiderea unui fișier, toate operațiile asupra fișierului vor fi efectuate cu pointerul său. Operațiile de citire și scriere într-un fișier text pot fi:

- ❑ intrări/ieșiri la nivel de caracter (de octet);
- ❑ intrări/ieșiri la nivel de cuvânt (2 octeți);

- intrări/ieșiri de șiruri de caractere;
- intrări/ieșiri cu formatare.

Comunicarea de informație de la un fișier către un program este asigurată prin *funcții de citire* care transferă o cantitate de octeți (unitatea de măsură în cazul nostru) din fișier într-o variabilă-program pe care o vom numi buffer, ea însăși având sensul unei înșirui de octeți prin declarația **void *buf**. Comunicarea de informație de la un program către un fișier este asigurată prin *funcții de scriere* care transferă o cantitate de octeți dintr-o variabilă-program de tip buffer în fișier.

Fișierele sunt percepute în limbajul C ca fiind, implicit, secvențiale (informația este parcursă succesiv, element cu element). Pentru aceasta, atât fluxurile de date cât și indicatorii de fișier au asociat un *indicator de poziție curentă* în cadrul fișierului. Acesta este inițializat la 0 în momentul deschiderii, iar operațiile de citire, respectiv scriere, se referă la succesiunea de octeți care începe cu poziția curentă. Operarea asupra fiecărui octet din succesiune determină incrementarea indicatorului de poziție curentă.

4.1. Prelucrarea unui fișier la nivel de caracter. Fișierele pot fi scrise și citite caracter cu caracter folosind funcțiile **putc** (pentru scriere) și **getc** (citire).

□ Funcția putc

```
int putc (int c, FILE *pf);
```

c – este codul ASCII al caracterului care se scrie în fișier;

pf – este pointerul spre tipul **FILE** a cărui valoare a fost returnată de funcția *fopen*.

Funcția **putc** returnează valoarea lui **c** (valoarea scrisă în caz de succes), sau – 1 (EOF) în caz de eroare sau sfârșit de fișier.

□ Funcția getc

```
int getc (FILE *pf);
```

Funcția citește un caracter dintr-un fișier (pointerul spre tipul **FILE** transmis ca argument) și returnează caracterul citit sau EOF la sfârșit de fișier sau eroare.

Exercițiu: Să se scrie un program care crează un *fișier text* în care *se vor scrie caracterele* introduse de la tastatură (citite din fișierul standard de intrare), până la întâlnirea caracterului ^Z = Ctrl+Z.

```
#include <stdio.h>
#include <process.h>
void main()
{
    int c, i=0; FILE *pfc;
    char mesaj[]="\nIntrodu caractere urmate de Ctrl+Z (Ctrl+D sub Linux):\n";
    char eroare[]="\n Eroare deschidere fișier \n";
    while(mesaj[i]) putchar(mesaj[i++]);
    pfc=fopen("f_carl.txt","w"); // crearea fișierului cu numele extern f_carl.txt
    if(pfc==NULL)
    {
        i=0;
        while(eroare[i])putc(eroare[i++],stdout);
        exit(1);
    }while((c=getchar())!=EOF) // sau: while ((c=getc(stdin)) != EOF)
        putc(c,pfc); // scrierea caracterului în fișier
    fclose(pfc); // închiderea fișierului
}
```

Exercițiu: Să se scrie un program care citește **un fișier text**, caracter cu caracter, și afișează conținutul acestuia.

```
#include <stdio.h>
#include <process.h>
void main()
{
    int c, i=0;
    FILE *pfcara;
    char eroare[]="\n Eroare deschidere fișier \n";
    pfcara=fopen("f_car1.txt", "r");          //deschiderea fișierului numit f_car1.txt în
    citire
    if(pfcara==NULL)
    {
        i=0;
        while(eroare[i])putc(eroare[i++], stdout);
        exit(1);
    } while((c=getc(pfcara))!=EOF)          //citire din fișier, la nivel de caracter
        putc(c, stdout);
    //scrierea caracterului citit în fișierul standard de ieșire (afișare pe monitor)
    fclose(pfcara);
}
```

4.2. Prelucrarea unui fișier la nivel de cuvânt. Funcțiile **putw** și **getw** (putword și getword) sunt echivalente cu funcțiile **putc** și **getc**, cu diferența că unitatea transferată nu este un singur octet (caracter), ci un cuvânt (un int).

```
int getw(FILE *pf);
int putc (int w, FILE *pf);
```

Se recomandă utilizarea funcției **feof** pentru a testa întâlnirea sfârșitului de fișier.

Exemplu:

```
int tab[100];
FILE *pf;
//... deschidere fișier
while (!feof(pf)){
    for (int i=0; i<100; i++){
        if (feof(pf))
            break;
        tab[i]=getw(pf);
        //citire din fișier la nivel de cuvânt și memorare în vectorul tab
        //...
    }
}
printf("Sfarșit de fișier\n");
```

4.3. Prelucrarea unui fișier la nivel de șir de caractere. Într-un fișier text, liniile sunt considerate ca linii de text separate de sfârșitul de linie ('\\n'), iar în memorie, ele devin șiruri de caractere terminate de caracterul nul ('\\0'). Citirea unei linii de text dintr-un fișier se realizează cu ajutorul funcției **fgets**, iar scrierea într-un fișier - cu ajutorul funcției **fputs**.

Funcția **fgets** este identică cu funcția **gets**, cu deosebirea că funcția **gets** citește din fișierul standard de intrare (**stdin**). Funcția **fputs** este identică cu funcția **puts**, cu deosebirea funcția **puts** scrie în fișierul standard de ieșire (**stdout**).

□ **Funcția fputs**

```
int fputs(const char *s, FILE *pf);
```

Funcția scrie un șir de caractere într-un fișier și primește ca argumente pointerul spre zona de memorie (buffer-ul) care conține șirul de caractere (s) și pointerul spre structura FILE. Funcția returnează ultimul caracter scris, în caz de succes, sau -1 în caz de eroare.

□ Funcția fgets

```
char *fgets(char *s, int dim, FILE *pf);
```

Funcția citește maximum dim-1 octeți (caractere) din fișier, sau până la întâlnirea sfârșitului de linie. Pointerul spre zona în care se face citirea caracterelor este s. Terminatorul null ('\0') este plasat automat la sfârșitul șirului (buffer-ului de memorie). Funcția returnează un pointer către buffer-ul în care este memorat șirul de caractere, în caz de succes, sau pointerul NULL în cazul eșecului.

Exercițiu: Să se scrie un program care crează un fișier text în care se vor scrie șirurile de caractere introduse de la tastatură.

```
#include <stdio.h>
void main()
{ int n=250; FILE *pfsir;
  char mesaj[]="\nIntrodu siruri car.urmate de Ctrl+Z(Ctrl+D sub
Linux):\n";
  char sir[250],*psir; fputs(mesaj,stdout);
  pfsir=fopen("f_sir.txt","w"); //deschiderea fișierului f_sir.txt pentru scriere
  psir=fgets(sir,n,stdin); // citirea șirurilor din fișierul standard de intrare
  while(psir!=NULL)
  { fputs(sir,pfsir); // scrierea în fișierul text
    psir=fgets(sir,n,stdin);
  }
  fclose(pfsir);
}
```

Exercițiu: Să se scrie un program care citește un fișier text, linie cu linie, și afișează conținutul acestuia

```
#include <stdio.h>
void main()
{ int n=250; FILE *pfsir; char sir[250],*psir;
  pfsir=fopen("f_sir.txt","r"); psir=fgets(sir,n,pfsir);
  while(psir!=NULL)
  { fputs(sir,stdout); //sau: puts(sir);
    //afișarea (scrierea în fișierul standard de ieșire) șirului (liniei) citit din fișierul text
    psir=fgets(sir,n,pfsir); //citirea unei linii de text din fișier
  }
  fclose(pfsir);}
```

4.4. Intrări/ieșiri formate. Operațiile de intrare/ieșire formate permit citirea, respectiv scrierea într-un fișier text, impunând un anumit format. Se utilizează funcțiile **fscanf** și **fprintf**, similare funcțiilor **scanf** și **printf** (care permit citirea/scrierea formatată de la tastatură/monitor).

□ Funcția fscanf

```
int fscanf(FILE *pf, const char *format, . . .);
```

□ Funcția fprintf

```
int fprintf(FILE *pf, const char *format, . . .);
```

Funcțiile primesc ca parametri ficși pointerul (pf) spre tipul FILE (cu valoarea

atribuită la apelul funcției `fopen`), și specificatorul de format (cu structură identică celui prezentat pentru funcțiile `printf` și `scanf`). Funcțiile returnează numărul câmpurilor citite/scrise în fișier, sau -1 (EOF) în cazul detectării sfârșitului fișierului sau al unei erori.

5. Intrări/ieșiri binare

Reamintim că fluxurile de tip binar transferă blocuri de octeți (fără nici o structură), neputând fi citite direct, ca fișierele text (vezi [paragraful 8.1.](#)). Comunicarea de informație dintre un program și un fișier este asigurată prin funcții de citire/scriere care transferă un număr de octeți, *prin intermediul unui buffer*.

Funcțiile de citire

- **Funcția `fread`.** Citește date dintr-un flux, sub forma a `n` blocuri (entități), fiecare bloc având dimensiunea `dim`, într-un buffer (`buf`). Returnează numărul de blocuri citite efectiv, sau -1 în caz de eroare (prototip în **stdio.h**).

```
size_t fread(void *buf, size_t dim, size_t n, FILE *flux_date);
```

- **Funcția `read`.** Citește dintr-un fișier (precizat prin indicatorul său, `indicator`) un număr de `n` octeți și îi memorează în bufferul `buf`. Funcția returnează numărul de octeți citați efectiv (pentru fișierele deschise în mod text nu se numără simbolurile de sfârșit de linie), sau -1 în caz de eroare (prototip în **io.h**).

```
int read(int indicator, void *buf, unsigned n);
```

Funcțiile de scriere. Fișierele organizate ca date binare pot fi prelucrate cu ajutorul funcțiilor `fread` și `fwrite`. În acest caz, se consideră că înregistrarea este o *colecție de date structurate* numite **articole**. La o citire se transferă într-o zonă specială, numită **zona tampon**, un număr de articole care se presupune că au o lungime fixă.

- **Funcția `fwrite`.** Scrie informația (preluată din buffer, `buf` este pointerul spre zona tampon care conține articolele citite) într-un flux de date, sub forma a `n` entități de dimensiune `dim`. Returnează numărul de entități scrise efectiv, sau -1 în caz de eroare (prototip în **stdio.h**).

```
size_t fwrite(const void *buf, size_t dim, size_t n, FILE *flx_date);
```

- **Funcția `write`.** Scrie într-un fișier (desemnat prin indicatorul său, `indicator`) un număr de `n` octeți preluați dintr-un buffer (`buf` este pointerul spre acesta). Returnează numărul de octeți scriși efectiv sau -1 în caz de eroare (prototip în **io.h**).

```
int write(int indicator, void *buf, unsigned n);
```

Exercițiu: Să se scrie un program care crează un fișier binar în care se vor introduce numere reale, nenule.

```
#include <iostream.h>
#include <stdio.h>
int main()
{ FILE *f; double nr; int x;
  if ((f= fopen("test_nrb.dat", "wb")) == NULL) //deschidere flux binar, scriere
  {
    cout<<"\nNu se poate deschide fișierul test_nrb.dat"<<"\n";
    return 1;
  }
  cout<<"\nIntroduceți numere(diferite de 0) terminate cu un
0:"<<"\n";
  cin>>nr;
  while(nr!=0)
```

```

{
    x=fwrite(&nr, sizeof(nr), 1, f);    //scriere în fișier
    cin>>nr;
}
fclose(f);
return 0;
}

```

Exemplu: Să se scrie un program ce citește dintr-un fișier binar numere reale, nenule.

```

#include <iostream.h>
#include <stdio.h>
int main()
{
    FILE *f; double buf;
    if ((f= fopen("test_nrb.dat", "rb")) == NULL)
    {
        cout<<"\nNu se poate deschide fișierul test_nrb.dat"<<"\n";
        return 1;
    }
    cout<<"\nNumerele nenule citite din fișier sunt:"<<"\n";
    while((fread(&buf, sizeof(buf), 1, f))==1)
        // funcția sizeof(buf) care returneaza numarul de octeți necesari variabilei buf.
        cout<<buf<<" ";
    fclose(f);
    cout<<"\n";
    return 0;
}

```

6. Poziționarea într-un fișier

Pe lângă mecanismul de poziționare implicit (asigurat prin operațiile de citire și scriere) se pot folosi și operațiile de poziționare explicită.

□ Funcția **fseek**

int fseek(FILE *pf, long deplasament, int referința);

Funcția deplasează capul de citire/scriere al discului, în vederea prelucrării înregistrărilor fișierului într-o ordine oarecare. Funcția setează poziția curentă în fluxul de date la *n* octeți față de referință):

- deplasament – definește numărul de octeți peste care se va deplasa capul discului;
- referința – poate avea una din valorile:
 - 0 - începutul fișierului (SEEK_SET);
 - 1 - poziția curentă a capului (SEEK_CUR);
 - 2 - sfârșitul fișierului (SEEK_END).

Funcția returnează valoarea zero la poziționarea corectă și o valoare diferită de zero în caz de eroare (prototip în **stdio.h**).

□ Funcția **lseek**

int lseek(int indicator, long n, int referinta);

Setează poziția curentă de citire/scriere în fișier la *n* octeți față de referință. Returnează valoarea 0 în caz de succes și diferită de zero în caz de eroare (prototip în **io.h**).

□ Funcția **fgetpos**

int fgetpos(FILE *flux_date, fpos_t *poziție);

Determină poziția curentă (pointer către o structură, **fpos_t**, care descrie această

poziție în fluxul de date). Înscrie valoarea indicatorului în variabila indicată de `poziție`. Returnează 0 la determinarea cu succes a acestei poziții sau valoare diferită de zero în caz de eșec. Structura care descrie poziția poate fi transmisă ca argument funcției `fsetpos` (prototip în **stdio.h**).

□ **Funcția `fsetpos`**

`int fsetpos(FILE *flux_date, const fpos_t *poziție);`

Setează poziția curentă în fluxul de date (atribuie indicatorului valoarea variabilei indicate `poziție`), la o valoare obținută printr apelul funcției `fgetpos`. Returnează valoarea 0 în caz de succes, sau diferită de 0 în caz de eșec (prototip în **stdio.h**).

Există funcții pentru modificarea valorii indicatorului de poziție și de determinare a poziției curente a acestuia.

□ **Funcția `ftell`**

`long ftell(FILE *pf);`

Indică poziția curentă a capului de citire în fișier. Funcția returnează o valoare de tip `long int` care reprezintă poziția curentă în fluxul de date (deplasamentul în octeți a poziției capului față de începutul fișierului) sau -1L în caz de eroare (prototip în **stdio.h**).

□ **Funcția `tell`**

`long tell(int indicator);`

Returnează poziția curentă a capului de citire/scriere în fișier (exprimată în număr de octeți față de începutul fișierului), sau -1L în caz de eroare (prototip în **io.h**).

□ **Funcția `rewind`**

`void rewind(FILE *flux_date);`

Poziționează indicatorul la începutul fluxului de date specificat ca argument (prototip în **stdio.h**).

7. Funcții utilitare pentru lucrul cu fișiere

Funcții *de testare a sfârșitului de fișier:*

□ **Funcția `feof`**

`int feof(FILE *flux_date);`

Returnează o valoare diferită de zero în cazul întâlnirii sfârșitului de fișier sau 0 în celelalte cazuri (prototip în **stdio.h**).

□ **Funcția `eof`**

`int eof(int indicator);`

Returnează valoarea 1 dacă poziția curentă este sfârșitul de fișier, 0 dacă indicatorul este poziționat în altă parte, sau -1 în caz de eroare (prototip în **io.h**).

Funcții *de golire a fluxurilor de date*

□ **Funcția `fflush`**

`int fflush(FILE *flux_date);`

Golește un fluxul de date specificat ca argument. Returnează 0 în caz de succes și -1 (EOF) în caz de eroare (prototip în **stdio.h**).

□ **Funcția `flushall`**

`int flushall(void);`

Golește toate fluxurile de date existente, pentru cele de scriere efectuând și scrierea în fișiere. Returnează numărul de fluxuri asupra cărora s-a efectuat operația (prototip în **stdio.h**).

8. Alte operații cu fișiere

Funcții care *permit operații ale sistemului de operare asupra fișierelor*

❑ Funcția **remove**

```
int remove(const char *nume_fișier);
```

Șterge un fișier. Returnează valoarea 0 pentru operație reușită și -1 pentru operație eșuată (prototip în **stdio.h**).

❑ Funcția **rename**

```
int rename(const char *nume_vechi, const char *nume_nou);
```

Redenumeste un fișier. Returnează 0 pentru operație reușită și -1 în cazul eșecului (prototip în **stdio.h**).

❑ Funcția **unlink**

```
int unlink(const char *nume_fișier);
```

Șterge un fișier. Returnează 0 la operație reușită și -1 la eșec; dacă fișierul are permisiune read-only, funcția nu va reuși operația (prototip în **io.h**, **stdio.h**).

Funcții care *permit manipularea aceluiași fișier prin două indicatoare de fișier independente*

❑ Funcția **dup**

```
int dup(int indicator);
```

Duplică un indicator de fișier. Returnează noul indicator de fișier pentru operație reușită sau -1 în cazul eșecului (prototip în **io.h**).

❑ Funcția **dup2**

```
int dup2(int indicator_vechi, int indicator_nou);
```

Duplică un indicator de fișier la valoarea unui indicator de fișier deja existent. Returnează 0 în caz de succes și -1 în caz de eșec (prototip în **io.h**).

Funcții pentru *afierea sau modificarea dimensiunii în octeți a fișierelor*

❑ Funcția **chsize**

```
int chsize(int indicator, long lungime);
```

Modifică dimensiunea unui fișier, conform argumentului lungime. Returnează 0 pentru operație reușită sau -1 în caz de eșec (prototip în **stdio.h**).

❑ Funcția **filelength**

```
long filelength(int indicator);
```

Returnează lungimea unui fișier (în octeți) sau -1 în caz de eroare (prototip în **io.h**).

Funcții de *lucru cu fișiere temporare* care oferă facilități de lucru cu fișiere temporare prin generarea de nume unice de fișier în zona de lucru.

❑ Funcția **tmpfile**

```
FILE *tmpfile(void);
```

Deschide un fișier temporar, ca flux de date, în mod binar (w+b). Returnează pointerul către fișierul deschis în cazul operației reușite, sau NULL în caz de eșec (prototip în **stdio.h**).

❑ Funcția **tmpnam**

```
char *tmpnam(char *spttr);
```

Crează un nume unic pentru fișierul temporar (prototip în **stdio.h**).

❑ Funcția **creattemp**

```
int creattemp(char *cale, int attrib);
```

Crează un fișier unic ca nume, cu atributele specificate în argumentul `attrib` (prin `_fmode`, `O_TEXT` sau `O_BINARY`), în directorul dat în argumentul `cale`. Returnează indicatorul (handler-ul) către fișierul creat sau -1 (și setarea `errno`) în cazul eșecului (prototip în **io.h**).

Exemplu: Să se creeze un fișier binar, care va conține informațiile despre angajații unei întreprinderi: nume, marca, salariu. Să se afișeze apoi conținutul fișierului.

```
#include<iostream.h>
#include <stdio.h>
#include <ctype.h>
typedef struct
{ char nume[20];int marca;double salariu;
}angajat;
union
{angajat a;char sbinar[sizeof(angajat)];}buffer;

int main()
{angajat a; FILE *pf; char cont;char *nume_fis;
cout<<"Nume fisier care va fi creat:"; cin>>nume_fis;
if ((pf= fopen(nume_fis, "wb")) == NULL)
{ cout<<"\nEroare creare fișier "<<nume_fis<<"!\n";
return 1; }
do
{cout<<"Marca : ";cin>>a.marca;
cout<<"Nume : ";cin>>a.nume;
cout<<"Salariu : ";cin>>a.salariu;
buffer.a=a;
fwrite(buffer.sbinar,1,sizeof(angajat),pf);
cout<<"Continuati introducerea de date (d/n) ?";
cin>>cont;
} while(toupper(cont)!='N');
fclose(pf);
//citirea informatiilor
if ((pf= fopen(nume_fis, "rb")) == NULL)
{ cout<<"\nEroare citire fișier "<<nume_fis<<"!\n";
return 1; }
for(;;)
{
fread(buffer.sbinar,1,sizeof(a),pf);
a=buffer.a;
if(feof(pf)) exit(1);
cout<<" Marca : "<<a.marca;
cout<<" Numele : "<<a.nume<<"\n";
cout<<" Salariul : "<<a.salariu<<"\n";
}
fclose(pf);
}
```

Exemplu: Aplicație pentru gestiunea materialelor dintr-un depozit. Aplicația va avea un meniu principal și va permite gestiunea următoarelor informații: codul materialului (va fi chiar "numărul de ordine"), denumirea acestuia, unitatea de măsură, prețul unitar, cantitatea contractată și cea recepționată (vectori cu 4 elemente). Memorarea datelor se va face într-un fișier de date (un fișier binar cu structuri), numit "material.dat". Aplicația conține următoarele funcții:

1. `help()` - informare privind opțiunile programului

2. Funcții pentru fișierele binare, care să suplinească lipsa funcțiilor standard pentru organizarea directă a fișierelor binare:

citireb() - citire în acces direct din fișier;
scrieb() - scriere în acces direct în fișier;
citmat() - citirea de la terminal a informațiilor despre un material;
afismat() - afișarea informațiilor despre un material (apelată de list);
lungfisis() - determinarea lungimii fișierului existent;
crefis() - creare fișier.

3. Funcții pentru adaugarea, modificarea, ștergerea și listarea de materiale.

```
#include <process.h>
#include <iostream.h>
#include <stdio.h>
#include <ctype.h>
typedef struct material
{ int codm,stoc,cant_c[4],cant_r[4];
  char den_mat[20],unit_mas[4];
  float pret;
};
material mat;
FILE *pf;
void crefis(),adaug(),modif(),sterg(),list(),help();
void main()
{ char opțiune;
  do //afișarea unui meniu de opțiuni și selecția opțiunii
  {cout<<"\n"<<"Opțiunea Dvs. de lucru este"<<"\n"
    <<"(c|a|m|s|l|e|h pentru help) : ";
    cin>>opțiune;
    switch(opțiune)
    {
      case 'c':case 'C':crefis();break;
      case 'a':case 'A':adaug();break;
      case 'm':case 'M':modif();break;
      case 's':case 'S':sterg();break;
      case 'l':case 'L':list();break;
      case 'h':case 'H':help();break;
      case 'e':case 'E': break;
      default:help(); break;
    }
  }while(toupper(opțiune)!='E');
}
void help() // afișare informații despre utilizarea meniului și opțiunile acestuia
{cout<<"Opțiunile de lucru sunt :"<<"\n";
  cout<<" C,c-creare fisier"<<"\n";
  cout<<" A,a-adaugare"<<"\n";
  cout<<" M,m-modificare"<<"\n";
  cout<<" L,l-listare"<<"\n";
  cout<<" S,s-ștergere"<<"\n";
  cout<<" H,h-help"<<"\n";
  cout<<" E,e-exit"<<"\n";
}
long int lungfis(FILE *f) // returnează lungimea fișierului
{long int posi,posf;
  posi=ftell(f); fseek(f,0,SEEK_END);
  posf=ftell(f); fseek(f,posi,SEEK_SET);
  return posf;
}
void scrieb(int nr,void *a,FILE *f) //scriere în fișierul binar
{long depl=(nr-1)*sizeof(material);
  fseek(f,depl,SEEK_SET);
  if(fwrite(a,sizeof(material),1,f)!=1)
  {cout<<"Eroare de scriere in fișier !"<<"\n";
    exit(1); }
```

```

    }
void citireb(int nr,void *a,FILE *f)    //citire din fişierul binar
{
    long depl=(nr-1)*sizeof(material);
    fseek(f,depl,SEEK_SET);
    if(fread(a,sizeof(material),1,f)!=1)
        {cout<<"Eroare de citire din fişier !"<<"\n";
         exit(2);    }
}

void afismat(material *a) //afişarea informaţiilor despre un anumit material
{
    int i;
    if(a->codm)
        {cout<<"Cod material      : "<<a->codm<<"\n";
         cout<<"Denumire material: "<<a->den_mat<<"\n";
         cout<<"Cantitaţi contractate:"<<"\n";
         for(i=0;i<4;i++)
             cout<<"Contractat  "<<i<<"      : "<<a->cant_c[i]<<"\n";
         cout<<"Cantitaţi recepţionate:"<<"\n";
         for(i=0;i<4;i++)
             cout<<"Receptionat "<<i<<"      : "<<a->cant_r[i]<<"\n";
         cout<<"Stoc              : "<<a->stoc<<"\n";
         cout<<"Unitate de masura:  "<<a->unit_mas<<"\n";
         cout<<"Preţ unitar       : "<<a->preţ<<"\n";
        }
    else    cout<<"Acest articol a fost şters !"<<"\n";
}

void citmat(material *a)    //citirea informaţiilor despre un anumit material
{
    int i;float temp;
    cout<<"Introduceţi codul materialului (0=End): ";cin>>a->codm;
    if(a->codm==0) return;
    cout<<"Introduceţi denumirea materialului : ";cin>>a->den_mat;
    cout<<"Introduceţi unitatea de măsură : ";cin>>a->unit_mas;
    cout<<"Introduceţi preţul : ";cin>>temp;a->preţ=temp;
    cout<<"Introduceţi cantităţile contractate : "<<"\n";
    for(i=0;i<4;i++)
        {cout<<"Contractat "<<i+1<<"      : ";cin>>a->cant_c[i]; }
    cout<<"Introduceţi cantităţile recepţionate : "<<"\n";
    for(i=0;i<4;i++)
        {cout<<"Receptionat "<<i+1<<"      : ";cin>>a->cant_r[i]; }
}

void crefis() //deschidere fisier
{
    if((pf=fopen("material.dat","r"))!=NULL)
        cout<<"Fişierul exista deja !"<<"\n";
    else
        pf=fopen("material.dat","w");
    fclose(pf);
}

void adaug() //adăugare de noi materiale
{
    int na; pf=fopen("material.dat","a");//deschidere pentru append
    na=lungfis(pf)/sizeof(material);
    do
        {citmat(&mat);
         if(mat.codm) scrieb(++na,&mat,pf);
        } while(mat.codm);
    fclose(pf);
}

void modif() //modificarea informaţiilor despre un material existent
{
    int na; char ch; pf=fopen("material.dat","r+");
    do
        { cout<<"Numarul articolului de modificat este (0=END): ";cin>>na;
          if(na)
              {citireb(na,&mat,pf);
               afismat(&mat);
              }
        }
    while(ch!='q');
    fclose(pf);
}

```

```

        cout<<"Modificați articol (D/N) ? :";
        do
        {
            cin>>ch;
            ch=toupper(ch);
        } while(ch!='D' && ch!='N');
        if(ch=='D')
        {
            citmat(&mat);
            scrieb(na,&mat,pf);
        }
    }
    }while(na);
    fclose(pf);
}

void sterg() //ștergerea din fișier a unui material
{
    int n; long int na; pf=fopen("material.dat","r+");
    mat.codm=0; na=lungfis(pf)/sizeof(material);
    do
    {
        do
        {
            cout<<"Numarul articolului de șters este (0=END): ";cin>>n;
            if(n<0||n>na) cout<<"Articol eronat"<<"\n";
        }while(!(n>=0 && n<=na));
        if(n) scrieb(n,&mat,pf);
    }while(n);
    fclose(pf);
}

void list() //afișare informații despre un anumit material
{
    int na; pf=fopen("material.dat","r");
    do
    {
        cout<<"Numarul articolului de listat este (0=END): ";cin>>na;
        if(na)
        {
            citireb(na,&mat,pf);
            afismat(&mat);
            cout<<"\n";
        }
    }while(na);
    fclose(pf);
}

```

Întrebări și exerciții

1. Scrieți un program de tipărire a conținuturilor mai multor fișiere, ale căror nume se transmit ca parametri către funcția main. Tipărirea se face pe ecran (lungimea paginii = 22) sau la imprimantă (lungimea paginii = 61). Conținutul fiecărui fișier va începe pe o pagină nouă, cu un titlu care indică numele fișierului. Pentru fiecare fișier, paginile vor fi numerotate (cu ajutorul unui contor de pagini).
2. Scrieți un program care citește un fișier text. Pornind de la conținutul acestuia, se va crea un alt fișier, prin înlocuirea spațiilor consecutive cu unul singur. Se vor afișa pe ecran conținutul fișierului de la care s-a pornit și conținutul fișierului obținut.
3. Să se consulte conținutul unui fișier și să se afișeze următoarele informații statistice: numărul de cuvinte din fișier, numărul de caractere, numărul de linii, numărul de date numerice (nu cifre, numere!).
4. Scrieți un program care să compare conținutul a două fișiere, și afișați primele linii care diferă și poziția caracterelor diferite în aceste linii.
5. Scrieți un program care citește conținutul unui fișier sursă scris în limbajul C și afișează în ordine alfabetică fiecare grup al numelor de variabile care au primele n caractere identice (n este citit de la tastatură).
6. Scrieți un program care consultă un fișier text și afișează o listă a tuturor cuvintelor din fișier. Pentru fiecare cuvânt se vor afișa și numerele liniilor în care apare cuvântul.

7. Scrieți un program care citește un text introdus de la tastatură și afișează cuvintele distincte, în ordinea crescătoare a frecvenței lor de apariție. La afișare, fiecare cuvânt va fi precedat de numărul de apariții.
8. Scrieți un program care citește un text introdus de la tastatură, ordonează alfabetic liniile acestuia și le afișează.
9. Scrieți o aplicație pentru gestiunea informațiilor despre cărțile existente într-o bibliotecă. Aplicația va avea un meniu principal care va permite:
 - a) Memorarea datelor într-un fișier (un fișier binar cu structuri), al cărui nume se introduce de la tastatură. Fișierul va conține informațiile: nume carte, autor, editura, anul apariției, preț. Pentru fiecare carte, se va genera o cotă (un număr unic care să constituie cheia de căutare).
 - b) Adăugarea de noi cărți;
 - c) Afișarea informațiilor despre o anumită carte;
 - d) Căutarea titlurilor după un anumit autor;
 - e) Modificarea informațiilor existente;
 - f) Lista alfabetică a tuturor autorilor;
 - g) Ștergerea unei cărți din bibliotecă;
 - h) Ordonarea descrescătoare după anul apariției;
 - i) Numele celei mai vechi cărți din bibliotecă;
 - j) Numele celei mai scumpe cărți din bibliotecă;
 - k) Numele autorului cu cele mai multe cărți;
 - l) Valoarea totală a cărților din bibliotecă.

BIBLIOGRAFIE

1. Borland C++ 4, Ghidul programatorului, Editura Teora, 1996
2. Brookshear, J.G., Introducere în informatică, Editura Teora, București, 1998
3. Bumbaru, S., Note de curs
4. Somnea, D., Turturea, D., Inițiere în C++ - Programarea orientată pe obiecte, Editura Tehnică, București, 1993
5. Spircu, C., Lopătan, I., POO-Analiza, proiectarea și programarea orientate spre obiecte, Editura Teora, București, 1996
6. Stroustrup, B., A Beginners C++, www.cs.uow.edu.au/people/nabg/ABC/ABC.html
7. Stroustrup, B., C++ Annotations, www.icce.rug.nl/docs/cplusplus/cplusplus.html