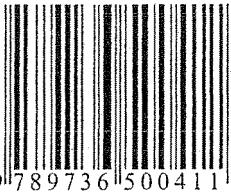


str. Observatorului 1, bl. OS1
3400 Cluj-Napoca
C.P. 186, of. Post. Cluj-Napoca 1
fax 064.198263
tel. 064.438328*
<http://www.gmi.ro>

I.S.B.N. 973-650-041-1



9789736500411

I.S.B.N. 973-650-043-8



00260

Cartea se adresează unui cerc larg de cititori care doresc să se inițieze în programarea și utilizarea limbajelor C și C++.

Primele două volume realizează o introducere elementară în aceste limbaje.

Volumul trei conține programe diverse în C și C++ cu aplicații la rezolvarea problemelor științifice și tehnico-ingenieresti, în prelucrari de date precum și în scrierea programelor de sistem.







CLUJ-NAPOCA
2001

P. M. Popescu
P. M. Popescu

LIMBAJUL C

**LIMBAJELE
C ȘI C++
PENTRU
ÎNCEPĂTORI
VOLUMUL I
Partea II-a**

Autor :

LIVIU NEGRESCU

- A confruntat cu originalul
Irina Mitrov

Editura Albastra

Director editură
Smaranda Derveșteanu

Coordonator serie
Codruța Poenaru

Coperta
Liviu Derveșteanu

Tiraj: 1000 exemplare

Tipărit
EDITURA ALBASTRĂ
comanda 142 / 2001



CUPRINS

10. STRUCTURI ȘI TIPURI DEFINITE DE UTILIZATOR	317
10.1. Declarația de structură	318
10.2. Accesul la elementele unei structuri	322
10.3. Asignări de nume pentru tipuri de date (declarații de tip)	325
Exerciții	329
10.4. Reuniune	358
Exerciții	361
10.5. Cimp	370
Exerciții	372
10.6. Tipul enumerare	384
10.7. Date definite recursiv	387
Exerciții	389
11. LISTE	397
11.1. Lista simplu înlățuită	398
11.1.1. Crearea unei liste simplu înlățuite	399
11.1.2. Accesul la un nod al unei liste simplu înlățuite	402
11.1.3. Inserarea unui nod intr-o listă simplu înlățuită	404
11.1.3.1. Inserarea unui nod intr-o listă simplu înlățuită înaintea primului ei nod	405
11.1.3.2. Inserarea unui nod intr-o listă simplu înlățuită înaintea unui nod precizat printr-o cheie	406
11.1.3.3. Inserarea unui nod intr-o lista simplu înlățuită după un nod precizat printr-o cheie	408
11.1.3.4. Adăugarea unui nod la o listă simplu înlățuită	409
11.1.4. Stergerea unui nod dintr-o listă simplu înlățuită	411
11.1.4.1. Stergerea primului nod al unei liste simplu înlățuite	411
11.1.4.2. Sterge un nod precizat printr-o cheie, dintr-o listă simplu înlățuită	412
11.1.4.3. Stergerea ultimului nod dintr-o lista simplu înlățuită	413
11.1.5. Stergerea unei liste simplu înlățuite	414
Exerciții	415
11.2. Stive și cozi	421
Exerciții	425
11.3. Listă circulară simplu înlățuită	432
11.3.1. Crearea unei liste circulare	434
11.3.2. Accesul la un nod al unei liste circulare	436
11.3.3. Inserarea unui nod intr-o listă circulară	437

11.3.3.1.	Inserarea unui nod înaintea unuia precizat printr-o cheie de tip int	437
11.3.3.2.	Inserarea unui nod după un nod precizat printr-o cheie de tip int	438
11.3.4.	Stergerea unui nod dintr-o listă circulară	438
11.3.5.	Stergerea unei liste circulare	439
	Exerciții	439
11.4.	Lista dublu înlanțuită	444
11.4.1.	Crearea unei liste dublu înlanțuite	446
11.4.2.	Inserarea unui nod într-o listă dublu înlanțuită	446
11.4.2.1.	Inserarea unui nod într-o listă dublu înlanțuită	447
11.4.2.2.	Inserarea unui nod într-o listă dublu înlanțuită	447
11.4.2.3.	Inserarea unui nod într-o listă dublu înlanțuită după unul precizat printr-o cheie	448
11.4.2.4.	Inserarea unui nod într-o listă dublu înlanțuită după ultimul nod (adaugarea unui nod la o listă dublu înlanțuită)	449
11.4.3.	Stergerea unui nod dintr-o listă dublu înlanțuită	450
11.4.3.1.	Stergerea primului nod al unei liste dublu înlanțuite	450
11.4.3.2.	Stergerea unui nod dintr-o listă dublu înlanțuită precizat printr-o cheie	451
11.4.3.3.	Stergerea ultimului nod al unei liste dublu înlanțuite	451
	Exerciții	453
12.	ARBORI	458
12.1.	Inserarea unui nod frunză într-un arbore binar	466
12.2.	Accesul la un nod al unui arbore binar	469
12.3.	Parcursarea unui arbore binar	469
12.3.1.	Parcursarea arborilor binari în preordine	474
12.3.2.	Parcursarea arborilor binari în inordine	475
12.3.3.	Parcursarea arborilor binari în postordine	475
12.4.	Stergerea unui arbore binar	476
	Exerciții	476
13.	TABELE	486
13.1.	Tabela de cuvinte rezervate	487
13.2.	Tabela de dispersie	489
	Exerciții	494
14.	SORTARE	502
14.1.	Sortare Shell	503
14.2.	Sortare rapidă	507
	Exerciții	520

15.	DIN NOU DESPRE PREPROCESARE ÎN C	526
15.1.	Definiții și apeluri de macrouri	526
15.2.	Compilare condiționată	531
	Exerciții	534
16.	INTRĂRI/IEȘIRI	537
16.1.	Nivelul inferior de prelucrare a fișierelor	538
16.1.1.	Deschiderea unui fișier	538
16.1.2.	Citirea dintr-un fișier	541
16.1.3.	Scrierea într-un fișier	542
16.1.4.	Pozitionarea într-un fișier	542
16.1.5.	Închiderea unui fișier	543
	Exerciții	544
16.2.	Nivelul superior de prelucrare a fișierelor	548
16.2.1.	Deschiderea unui fișier	548
16.2.2.	Prelucrarea pe caractere a unui fișier	549
	Exerciții	550
16.2.3.	Închiderea unui fișier	550
	Exerciții	551
16.2.4.	Intrări/ieșiri de siruri de caractere	551
16.2.5.	Intrări/ieșiri cu format	552
16.2.6.	Vidarea zonei tampon a unui fișier	553
16.2.7.	Pozitionarea într-un fișier	554
16.2.8.	Prelucrarea fișierelor binare	554
16.3.	Stergerea unui fișier	555
16.4.	Redirectarea fișierelor de intrare/ieșire standard	556
	Exerciții	557
17.	FUNCTII STANDARD	560
17.1.	Macrouri de clasificare	562
17.2.	Macrouri de transformare a caracterelor	563
17.3.	Funcții de conversie	563
17.4.	Funcții de calcul	564
17.5.	Funcții pentru gestiunea memoriei	566
17.6.	Funcții de control ale proceselor	566
17.7.	Funcții pentru gestiunea datei și a orei	567
17.8.	Diferite funcții de uz general	568
17.9.	Tratarea erorilor	570
	Exerciții	570
18.	GESTIUNEA ECRANULUI ÎN MOD TEXT	577
18.1.	Setarea ecranului în mod text	578

18.2.	Definirea unei ferestre	579
18.3.	Stergerea unei ferestre	579
18.4.	Gestiunea cursorului	580
18.5.	Determinarea parametrilor ecranului	581
18.6.	Modurile video alb/negru	581
- 18.7.	Setarea culorilor	582
	Exerciții	582
18.8.	Gestiunea textelor	583
	Exerciții	585
19.	GESTIUNEA ECRANULUI ÎN MOD GRAFIC	598
19.1.	Setarea modului grafic	598
	Exerciții	601
19.2.	Gestiunea culorilor	602
	Exerciții	606
19.3.	Starea ecranului	606
	Exerciții	607
19.4.	Gestiunea textelor	608
	Exerciții	610
19.5.	Gestiunea imaginilor	612
	Exerciții	616
19.6.	Tratarea erorilor	621
	Exerciții	621
19.7.	Desenare și colorare	625
	Exerciții	630
	BIBLIOGRAFIE	639

10. STRUCTURI ȘI TIPURI DEFINITE DE UTILIZATOR

Limbajele de programare oferă utilizatorului facilități de prelucrare atât a datelor *singulare* (izolate), cit și a celor *grupate*.

Datele se grupează pentru a forma multimi de elemente care să poată fi prelucrate atât element cu element, cit și global. De obicei, aceste grupe sunt multimi ordonate de date, adică datele unei astfel de grupe satisfac anumite relații.

Un mod simplu de grupare a datelor ne conduce la noțiunea de *tablou*.

Tabloul este o mulțime ordonată de date de un *același tip*, relația de ordine între elementele sale fiind definită cu ajutorul indicilor. *Numărul* indicilor definește *dimensiunea* tabloului. Tipul comun elementelor tabloului este și *tipul tabloului*.

Adesea este util să grupăm date într-un alt mod decit cel utilizat în cazul tablourilor. Astfel, se pot grupa date care nu neapărat sunt de același tip. Gruparea se face cu scopul de a prelucra datele respective nu numai separat, individual ci și global. Grupa și în acest caz este o mulțime ordonată. Referirea la elementele ei nu se mai face însă folosind indicii, ci construcții de felul numelui. Componentele unei grupe pot fi ele însele grupe de alte date și aşa mai departe. În felul acesta, datele sunt grupate pe *niveluri*. Grupa care nu este componentă a unei alte grupe se spune că se află la nivelul *cel mai înalt*. Datele care nu mai sunt grupe de alte date se numesc *date elementare*. Ele se află la nivelurile cele *mai inferioare*. În felul acesta datele sunt grupate potrivit unei ierarhii. Datele grupate, conform unei ierarhii, se numesc *structuri*.

Un exemplu simplu de structură este *data calendaristică*. Ea este o grupă de 3 date elementare: *zi*, *lună* și *an*.

Componentele *zi* și *an* sunt date de tip intreg. Componenta *lună* poate fi un sir de caractere, care să reprezinte luna prin denumirea ei: ianuarie, februarie etc. De aceea, componenta *lună* este un tablou de tip caracter care permite memorarea denumirii unei luni calendaristice.

Observație:

Datele elementare ale unei structuri pot fi izolate sau tablouri.

Dacă în cazul tablourilor, tipul lui este tipul comun elementelor sale, în cazul structurilor fiecare structură reprezintă un *tip nou* de date. Un astfel de tip se introduce printr-o *declarație de structură* și se spune că este un *tip definit de utilizator*.

În capitolul de față vom trece în revistă aspectele de bază cu privire la definirea și folosirea structurilor și a construcțiilor înrudite cu ele (reuniuni, cimpuri de biți etc.).

10.1. Declarația de structură

Pentru a declara o structura se pot folosi mai multe formate. Un format utilizat frecvent este:

```
struct nume {  
    (1)    lista_de_declaratii  
} nume1,nume2,...,numen;
```

unde:

nume, nume1, ..., numen - Sunt *nume* care pot și lipsi, dar nu toate deodată.
nume2,...,numen

Dacă *nume* este absent, atunci cel puțin *nume1* trebuie să fie prezent.

Dacă lista *nume1,nume2,...,numen* este absentă, atunci trebuie ca *nume* sa fie prezent.

Dacă *nume* este prezent, atunci el definește un *tip nou*, introdus prin declarația de structură respectivă. El poate fi utilizat în continuare pentru a defini date de acest tip, în mod asemanător cum definim date de tipuri predefinite.

În declarația de mai sus, *nume1,nume2,...,numen* sunt structuri de tipul *nume*.

În declarația de mai sus *numei* ($i=1,2,\dots,n$) se poate înlocui cu:

numei [*lim1*] [*lim2*]...[*limk*]

și atunci *numei* este un tablou k-dimensional de elemente de tip *nume*.

Prin *lista_de_declaratii* înțelegem una sau mai multe declarații prin care se declară componentele structurii de tip *nume*.

O structură de tip *nume* poate fi declarată și ulterior, folosind formatul:

```
(2) struct nume nume_de_structura;
```

unde:

nume_de_structura - Este un *nume*.

Accastă declarație scămana cu o declarație obișnuită în care se folosesc tipuri predefinite:

tip nume_data_izolata;

Deci, *tip* dintr-o declarație obișnuită se înlocuiește cu *struct nume*, unde *nume* este definit în prealabil într-o declarație de structură de formă (1), în care *nume1,nume2,...,numen* pot și lipsi.

Exemple:

1. Declaram *data_nasterii* și *data_angajarii* ca structuri de tip *data_calendaristica*, care este compusă din cele trei date amintite mai sus: *zi,luna* și *an*:

```
struct data_calendaristica {  
    int zi;  
    char luna[11];  
    int an;
```

```
} data_nasterii, data_angajarii;
```

Declarația de față este de forma (1).

În cazul în care nu dorim să introducем tipul *data_calendaristica*, se poate utiliza declarația:

```
struct {  
    int zi;  
    char luna[11];  
    int an;  
} data_nasterii, data_angajarii;
```

În sfîrșit, este posibil să definim tipul utilizator *data_calendaristica* și apoi să declarăm datele *data_nasterii* și *data_angajarii* folosind formatul (2).

```
struct data_calendaristica {  
    int zi;  
    char luna[11];  
    int an;  
};  
  
struct data_calendaristica data_nasterii,  
    data_angajarii;
```

Prima declarație introduce tipul utilizator *data_calendaristica*.

Declarația a două definește datele *data_nasterii* și *data_angajarii* ca fiind structuri de tipul *data_calendaristica*.

2. Fie tipul *data_calendaristica* definit mai sus. Declarația:

```
struct data_calendaristica d[100];
```

definește pe *d* ca un tablou de 100 de elemente, fiecare element fiind de tipul *data_calendaristica*.

3. Considerăm un model simplificat de date personale:

- *nume_prenume*;
- *adresa*;
- *localitatea_nasterii*;
- *cod*;
- *data_nasterii*;
- *data_angajarii*;
- *sex*;
- *stare_civila*;
- *numar_copii*.

Prin declarația de mai jos introducem tipul *date_pers*:

```
struct date_pers {  
    char nume_prenume[100];  
    char adresa[100];  
    char localitatea_nasterii[100];  
    long cod;
```

```

struct data_calendaristica
{
    data_nasterii,
    data_angajarii;
    char sex;
    char starea_civila[15];
    int numar_copii;
};

}

```

La definirea tipului *date_pers* s-a utilizat tipul *data_calendaristica*. Acest lucru este posibil dacă tipul *data_calendaristica* este în prealabil definit. Fie declarația:

```
struct date_pers unangajat,nangajati[1000];
```

Prin această declarație se declară structura *unangajat* și tabloul de structuri *nangajati*. Ambele au tipul *date_pers*.

4. În acest exemplu se introduce tipul *complex*:

```

struct complex {
    double real;
    double imag;
};

```

Datele declarate cu ajutorul declarației:

```
struct complex z,tz[100];
```

sunt date de tip *complex*. Atât partea reală, cât și partea imaginară sunt date de tip *double*.

5. Fie declarația:

```

struct dmatp2 {
    double matrice[2][2];
} ;
struct dmatp2 a,b;

```

Variabilele *a* și *b* sunt fiecare cîte o matrice pătratică de ordinul 2 ale cărei elemente sunt numere de tip *double*.

6. Fie declarația:

```

struct elev {
    char *nume;
    char *prenume;
    long nr_matricol;
    struct data_calendaristica data_nasterii;
    char *adresa;
    char *locul_nasterii;
    int clasa;
    int note[15];
};

```

Prin această declarație s-a introdus tipul *elev*. Declarația următoare definește date de tipul *elev*:

```
struct elev unelev,claselevi[25];
```

7. Ecranul pe care se afișeză datele poate fi gestionat în mod *text* sau în mod *grafic*. În mod implicit, ecranul se gestionează în mod *text*. În acest mod, ecranul se compune, de obicei, din 25 linii a 80 de coloane.

Pozitia unui punct pe ecran se definește prin 2 valori (coordonate):

x - numărul coloanei

și

y - numărul liniei.

Colțul din stînga sus are coordonatele:

x = 1;
y = 1.

Colțul din dreapta jos are coordonatele:

x = 80;
y = 25.

Pentru a controla afișarea datelor pe ecran, se pot utiliza diferite funcții standard de gestiune a ecranului. O parte dintre aceste funcții folosesc coordonatele *x* și *y* de felul celor indicate mai sus.

Adesea este util să definim tipul *punct* printr-o declarație de felul următor:

```
struct punct {
    int x;
    int y;
};
```

Cu ajutorul acestui tip se pot defini poziții pe ecran:

```
struct punct poz;
```

Structurile sunt date care se pot aloca la fel ca și celelalte date (variabile simple sau tablouri). Astfel, o structură este *globală* dacă se declară în afara corpului oricărei funcții, este *automatică* dacă se declară în corpul unei funcții sau *statică* dacă declarația ei începe cu cuvintul cheie *static*.

Datele elementare care sunt componente ale unei structuri se pot inițializa. În acest scop, într-o declarație sau definiție de structură, după numele structurii, se scrie caracterul *=*, iar după acesta construcțiile prin care se vor inițializa componente elementare ale structurii, separate prin virgulă și incluse între acolade. Construcțiile utilizate la inițializare sunt expresii constante care corespund elementelor pe care le inițializează.

Exemple:

1. struct data_calendaristica
dc= { 1,"septembrie",1994};

unde:

data_calendaristica - Este tipul declarat în prealabil astfel:

```
struct data_calendaristica {  
    int zi;  
    char luna[11];  
    int an;  
};
```

2. Fie tipul *complex*:

```
struct complex {  
    double real;  
    double imag;  
};
```

Declarația de mai jos definește variabila complexă *z*, care inițial are valoarea:

$1 + 2i$

```
struct complex z = { 1.0, 2.0 };
```

Componentele elementare neinitializate ale unei structuri au valoarea inițială *zero* dacă structura este *globală* sau *statică*.

În cazul structurilor *automatic*, elementele neinitializate au o valoare inițială *nedefinită*.

Operatorul unar adresa se poate aplica numelui unei structuri. Ca rezultat se obține adresa primei sale componente elementare. De asemenea, se pot defini pointeri spre tipuri utilizator.

Exemplu:

```
struct complex {  
    double real;  
    double imag;  
};  
struct complex *p;  
struct complex z;  
...  
p = &z;
```

10.2. Accesul la elementele unei structuri

Fie tipul utilizator *data_calendaristica* definit în paragraful precedent:

```
struct data_calendaristica {  
    int zi;  
    char luna[11];  
    int an;  
};
```

și variabila *dc* de tip *data_calendaristica*:

```
struct data_calendaristica dc;
```

Data *dc* are 3 componente:

zi, *luna* și *an* - Sunt date elementare.

Se pune problema accesului (referirii) la componentele structurii *dc*.

Referirea la componentele unei structuri se face utilizând atât numele structurii cit și a componentei respective. Aceasta se realizează printr-o construcție de formă:

nume.numel

unde:

nume - Este numele structurii.

numel - Este numele componentei la care se dorește să se facă acces.

În exemplul de mai sus, vom avea acces la componenta *zi* a structurii *dc*, prin construcția:

dc.zi

În mod analog:

dc.an

ne asigură accesul la componenta *an* a structurii *dc*. Datele *dc.zi* și *dc.an* sunt de tip *int*.

Construcția *dc.luna* este un pointer spre caractere și permite accesul la componenta *luna* a structurii *dc*.

Punctul utilizat într-o construcție de formă:

(1)*nume.numel*

se consideră un *operator* și el are prioritate maximă, la fel ca și operatorii paranteză.

O construcție de formă (1) se spune că este un *nume calificat*.

Exemplu:

```
1. struct data_calendaristica {  
    int zi;  
    char luna[11];  
    int an;  
};  
struct data_calendaristica dc, d[10];  
...  
dc.zi = 1;  
dc.an = 1995;  
strcpy(dc.luna, "septembrie");  
...  
d[3].zi = dc.zi;  
d[3].an = dc.an;  
strcpy(d[3].luna, dc.luna);  
...
```

Primele 3 instrucțiuni, atribuie structurii *dc* data calendaristică:

1 septembrie 1995

Ultimele 3, atribuie elementului *d[3]* valorile componentelor structurii *dc*, deci elementul respectiv păstrează aceeași dată calendaristică.

```
2. struct data_calendaristica {
    int zi;
    char luna[11];
    int an;
};

struct date_pers {
    char nume[50];
    char prenume[50];
    long cod;
    struct data_calendaristica
        data_nasterii,
        data_angajarii;
};

struct date_pers angajat, sectie[100];
strcpy(angajat.nume, "Popescu");
strcpy(angajat.prenume, "Cheorghe");
angajat.cod = 123456789;
```

Pentru a defini data nașterii va trebui să folosim o dublă calificare:

```
angajat.data_nasterii.zi = 3;
strcpy(angajat.data_nasterii.luna, "august");
angajat.data_nasterii.an = 1970;
```

În mod analog se definește data angajării:

```
angajat.data_angajarii.zi = 15;
strcpy(angajat.data_angajarii.luna, "septembrie");
angajat.data_angajarii.an = 1993;
```

În limbajul C structurile nu se pot transfera prin parametri, ci numai pointerii spre ele. Așa de exemplu, structura *angajat* poate fi prelucrată de funcția *f*, dacă aceasta se apelează cu parametrul efectiv *&angajat* (adresa de început a zonei de memorie alocată structurii *angajat*):

```
f(&angajat);
```

Funcția *f* are ca parametru un pointer spre tipul structurii *angajat*, adică spre tipul: *date_pers*.

Rezultă că funcția *f* are antetul:

```
void f(struct date_pers *p)
```

Într-adevăr, declarația lui *p* specifică faptul că el are ca valoare adresa de început a unei zone de memorie în care se păstrează date de tip *date_pers*. Ori *&angajat* este chiar o astfel de adresă.

Problema care se pune în continuare este aceea a accesului la componentele structurii transferate prin adresa ei. În cazul de față se pune problema de a avea

acces, în corpul funcției *f*, la componentele structurii *angajat*: nume, prenume etc.

În corpul funcției *f* nu se pot utiliza numele calificate:

angajat.nume, *angajat.prenume*, *angajat.data_nasterii.zi* etc.

deoarece nu este cunoscut numele structurii *angajat*. În locul numelui structurii se cunoaște pointerul spre ea: *p*. De aceea, în construcțiile de mai sus nu avem decit să înlocuim numele *angajat* cu **p*. Se vor obține construcțiile:

```
(*p).nume;
(*p).prenume;
(*p).data_angajarii.zi
etc.
```

Pentru a simplifica scrierea construcțiilor de acest fel s-a introdus o notație cu sageata (\rightarrow minus urmat de caracterul mai mare) și anume:

*(*p)*.se înlocuiește cu *p* \rightarrow

În felul acesta, construcțiile de mai sus se scriu mai simplu:

```
p  $\rightarrow$  nume;
p  $\rightarrow$  prenume;
p  $\rightarrow$  data_angajarii.zi etc.
```

Săgeata, ca și punctul, se consideră un operator de prioritate maximă, la fel ca și parantezele. Amintim că acești operatori se asociază de la stanga spre dreapta.

În concluzie, accesul la componentele unei structuri se realizează fie printr-o construcție de forma:

nume.nume1

fie prin:

pointer \rightarrow *nume1*

unde:

- | | |
|----------------|-----------------------------------|
| <i>nume</i> | - Este numele structurii. |
| <i>nume1</i> | - Este numele componentei. |
| <i>pointer</i> | - Este un pointer spre structura. |

10.3. Asignări de nume pentru tipuri de date (declarații de tip)

Tipurile predefinite se identifică printr-un cuvint cheie. În cazul tipurilor utilizator în locul cuvintului cheie se utilizează construcția:

struct *nume*

unde:

nume - Este introdus printr-o declarație de forma:

```
struct nume {  
    ...  
};
```

În limbajul C se poate atribui un nume unui tip, indiferent că el este un tip predefinit sau unul utilizator.

În acest scop se utilizează o construcție de forma:

(1) `typedef tip nume_tip;`

unde:

tip - Este fie un tip predefinit, fie un tip utilizator.
nume_tip - Este numele care se atribuie tipului definit de *tip*.

După ce s-a atribuit un nume unui tip, numele respectiv poate fi utilizat pentru a declara date de acel tip, exact la fel cum se utilizează în declarații cuvintele cheie ale tipurilor predefinite: *int*, *char*, *float* etc.

De obicei, numele atribuit unui tip se scrie cu litere mari. Construcția (1) de mai sus o vom considera ca fiind o declarație prin care se atribuie un nume unui tip sau mai scurt *declarație de tip*.

Exemple:

1. Fie declarația:

`typedef int INTREG;`

După această declarație, cuvintul INTREG se poate utiliza pentru a defini date de tip *int*. Deci declarația:

`INTREG x;`

este identică cu declarația:
`int x;`

2. Fie declarația:

`typedef float REAL;`

Datele:

`REAL a,b;`

sunt de tip *float*.

3. `typedef struct data_calendaristica {`

`int zi;
 char luna[11];
 int an;
} DC;`

Declarația:

`DC data_nasterii,data_angajarii;`

este identică cu declarația:

`struct data_calendaristica data_nasterii, data_angajarii;`

4. `typedef struct {
 double real;
 double imag;
} COMPLEX;`

În continuare se pot declara numere complexe prin declarații de forma:
`COMPLEX z, tz[10];`

5. `typedef struct {
 int x;
 int y;
} PUNCT;
PUNCT a, b;`

Variabilele *a* și *b* sunt structuri de tip PUNCT. Fiecare are o componentă *x* (abscisa) și una *y* (ordonata).

6. Un dreptunghi se poate defini prin două virfuri diametral opuse ale sale. De obicei, se consideră virful din colțul stanga sus și cel din dreapta jos. Definim tipul utilizator DREPTUNGHI:

```
typedef struct {  
    PUNCT stanga_sus;  
    PUNCT dreapta_jos;  
} DREPTUNGHI;
```

În continuare se pot defini date de tip DREPTUNGHI:
`DREPTUNGHI d, td[10];`

7. `typedef int *PINT;`

PINT este numele atribuit pentru tipul pointer spre *int*.

Declarația:

`PINT p, t[100];`

este identică cu declarația:

`int *p, *t[100];`

Construcția:

*tip (*p) (lista de tipuri)*

se poate folosi în antetul unei funcții și cu ajutorul ei, parametrul *p* se declară ca fiind pointer spre o funcție care returnează o valoare de tipul *tip*, iar *lista de tipuri* definește tipurile parametrilor funcției spre care pointează *p*. La tipul parametrului *p* de mai sus se poate atribui un nume printr-o construcție *typedef* de forma:

(2) `typedef tip (*nume_tip)(lista tipurilor);`

Exemple:

1. Fie funcția *f* de antet:

`void f(int (*p)(float,char,long,int))`

Declarația:

`typedef int (*PF) (float,char,long,int);`

ne permite să rescriem mai simplu antetul funcției f:

```
void f(PF p)
```

2. Folosind declarația:

```
typedef double (* PDF) (double);
```

putem rescrie antetul funcției *itrapez* din exercițiul 8.25, în felul următor:

```
double itrapez(double a, double b, int n, PDF p)
```

Din cele de mai sus se observă că declarațiile *typedef* ne permit adesea să facem simplificări la scrierea antetelor și prototipurilor funcțiilor. Acestea devin substanțiale mai ales atunci cînd apar expresii complicate cu pointeri.

Sensul unei expresii cu pointeri se poate stabili dacă se exprimă prin cuvinte efectul fiecărui operator, ținînd seama de regulile de prioritate și asociativitate.

În lucrarea [5] se dau exemple de expresii complexe cu pointeri și se stabilesc sensurile lor înlocuind prin cuvinte efectele operatorilor pe care-i conțin.

Un astfel de exemplu indicat în lucrarea [5] este următoarea declarație:

```
int *(*(*x)[6]) () ;
```

Sensul lui x se determină astfel:

*int ** definește tipul pointer spre *int*, deci:

- a. *(*(*x)[6]) ()* - Este pointer spre *int*.
- b. *(*(*x)[6])* - Este o funcție și ținînd seama de afirmația de la punctul a., rezultă că această funcție returnează un pointer spre *int*.
- c. **(*x)[6]* - Are același sens cu cel de la punctul b.
- d. *(*x)[6]* - Pointer spre construcția definită precedent, deci pointer spre o funcție care returnează un pointer spre întregi.
- e. *(*x)* - Tablou de 6 elemente, fiecare element este un pointer spre o funcție care returnează un pointer spre întregi.
- f. **x* - Construcție identică cu cea de la punctul e.
- g. *x* - Este pointer spre un tablou de 6 elemente, fiecare element fiind un pointer spre o funcție care returnează un pointer spre *int*.

Un alt exemplu din aceeași lucrare este declarația:

```
char *(*y0 )[1] ) () ;
```

Această declarație se interpretează astfel:

- a. *(*(*y0)[1])()* - Definește o valoare de tip *char*.
- b. *(*(*y0)[1])* - Este o funcție care returnează o valoare de tip *char*.

- c. **(*y0)[1]* - Este aceeași construcție ca și cea de la punctul b.
- d. *(*y0)[1]* - Este un pointer spre o funcție care returnează o valoare de tip *char*.
- e. *(*y0)* - Este un tablou de pointeri spre funcții care returnează o valoare de tip *char*.
- f. **y0)* - Este aceeași construcție ca și cea de la punctul c.
- g. *y0)* - Este pointer spre un tablou de pointeri spre funcții care returnează o valoare de tip *char*.
- h. *y0* - Este o funcție care returnează un pointer spre un tablou de pointeri spre funcții care returnează o valoare de tip *char*.

Declarațiile de acest gen pot fi simplificate folosind declarații de tip.

De exemplu, în lucrarea [5] se declară tipuri intermedii pentru a simplifica declarația:

```
int *(*(*x)[6])();
```

analizată mai sus.

Exerciții:

10.1 Să se scrie o funcție care calculează și returnează modulul unui număr complex.

Dacă:

$$z = x + iy$$

atunci modulul numărului complex este rădăcina patrata din:

$$\sqrt{x^2 + y^2}$$

FUNCȚIA BX1

```
double dmodul (COMPLEX *z)
/* calculeaza și returneaza modulul numărului complex z */
{
    return sqrt(z->x * z->x + z->y * z->y);
}
```

10.2 Să se scrie o funcție care calculează și returnează argumentul unui număr complex.

Dacă:

$$\arg z = \theta$$

atunci

$$\arg z = \theta$$

se calculează astfel:

- a. Dacă $x = y = 0$, $\arg z = 0$.

- b. Dacă $y = 0$ și x este diferit de zero,
atunci
dacă $x > 0$, $\arg z = 0$;
altfel
 $\arg z = \pi = 3.14159265358979$.
- c. Dacă $x = 0$ și y este diferit de zero,
atunci
 dacă $y > 0$, $\arg z = \pi/2$;
 altfel
 $\arg z = 3\pi/2$.
- d. Dacă x și y sunt diferiți de zero,
atunci fie:
 $a = \arctg(y/x)$
- d1. Dacă $x > 0$ și $y > 0$, atunci $\arg z = a$.
d2. Dacă $x > 0$ și $y < 0$, atunci $\arg z = 2\pi + a$.
d3. Dacă $x < 0$ și $y > 0$, atunci $\arg z = \pi + a$.
d4. Dacă $x < 0$ și $y < 0$, atunci $\arg z = \pi + a$.
- Se observă că pentru $x < 0$ și y diferit de zero:
 $\arg z = \pi + a$;
 altfel, dacă $x > 0$ și $y < 0$, atunci
 $\arg z = 2\pi + a$

FUNCȚIA BX2

```
double darg(COMPLEX *z)
{
    double a;

    if( z->x == 0 && z->y == 0) return 0.0;
    if( z->y == 0)
        if( z->x > 0) return 0.0;
        else /* y=0 si x<0 */
            return PI;
    if(z->x == 0)
        if(z->y > 0) return PI/2;
        else /* x=0 si y<0 */
            return (3*PI)/2;

    /* x != 0 si y != 0 */
    a = atan(z->y/z->x);
    if(z->x < 0) /* x<0 si y != 0 */
        return a + PI;
    else /* x>0 */
        if( z->y < 0) /* x>0 si y<0 */
            return 2*PI + a;

    /* x>0 si y>0 */
    return a;
}
```

- 10.3 Sa se scrie un program care citește numere complexe și le afișează împreună cu modulul și argumentul lor.

PROGRAMUL BX3

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

#define PI 3.14159265358979

typedef struct {
    double x;
    double y;
} COMPLEX;

#include "bx1.cpp"
#include "bx2.cpp"

main() /* citește numere complexe și le afișează împreună cu modulul și argumentul corespunzător */
{
    COMPLEX complex;

    while(scanf("%lf %lf", &complex.x, &complex.y) == 2 ){
        printf("a+ib= %g + i*(%g)\n", complex.x, complex.y);
        printf("modul=%g\targ=%g\n",
               dmodul(&complex), darg(&complex));
    }
}
```

- 10.4 Să se scrie o funcție care afișează un caracter într-un punct de coordonate date.

Ecranul, în mod text, poate fi gestionat folosind funcții care au prototip în fișierul *conio.h*. Astfel, pentru a poziționa cursorul în punctul de coordonate (x,y), se va apela funcția *gotoxy*. Prototipul acestei funcții este:

```
void gotoxy (int x, int y);
```

Pentru a afișa, începînd cu punctul poziționat cu ajutorul funcției *gotoxy*, se pot folosi diferite funcții, ca de exemplu, *putch* sau *cprintf* în locul funcției *printf*. Funcția *cprintf* are aceeași parametri ca și funcția *printf*. Prototipul ei se află în fișierul *conio.h* spre deosebire de al funcției *printf* care se află în *stdio.h*.

O diferență a funcției *cprintf* față de funcția *printf* constă în aceea că '\n' se interprează diferit. Astfel, apelul:

```
cprintf("\n");
```

trece cursorul pe rindul următor lăsându-l în același coloană. De accea, apelul:

```
printf("\n");
```

se echivalează cu apelul:

```
cprintf("\n\r");
```

dacă cursorul nu este pe ultimul rind (25).

Funcția *cprintf* nu realizează defilarea ecranului și de aceea, caracterul '\n' se neglețează dacă cursorul se află pe ultimul rind.

FUNCȚIA BX4

```
void afiscar(PUNCT *p, char c)
/* - afiseaza caracterul c in punctul de coordonate (p->x,p->y);
   - se presupune ca punctul nu este in afara ecranului. */
{
    gotoxy( p->x, p->y );
    putch(c);
}
```

10.5 Să se scrie o funcție care verifică dacă un punct are coordonatele în limitele ecranului(25 de linii a 80 coloane).

Funcția returnează valoarea 1 dacă punctul se află în limitele ecranului și 0 în caz contrar.

FUNCȚIA BX5

```
int limecr(PUNCT *p)
/* returneaza 1 daca punctul de coordonate (p->x,p->y) se afla in limitele ecranului
   si zero altfel */
{
    if(p->x <= 0 || p->x > 80) return 0;
    if(p->y <= 0 || p->y > 25) return 0;
    return 1;
}
```

10.6 Să se scrie o funcție care trasează un segment de dreaptă orizontal afișind repetat un același caracter dat.

Se presupune că limitele segmentului se află pe ecran.

FUNCȚIA BX6

```
void segoriz(PUNCT *a,PUNCT *b,char c)
/* - traseaza un segment de dreaptă orizontal prin afisarea repetata a caracterului c;
   - se presupune ca punctul a precede pe b si ca a->y = b->y */
{
    int i;
    PUNCT crt;

    crt.y = a->y;
    for(i=a->x; i <= b->x; i++) {
        crt.x = i;
        afiscar(&crt,c);
    }
}
```

10.7 Să se scrie o funcție care trasează un segment de dreaptă vertical afișind

repetat un același caracter dat.

Se presupune că limitele segmentului se află pe ecran.

FUNCȚIA BX7

```
void segvert(PUNCT *a,PUNCT *b,char c)
/* - traseaza un segment de dreapta vertical prin afisarea repetata a caracterului c;
   - se presupune ca punctul a este deasupra punctului b si ca
     a->x = b->x */
{
    int i;
    PUNCT crt;

    crt.x = a->x;
    for(i = a->y; i <= b->y; i++) {
        crt.y = i;
        afiscar(&crt,c);
    }
}
```

10.8 Să se scrie o funcție care trasează un segment de dreaptă oblic de pantă 45 de grade sexagesimale afișind repetat un același caracter dat.

FUNCȚIA BX8

```
void segoblic(PUNCT *a,PUNCT *b,char c)
/* - traseaza un segment de dreapta oblic de pantă 45 grade sexagesimale
   prin afisarea repetata a caracterului c;
   - se presupune ca punctele a si b permit trasarea unui astfel de segment de dreapta si ca ele au coordonate ce aparțin ecranului. */
{
    int i,j,k,l;
    int pasi,pask;

    i = a->x;
    j = b->x;
    k = a->y;
    l = b->y;
    if( i < j ) /* a in stinga lui b */
        pasi = 1;
    else /* a in dreapta lui b */
        pasi = -1;
    if( k < l ) /* a deasupra lui b */
        pask = 1;
    else /* b deasupra lui a */
        pask = -1;

    /* trasarea segmentului */
    for( ; (j-i)*pasi >= 0; i +=pasi, k += pask) {
        gotoxy(i,k);
        putch(c);
    }
}
```

Observație:

Segmentul se traseaza de la punctul a spre punctul b . De exemplu, daca punctul a este deasupra lui b (pask = 1) și în dreapta lui b (pasí = -1), atunci se observă ca i descerște pînă cînd $i < j$. În acest moment produsul $(j-i)*pasí < 0$ și deci ciclul *for* se termină.

De fiecare dată cînd i s-a mișorat cu o unitate, k a crescut cu o unitate. De aceea, poziția caracterului curent față de cel precedent se află în coloana din stînga și cu o linie mai jos.

Punctele a și b trebuie să satisfacă relația:

$$(j-i)*pasí = (l-k)*pask$$

și de aceea, cînd $i=j$, vom avea și $k=l$, adică se ajunge în punctul b .

10.9 Să se scrie un program care trasează un dreptunghi.

Laturile orizontale se trasează folosind caracterul "-" (minus), iar cele verticale folosind caracterul "!" (bara verticală).

PROGRAMUL BX9

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

typedef struct {
    int x;
    int y;
} PUNCT;

#include "bx4.cpp" /* afisare */
#include "bx6.cpp" /* segoriz */
#include "bx7.cpp" /* segvert */
#include "bviii2.cpp" /* pcit int */
#include "bviii3.cpp" /* pcit int lim */

main() /* citeste coordonatele coltului stînga sus și a coltului dreapta jos ale unui
        dreptunghi după care trasează dreptunghiul respectiv folosind caracterul
        - pentru laturile orizontale și
        ! pentru cele verticale. */
{
    PUNCT v1,v2,v;
    char er[] = "s-a tastat EOF\n";
    /* citeste coordonatele celor două colturi ale dreptunghiului */
    /* colțul din stînga sus */
    if(pcit_int_lim("x_stanga_sus: ",1,79,&v1.x) == 0) {
        printf(er);
        exit(1);
    }
    if(pcit_int_lim("y_stanga_sus: ",1,24,&v1.y) == 0) {
        printf(er);
    }
}
```

```
        exit(1);
    }

    /* colțul din dreapta jos */
    if(pcit_int_lim("x_dreapta_jos: ",v1.x+1,80,&v2.x) == 0) {
        printf(er);
        exit(1);
    }
    if(pcit_int_lim("y_dreapta_jos: ",v1.y+1,25,&v2.y) == 0) {
        printf(er);
        exit(1);
    }

    /* sterge ecranul */
    clrscr();

    /* trasarea laturilor dreptunghiului */
    /* laturile orizontale */
    /* latura de sus */
    v.x = v2.x;
    v.y = v1.y;
    segoriz(&v1,&v,'-');

    /* latura de jos */
    v.x = v1.x;
    v.y = v2.y;
    segoriz(&v1,&v,'-');

    /* laturile verticale */
    v1.y++;
    v2.y--;
    if(v1.y > v2.y)
        exit(0); /* nu sunt laturi verticale */

    /* latura din stînga */
    v.x = v1.x;
    v.y = v2.y;
    segvert(&v1,&v,'|');

    /* latura din dreapta */
    v.x = v2.x;
    v.y = v1.y;
    segvert(&v1,&v,'|');
    getch();
}
```

10.10 Să se scrie un program care afîșează figura de mai jos.

```

*
* *
* * *
* * *
* * *
*****
* * * **
* * * * *
* * * * *
* * * * *
*****
* * * * *
* * * * *
* * * * *
* * * **
*****
* * * *
* * *
* * *
* * *
* *

```

Această figură se compune din segmente de dreaptă orizontale, verticale și oblice de pantă 45 de grade sexagesimale. De aceea, ea se poate afișa folosind funcțiile *segoriz*, *segvert* și *segoblic*.

Virful de sus are coordonatele A(40,1). Segmentul vertical din stînga are extremitățile B(35,6) și C(35,16).

Virful de jos are coordonatele D(40,21). Segmentul vertical din dreapta are extremitățile F(45,6) și E(45,16).

Segmentul orizontal de sus are extremitățile B și F, iar cel de jos extremitățile C și E.

Segmentul orizontal, care este axă de simetrie a figurii, are extremitățile G(35,11) și H(45,11).

În afară de aceste segmente, figura mai conține următoarele segmente oblice:

AB, AF, BE, FC, CD și ED.

PROGRAMUL BX10

```

#include <conio.h>

typedef struct {
    int x;
    int y;
} PUNCT;

#include "bx4.cpp" /* afisare */

```

```

#include "bx6.cpp" /* segoriz */
#include "bx7.cpp" /* segvert */
#include "bx8.cpp" /* segoblic */

#define C ' '

main () /* afiseaza o figura formata din segmente orizontale, verticale
           si oblice folosind caracterul * */
{
    static PUNCT a = {40,1};
    static PUNCT b = {35,6};
    static PUNCT c = {35,16};
    static PUNCT d = {40,21};
    static PUNCT e = {45,16};
    static PUNCT f = {45,6};
    static PUNCT g = {35,11};
    static PUNCT h = {45,11};

    /* sterge ecranul */
    clrscr();

    /* afiseaza segmentele verticale */
    segvert(&b,&c,C);
    segvert(&a,&d,C);
    segvert(&f,&e,C);

    /* afiseaza segmentele orizontale */
    segoriz(&b,&f,C);
    segoriz(&c,&e,C);
    segoriz(&g,&h,C);

    /* afiseaza segmentele oblice */
    segoblic(&a,&b,C);
    segoblic(&a,&f,C);
    segoblic(&b,&e,C);
    segoblic(&f,&c,C);
    segoblic(&c,&d,C);
    segoblic(&e,&d,C);
    getch();
}

```

10.11 Să se scrie un program care realizează următoarele:

- citește datele de identificare și notele obținute de o grupă de candidați la admiterea în invățămîntul superior;
- listează datele citite împreună cu media notelor obținute de fiecare candidat, media grupei la fiecare materie și media mediilor.

La inceputul programului se citește un întreg ce reprezintă numărul examenelor susținute, apoi se citesc denumirile materiilor la care s-au susținut examene. În continuare, pentru fiecare candidat, se citesc:

- nume;
- prenume;

- data nașterii;
- adresa;
- nota obținuta la fiecare examen.

Dacă un candidat are mai multe prenume, acestea se concatenază prin liniuță de unire, nu se vor separa prin spații. În felul acesta, chiar dacă sunt mai multe prenume, ele se pot citi prin `scanf`, ca și cum ar fi un singur prenume.

Data nașterii se tastează printr-un sir de 6 cifre:

zlllaa

deci, primele două cifre reprezintă ziua, urmatoarele două luna, iar ultimele două anul.

Adresa este o succesiune de caractere care nu conține caractere albe. Spațiile din cadrul unei adrese se vor înlocui prin caracterul subliniere. În felul acesta, adresa poate fi citită ca un singur element.

Programul utilizează urmatoarele funcții definite în exercițiile precedente:

<code>pcit_int_lim</code>	- Pentru a citi date de tip <code>int</code> care se află într-un interval precizat (exercițiul 8.3).
<code>pcit_data_calend</code>	- Pentru a citi și valida o dată calendaristică (exercițiul 8.4).
<code>pcit_int</code>	- Funcție apelată de <code>pcit_int_lim</code> (exercițiul 8.2).
<code>v_calend</code>	- Funcție apelată de <code>pcit_data_calend</code> (exercițiul 6.5).

Programul mai apelează funcția `pcit_sir` definită în acest exercițiu, care realizează următoarele:

- afișează un text explicativ;
- citește o succesiune de caractere;
- păstrează succesiunea citită în memoria `heap`;
- returnează pointerul spre începutul zonei din memoria `heap` în care se păstrează caracterele respective.

Programul se execută conform pașilor de mai jos:

1. Citește pe `n`, numărul examenelor (se apeleză `pcit_int_lim`).
2. Citește denumirile celor `n` materii care se examinează (se apeleză `pcit_sir` de `n` ori).
3. $i = 0$ (numărător pentru candidați).
4. Se citesc datele de identificare ale unui candidat și notele acestuia.
Se păstrează datele citite, apoi se trece la pasul 5.
Citirile se realizează apelând funcțiile indicate mai sus.
Dacă nu mai sunt date de citit, se trece la punctul 7.
5. $i = i + 1$.
6. Se reia de la punctul 4.

7. Se fac inițializări pentru calcule de medii pe materii și a mediei generale.
8. Se afișează antetul listei.
9. Pentru fiecare candidat se afișează:
 - lista cu datele de identificare a candidatului, notele obținute la fiecare materie (nota 0 se consideră neprezentare la examen) și media notelor respective;
 - după ce s-au listat toate datele pentru toți candidații, se trece la punctul 10.
10. Se afișează mediile pe materii;
11. Se afișează media generală.

În programul de față se presupune că sunt cel mult 5 examene.

Datele relative la un candidat se păstrează în memoria `heap`. De asemenea, tot în memoria `heap` se păstrează denumirile materiilor de examinat.

PROGRAMUL BX11

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <string.h>

#include "bviii2.cpp" /* pcit_int */
#include "bviii3.cpp" /* pcit_int_lim */
#include "bvi5.cpp" /* v_calend */
#include "bviii4.cpp" /* pcit_data_calend */

#define MAX 100

int nrzile[] = {0,31,28,31,30,31,30,31,31,30,31,30,31};

int pcit_sir(char *, char **);

main() /* listează situația notelor unui grup de candidați obținute la
         admitera în învățământul superior */
{
    typedef struct {
        int zi;
        int luna;
        int an;
    } DC;

    typedef struct {
        char *nume;
        char *prenume;
        DC data_nasterii;
        char *adresa;
        int nrex;
        int nota[5];
    } CAND;

    CAND *tcand[MAX];
    char *denex[5];
}
```

```

int nex;
char er[]="s-a citit EOF\n";
int i,j,k;
char *temp;
float tmed[5];
float medgen;
float med;

/* citeste numarul examenelor */
if(pcit_int_lim("numarul examenelor:1-5: ",1,5,&nex) == 0 ){
    printf(er);
    exit(1);
}

/* citeste denumirile materiilor si le pastreaza in memoria heap */
for(i=0; i<nex; i++) {
    if(pcit_sir("denumire materie: ",&denex[i]) == 0 ) {
        printf(er);
        exit(1);
    }
}

/* se citesc datele de identificare ale elevilor si notele */
i = 0;
while( i < MAX ) { /* i */
    printf("se citesc datele elevului nr: %d\n",i+1);

    /* citeste numele candidatului */
    if(pcit_sir("nume: ",&temp) == 0 )
        break; /* nu mai sunt alti candidati */
    /* rezerva zona pentru o data de tip CAND */
    if((tcand[i]=(CAND *)malloc(sizeof(CAND))) == 0 ) {
        printf("memorie insuficienta\n");
        exit(1);
    }

    /* tcand[i] are ca valoare adresa de inceput a zonei de memorie din
       memoria heap in care se pastreaza o data structurata de tip CAND */

    /* se pastreaza pointerul spre numele candidatului curent citit, nume
       pastrat de functia pcit_sir in memoria heap */
    tcand[i] -> nume = temp;

    /* se citeste prenumele */
    if(pcit_sir("prenume: ",&tcand[i] -> prenume)== 0 ) {
        printf(er);
        exit(1);
    }

    /* citeste data nasterii */
    if(pcit_data_calend(
        &tcand[i] -> data_nasterii.zi,
        &tcand[i] -> data_nasterii.luna,
        &tcand[i] -> data_nasterii.an) == 0) {
        printf(er);
    }
}

exit(1);
}

/* citeste adresa */
if(pcit_sir("adresa: ",&tcand[i] -> adresa)== 0 ) {
    printf(er);
    exit(1);
}
tcand[i] -> nxex = nex; /* pastreaza numarul examenelor */

/* se citesc notele; se tasteaza in ordinea citirii denumirilor materiilor de examinat */
for(j=0; j < nex; j++)
    if(pcit_int_lim(denex[j],0,10,
        &tcand[i] -> nota[j])==0 ) {
        printf(er);
        exit(1);
    }

/* s-au citit toate datele pentru un candidat */
i++;
} /* sfarsit while */

/* initializari pentru calculul mediilor pe materii */
for(j=0; j < nex; j++) tmed[j] = 0.0;
medgen = 0.0;

/* afiseaza antetul */
printf("\n\n\nNotele de la examenul de admitere\
      din 7 septembrie 1994\n");
printf("\n\n\n");
printf("%20s", " "); /* 20 spatii pentru nume prenume */

/* se scriu denumirile examenelor */
for(j=0; j<nex; j++) {
    printf("%10s",denex[j]);
    printf(" "); /* cel putin un spatiu dupa fiecare denumire */
}
printf("Media\n\n");

/* lista cu datele de identificare, notele si media pentru fiecare candidat */
for(j=0; j<i; j++) {
    /* datele de identificare */
    printf("%25s %25s %02d/%02d/%02d\n",
           tcand[j] -> nume,
           tcand[j] -> prenume,
           tcand[j] -> data_nasterii.zi,
           tcand[j] -> data_nasterii.luna,
           tcand[j] -> data_nasterii.an);

    /* adresa */
    printf("\t%8s\n",tcand[j] -> adresa);
    /* note */
    printf("%20s", " ");
    for(k=0; med = 0.0; k < nex; k++) {

```

```

        printf("%10d ", tcand[j] -> nota[k]);
        med += tcand[j] -> nota[k];
        tmed[k] += tcand[j] -> nota[k];
    }
    printf("%6.2f\n", med/nex);
} /* sfirsit date candidati */

/* afisare media pe materii */
printf("\n\n Media pe materii ");
for(j=0;j<nex;j++) {
    printf("%11.2f", tmed[j]/i);
    medgen += tmed[j] / i;
}

/* afisare media generala */
printf("\n\n Media generala: %10.2f\n", medgen / nex);
} /* sfirsit main */

int pocit_sir( char *text, char **sir)
/* - afiseaza text si citeste o succesiune de caractere;
   - pastreaza succesiunea citita in memoria heap si atribuie adresa acestei zone, pointerului sir;
   - returneaza:
     0 - la sfirsit de fisier
     1 - in caz contrar
   - intrerupe executia programului daca nu exista memorie heap suficienta pentru a pastra
     sirul de caractere citit. */
{
    char t[255];
    char *p;

    printf(text);
    if(gets(t) == 0 ) return 0;

    /* rezerva zona in memoria hcap */
    if((p = (char *)malloc(strlen(t) + 1 ))== 0 ) {
        printf("memorie insuficienta\n");
        exit(1);
    }

    /* transfera sirul din t la adresa continuta in p */
    strcpy(p,t);
    *sir = p;
    return 1;
}

```

10.12 Să se scrie o funcție care schimbă semnul la partea reală și cea imaginată a unui număr complex.

Funcția utilizează tipul COMPLEX introdus prin declarația:

```

typedef struct {
    double x;
    double y;
} COMPLEX;

```

Acst tip se utilizează în exercițiile care urmează în acest capitol.

FUNCȚIA BX12

```

void negcomplex(COMPLEX *a,COMPLEX *b) /* b = -a */
{
    b -> x = - a -> x;
    b -> y = - a -> y;
}

```

10.13 Să se scrie o funcție care atribuie partea reală și imaginată a unui număr complex la partea reală și imaginată a unui alt număr complex.

FUNCȚIA BX13

```

void atribcomplex(COMPLEX *a,COMPLEX *b) /* b = a */
{
    b -> x = a -> x;
    b -> y = a -> y;
}

```

10.14 Să se scrie o funcție care adună două numere complexe.

FUNCȚIA BX14

```

void adcomplex(COMPLEX *a, COMPLEX *b,COMPLEX *c)
/* c = a+b */
{
    c -> x = a -> x + b -> x;
    c -> y = a -> y + b -> y;
}

```

10.15 Să se scrie o funcție care scade două numere complexe.

FUNCȚIA BX15

```

void sccomplex(COMPLEX *a,COMPLEX *b,COMPLEX *c)
/* c = a-b */
{
    c -> x = a -> x - b -> x;
    c -> y = a -> y - b -> y;
}

```

10.16 Să se scrie o funcție care înmulțește două numere complexe.

Dacă a și b sunt două numere complexe de forma:

$$a = ax + i*ay$$

și

$$b = bx + i*by$$

atunci produsul lor se definește astfel:

$$a*b = (ax + i*ay)*(bx + i*by) = ax*bx - ay*by + (ax*by + bx*ay)*i$$

FUNCȚIA BX16

```
void mulcomplex(COMPLEX *a, COMPLEX *b, COMPLEX *c)
/* c = a*b */
{
    c->x = a->x * b->x - a->y * b->y;
    c->y = a->x * b->y + b->x * a->y;
}
```

- 10.17 Să se scrie o funcție care împarte două numere complexe. Ea returnează valoarea zero dacă împărțitorul este nul și unu în caz contrar.

Dacă a și b sunt două numere complexe de forma:

$$a = ax + i*ay$$

și

$$b = bx + i*by$$

atunci:

$$\begin{aligned} c = a/b &= (ax + i*ay)/(bx + i*by) \\ &= [(ax + i*ay)(bx - i*by)]/(bx^2 + by^2) \\ &= (ax * bx + ay * by)/(bx^2 + by^2) + \\ &\quad i * (ay * bx - ax * by)/(bx^2 + by^2) \end{aligned}$$

FUNCȚIA BX17

```
int divcomplex(COMPLEX *a, COMPLEX *b, COMPLEX *c)
/* c = a/b */
{
    double d;

    d = b->x * b->x + b->y * b->y;
    if( d == 0 ) return 0;
    c->x = (a->x * b->x + a->y * b->y)/d;
    c->y = (a->y * b->x - a->x * b->y)/d;
    return 1;
}
```

- 10.18 Să se scrie o funcție care afișează un text și citește un număr de tip *DOUBLE*.

Funcția returnează zero dacă se intilnește sfîrșitul de fișier și unu în caz contrar.

Această funcție este similară cu funcția *pcit_int* definită în exercițiul 8.2.

FUNCȚIA BX18

```
int pcit_double(char text[], double *x)
/* - afișaza caracterele pastrate în tabloul text, citește un număr de tip
   double și-l pastrează în zona de memorie spre care pointează x;
   - returnează:
     0 - la intilnirea sfîrșitului de fișier;
     1 - altfel. */
{
```

```
char t[255];

for ( ; ; ) {
    printf(text);
    if(gets(t) == 0) return 0;
    if(sscanf(t,"%lf",x) == 1) return 1;
    printf("nu s-a tastat un număr\n");
}
```

- 10.19 Să se scrie o funcție care citește partea reală și partea imaginara a unui număr complex.

Funcția returnează zero la intilnirea sfîrșitului de fișier și unu în caz contrar.

FUNCȚIA BX19

```
int pcitcomplex(COMPLEX *a)
/* - citește partea reală și cea imaginara a numărului complex a;
   - returnează:
     0 - la intilnirea sfîrșitului de fișier;
     1 - altfel. */
{
    if(pcit_double("partea reală: ", &a->x)== 0) return 0;
    if(pcit_double("partea imaginara: ",&a->y)== 0) return 0;
    return 1;
}
```

- 10.20 Să se scrie un program care citește trei numere complexe a,b,c care sunt coeficienții ecuației:

$$a*x^2 + b*x + c = 0$$

rezolvă și afișează rădăcinile ecuației respective.

PROGRAMUL BX20

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

#define PI 3.14159265358979

typedef struct {
    double x;
    double y;
} COMPLEX;
```

```
#include "bx12.cpp" /* negcomplex */
#include "bx14.cpp" /* adcomplex */
#include "bx15.cpp" /* sccomplex */
#include "bx16.cpp" /* mulcomplex */
#include "bx17.cpp" /* divcomplex */
#include "bx1.cpp" /* dmodul */
#include "bx2.cpp" /* darg */
#include "bx18.cpp" /* pcit_double */
```

```

#include "bx19.cpp" /* pcitcomplex */

main() /* citeste pe a,b,c - numere complexe;
           - rezolva si afiseaza radacinile ecuatiei:
             a*x*x + b*x + c = 0. */
{
    COMPLEX a,b,c,x1,x2,bp,temp,aa,temp1;
    char er[] = "s-a tastat EOF\n";
    double r,arg;
    COMPLEX patru = {4.0,0.0};

    /* se citesc coeficientii */
    if(pcitcomplex(&a) == 0) {
        printf(er);
        exit(1);
    }
    if(pcitcomplex(&b) == 0) {
        printf(er);
        exit(1);
    }
    if(pcitcomplex(&c) == 0) {
        printf(er);
        exit(1);
    }
    if( a.x == 0 && a.y == 0 && b.x == 0 &&
        b.y == 0 && c.x == 0 && c.y == 0 ) {
        /* a=b=c=0 */
        printf("ecuatie nedeterminata\n");
        exit(0);
    }
    if( a.x == 0 && a.y == 0 && b.x == 0 && b.y == 0 ) {
        /* a=b=0 si c!=0 */
        printf("ecuatie nu are solutie\n");
        exit(1);
    }
    if(a.x == 0 && a.y == 0) {
        /* ecuatie de gradul 1 */
        negcomplex(&c,&x1);
        divcomplex(&x1,&b,&x2);
        printf("ecuatie de gradul 1\n");
        printf("x=%g + i*(%g)\n", x2.x, x2.y);
        exit(0);
    } /* temp = b*b - 4*a*c */

    mulcomplex(&b,&b,&bp);
    mulcomplex(&patru,&a,&temp);
    mulcomplex(&temp,&c,&temp1);
    sccomplex(&bp,&temp1,&temp);

    /* temp = x + i*y
       r = modul(temp)
       arg = argument(temp) */
    r = dmodul(&temp);

```

```

    arg = darg(&temp);

    /* sqrt(temp); temp = r*(cos(arg) + i*sin(arg));
       sqrt(temp)=sqrt(r)*(cos(arg/2)+i*sin(arg/2)) */
    r = sqrt(r);
    arg = arg/2;

    /* sqrt(b*b - 4*a*c) = r*(cos(arg)+i*sin(arg)) */
    temp.x = r*cos(arg);
    temp.y = r*sin(arg);

    /* aa = 2*a = a+a */
    adcomplex(&a,&a,&aa);

    /* x1=(-b+sqrt(b*b-4*a*c))/(2*a) */
    /* bp=b */
    negcomplex(&b,&bp);
    adcomplex(&bp,&temp,&temp1);
    divcomplex(&temp1,&aa,&x1);

    /* x2=(-b-sqrt(b*b-4*a*c))/(2*a) */
    sccomplex(&bp,&temp,&temp1);
    divcomplex(&temp1,&aa,&x2);
    printf("x1=%g+i(%g)\n", x1.x,x1.y);
    printf("x2=%g+i(%g)\n", x2.x,x2.y);
}

```

10.21 Să se scrie o funcție care adună două matrice pătratice de ordinul trei. Elementele matricelor sunt numere de tip *double*.

Funcția de față și cele care urmează în acest capitol vor folosi tipul MATP3 declarat astfel:

```

typedef struct {
    double x11,x12,x13,x21,x22,x23,x31,x32,x33;
} MATP3;

```

FUNCTIA BX21

```

void admatp(MATP3 *a, MATP3 *b,MATP3 *c) /* c = a + b */
{
    c->x11 = a->x11 + b->x11;
    c->x12 = a->x12 + b->x12;
    c->x13 = a->x13 + b->x13;
    c->x21 = a->x21 + b->x21;
    c->x22 = a->x22 + b->x22;
    c->x23 = a->x23 + b->x23;
    c->x31 = a->x31 + b->x31;
    c->x32 = a->x32 + b->x32;
    c->x33 = a->x33 + b->x33;
}

```

10.22 Să se scrie o funcție care înmulțește la stanga o matrice pătratică de ordinul trei cu un vector de trei elemente.

Dacă matricea are elementele:

```
x11 x12 x13  
x21 x22 x23  
x31 x32 x33
```

iar vectorul este b de componente:

```
b[0],b[1],b[2]
```

atunci produsul dintre matrice și vectorul b este un vector c de trei elemente:

```
c[i-1] = xi1*b[0] + xi2*b[1] + xi3*b[2] pentru i = 1,2,3.
```

FUNCȚIA BX22

```
void mulmatvect3(MATP3 *a, double b[],double c[])  
/* c = a*b */  
{  
    c[0] = a->x11 * b[0] + a->x12 * b[1] + a->x13 * b[2];  
    c[1] = a->x21 * b[0] + a->x22 * b[1] + a->x23 * b[2];  
    c[2] = a->x31 * b[0] + a->x32 * b[1] + a->x33 * b[2];  
}
```

10.23 Să se scrie o funcție care înmulțește două matrice pătratice de ordinul trei.

FUNCȚIA BX23

```
void mulmatp3(MATP3 *a, MATP3 *b,MATP3 *c )  
/* c=a*b */  
{  
    c->x11 = a->x11 * b->x11 + a->x12 *  
              b->x21 + a->x13 * b->x31;  
    c->x12 = a->x11 * b->x12 + a->x13 *  
              b->x22 + a->x13 * b->x32;  
    c->x13 = a->x11 * b->x13 + a->x12 *  
              b->x23 + a->x13 * b->x33;  
    c->x21 = a->x21 * b->x11 + a->x22 *  
              b->x21 + a->x23 * b->x31;  
    c->x22 = a->x21 * b->x12 + a->x22 *  
              b->x22 + a->x23 * b->x32;  
    c->x23 = a->x21 * b->x13 + a->x22 *  
              b->x23 + a->x23 * b->x33;  
    c->x31 = a->x31 * b->x11 + a->x32 *  
              b->x21 + a->x33 * b->x31;  
    c->x32 = a->x31 * b->x12 + a->x32 *  
              b->x22 + a->x33 * b->x32;  
    c->x33 = a->x31 * b->x13 + a->x32 *  
              b->x23 + a->x33 * b->x33;  
}
```

10.24 Să se scrie o funcție care negativează elementele unei matrice pătratice de ordinul 3.

FUNCȚIA BX24

```
void negmatp3(MATP3 *a, MATP3 *b) /* b = -a */  
{  
    b->x11 = - a->x11;  
    b->x12 = - a->x12;  
    b->x13 = - a->x13;  
    b->x21 = - a->x21;  
    b->x22 = - a->x22;  
    b->x23 = - a->x23;  
    b->x31 = - a->x31;  
    b->x32 = - a->x32;  
    b->x33 = - a->x33;  
}
```

10.25 Să se scrie o funcție care calculează și returnează valoarea determinantului unei matrice pătratice de ordinul 3.

Calculul determinantului se face folosind regula lui *Sarrus*.

FUNCȚIA BX25

```
double detmatp3(MATP3 *a)  
/* calculeaza si returneaza valoarea determinantului matricei a */  
{  
    return a->x11 * a->x22 * a->x33 +  
           a->x12 * a->x23 * a->x31 +  
           a->x21 * a->x32 * a->x13 -  
           a->x31 * a->x22 * a->x13 -  
           a->x11 * a->x32 * a->x23 -  
           a->x21 * a->x12 * a->x33;  
}
```

10.26 Să se scrie o funcție care calculează inversa unei matrice pătratice de ordinul 3. Funcția returnează 0 dacă matricea are determinantul nul și unu în caz contrar.

Fie a matricea de ordinul 3:

```
x11 x12 x13  
x21 x22 x23  
x31 x32 x33
```

Inversa matricei a este matricea:

```
X11 X21 X31  
X12 X22 X32  
X13 X23 X33
```

unde prin X_{ij} am notat complementul algebric al elementului x_{ij} al matricei date. Dacă notăm cu d determinantul matricei a , atunci X_{ij} se calculează astfel:

$X_{ij} = A_{ij} / d$ dacă $i+j$ este par;

și

$X_{ij} = -A_{ij} / d$ daca $i+j$ este impar,
unde prin A_{ij} s-a notat determinantul matricei obtinut din matricea initială suprimind linia a i -a și coloana a j -a.

FUNCȚIA BX26

```
int invmatp3(MATP3 *a, MATP3 *b)
/* - în b se obtine inversa lui a;
   - returneaza:
     0 - daca determinantul matricei este 0;
     1 - in caz contrar. */
{
    double d;

    if(( d = detmatp3( a ) ) == 0 ) return 0;
    b->x11=(a->x22 * a->x33 - a->x33 * a->x22)/d;
    b->x12=(a->x32 * a->x13 - a->x12 * a->x33)/d;
    b->x13=(a->x12 * a->x23 - a->x13 * a->x22)/d;
    b->x21=(a->x31 * a->x23 - a->x21 * a->x33)/d;
    b->x22=(a->x11 * a->x33 - a->x31 * a->x13)/d;
    b->x23=(a->x21 * a->x13 - a->x11 * a->x23)/d;
    b->x31=(a->x21 * a->x32 - a->x31 * a->x22)/d;
    b->x32=(a->x12 * a->x31 - a->x11 * a->x32)/d;
    b->x33=(a->x11 * a->x22 - a->x12 * a->x21)/d;
    return 1;
}
```

10.27 Să se scrie o funcție care citește elementele unei matrice pătratice de ordinul 3 de tip *double*.

Funcția returnează zero la intilnirea sfîrșitului de fișier și unu în caz contrar.

FUNCȚIA BX27

```
int citelem(double *x, int i,int j); /* prototip */
int citmatp3(MATP3 *a)
/* - citește elementele unei matrice patratice de ordinul 3;
   - returneaza:
     0 - la intilnirea sfîrșitului de fișier;
     1 - in caz contrar. */
{
    double l;

    if(citelem(&l,1,1) == 0 ) return 0;
    else a->x11 = l;
    if(citelem(&l,1,2) == 0 ) return 0;
    else a->x12 = l;
    if(citelem(&l,1,3) == 0 ) return 0;
    else a->x13 = l;
    if(citelem(&l,2,1) == 0) return 0;
    else a->x21 = l;
    if(citelem(&l,2,2) == 0 ) return 0;
    else a->x22 = l;
    if(citelem(&l,2,3) == 0 ) return 0;
```

```
else a->x23 = l;
if(citelem(&l,3,1) == 0 ) return 0;
else a->x31 = l;
if(citelem(&l,3,2) == 0 ) return 0;
else a->x32 = l;
if(citelem(&l,3,3) == 0 ) return 0;
else a->x33 = l;
return 1;
} /* sfîrșit funcția citmatp3 */

int citelem( double *x, int i,int j)
/* - citește un număr de tip double care reprezintă elementul unei matrice din linia i și coloana j;
   - returneaza:
     0 - la intilnirea sfîrșitului de fișier;
     1 - altfel. */
{
    char t[255];

    for( ; ; ) {
        printf("elementul din linia %d si coloana %d: ",i,j );
        if(gets(t) == 0 ) return 0;
        if(sscanf(t,"%lf",x) == 1 ) return 1;
        printf("nu s-a tastat un numar\n");
    }
}
```

10.28 Să se scrie un program care rezolvă un sistem de trei ecuații cu trei necunoscute de forma:

$$(1) A^*x = b$$

Programul de față rezolvă sistemul înmulțind relația (1) la stînga cu inversa lui *A*. Deci:

$$x = \text{inv}(A) * b$$

Pașii programului sint:

1. Citește elementele matricei *A* (funcția *citmatp3*, exercițiul 10.27).
2. Citește termenul liber *b* (funcția *pndcit*, exercițiul 8.8).
3. Inversează matricea *A* (funcția *invmatp3*, exercițiul 10.26).
4. Calculează produsul la stînga dintre inversa matricei *A* și termenul liber *b* (funcția *mulmatvect3*, exercițiul 10.22).
5. Afisează soluția obținută *x0*.
6. Afisează reziduurile, adică diferența:
 $r = A^*x0 - b$.

În acest scop se realizează:

- produsul A^*x0

(funcția *mulmatvect3*, exercițiul 10.22);
 – diferență:
 $A \cdot x_0 - b$.

PROGRAMUL BX28

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

typedef struct {
    double x11,x12,x13,x21,x22,x23,x31,x32,x33;
} MATP3;

#include "bviii8.cpp" /* pndcit      */
#include "bx22.cpp"   /* mulmatvect3 */
#include "bx25.cpp"   /* detmatp3   */
#include "bx26.cpp"   /* invmatp3  */
#include "bx27.cpp"   /* citmatp3  */

main() /* - rezolva un sistem de 3 ecuatii cu 3 necunoscute inmultind
           la stanga cu inversa matricei sistemului;
           - afiseaza solutia determinata si reziduurile.*/
{
    MATP3 a,inv;a;
    int i;
    double b[3],x0[3],r[3];
    char er[] = "s-a tastat EOF\n";
    /* citeste matricea sistemului */
    if(citmatp3(&a) == 0 ) {
        printf(er);
        exit(1);
    }
    /* citeste termenul liber */
    if(pndcit(3,b) != 3 ) {
        printf(er);
        exit(1);
    }
    /* inverseaza matricea sistemului */
    if(invmatp3(&a,&inv) == 0 ) {
        printf("determinantul sistemului este nul\n");
        exit(1);
    }
    /* produsul la stanga dintre inversa matricei a si termenul liber b */
    mulmatvect3(&inv,a,b,x0);

    /* afisarea solutiei */
    for( i=0; i < 3; i++) printf("x%d= %g\n", i+1,x0[i]);
}
```

```
/* calculeaza reziduurile */
/* calculeaza produsul a*x0 */
mulmatvect3(&a,x0,r);

/* se afiseaza reziduurile */
for( i = 0; i < 3; i++)
    printf("reziduu linia %d= %g\n",i+1, r[i] - b[i] );
}
```

Observații:

1. Toate funcțiile din exercițiile 10.21 - 10.28, pot fi rescrise folosind tablouri de tip *double* în locul structurilor de tip *MATP3*. Propunem cititorului să facă acest lucru, funcțiile simplificându-se pe baza utilizării instrucțiunilor ciclice.
2. Funcția *citelem* poate fi scrisă mai simplu dacă instrucțiunea:

```
if(citelem(&e,i,j) == 0 )
    return 0;
else
    a -> xij = e; pentru i,j=1,2,3
```

se înlocuiește cu:

```
if(citelem(&a -> xij,i,j) == 0)
    return 0;
```

- 10.29 Să se scrie o funcție care calculează și returnează cel mai mare divizor comun a două numere *m* și *n* de tip *long*.

Funcția utilizează algoritmul lui Euclid (vezi exercițiul 4.24).

FUNCȚIA BX29

```
long cmmdc(long m, long n)/* calculeaza si returneaza (m,n)*/
{
    long r;
    do {
        r=m%n;
        if(r) {
            m=n;
            n=r;
        }
    } while(r);
    return n;
}
```

Observație:

Funcția presupune că *n* este diferit de zero.

- 10.30 Să se scrie o funcție care calculează și returnează cel mai mic multiplu comun a două numere *m* și *n* de tip *long*.

Dacă notăm cu:

$[m,n]$

cel mai mic multiplu comun al numerelor m și n , atunci:

$$[m,n] = m * n / (m,n)$$

unde prin:

(m,n) - Am notat cel mai mare divizor comun al acelorași numere.

FUNCȚIA BX30

```
long cmmmc (long m, long n) /* calculeaza si returneaza [m,n] */
{
    return m*n/cmmdc(m,n);
}
```

Observație:

Funcția presupune că ambele numere sunt diferite de zero.

10.31 În acest exercițiu și în urmatoarele din acest capitol, vom utiliza tipul RATIONAL definit astfel:

```
typedef struct {
    long numrator;
    long numitor;
} RATIONAL;
```

O dată de tip RATIONAL o vom numi *fracție*. Valoarea unei astfel de date este egală cu raportul:

numrator / numitor, dacă *numitor* este diferit de zero

și

nu este definită, dacă *numitor* are valoarea zero

O fracție este *ireductibilă* dacă număratorul și numitorul ei sunt numere prime între ele. Pentru ca o fracție să fie ireductibilă, este suficient să simplificăm fracția respectivă cu cel mai mare divizor comun al număratorului și numitorului ei. Funcția de față transformă o dată de tip RATIONAL într-o fracție ireductibilă.

FUNCȚIA BX31

```
void simplifica(RATIONAL *a)
/* simplifica pe a cu (numrator,numitor) */
{
    long d;

    if(a->numrator == 0 || a->numitor == 0) return;
    d = cmmdc(a->numrator,a->numitor);
    a->numrator /= d;
    a->numitor /= d;
}
```

10.32 Să se scrie o funcție care aduna două date de tip RATIONAL.

Fie a și b cele două date.

Adunarea se execută conform următorilor pași:

1. Dacă număratorul lui a este zero, rezultatul este b și se revine din funcție.
2. Dacă număratorul lui b este zero, rezultatul este a și se revine din funcție.
3. Se calculează cel mai mic multiplu comun al numitorilor lui a și b , fie acesta m .
4. Se calculează:
$$a_1 = (\text{număratorul lui } a) * (m / (\text{numitorul lui } a))$$
5. Se calculează b_1 ca și a_1 , folosind număratorul și numitorul lui b .
6. Număratorul rezultatului este suma:
$$a_1 + b_1$$
7. Numitorul rezultatului este m .
8. Se simplifică rezultatul dacă este posibil.

FUNCȚIA BX32

```
void adfr(RATIONAL *a, RATIONAL *b,RATIONAL *c)
/* c = a + b */
{
    long m;

    if(a->numitor == 0 || b->numitor == 0) return ;
    if(a->numrator == 0) {
        c->numrator = b->numrator;
        c->numitor = b->numitor;
    }
    else
        if(b->numrator == 0) {
            c->numrator = a->numrator;
            c->numitor = a->numitor;
        }
        else {
            m=cmmmc(a->numitor,b->numitor);
            c->numrator = a->numrator*(m/a->numitor)
                + b->numrator * (m/b->numitor);
            c->numitor = m;
        }
    simplifica(c);
}
```

10.33 Să se scrie o funcție care scade două date de tip RATIONAL.

FUNCȚIA BX33

```
void subfr(RATIONAL *a, RATIONAL *b,RATIONAL *c)
/* c = a-b */
{
    RATIONAL temp;
    /* temp = -b */
}
```

```

    temp.numarator = -b->numarator;
    temp.numitor = b->numitor;
    adfr(a,&temp,c);
}

```

10.34 Să se scrie o funcție care înmulțește două date de tip RATIONAL.

FUNCȚIA BX34

```

void mulfr(RATIONAL *a,RATIONAL *b,RATIONAL *c)
/* c = a*b */
{
    c->numarator = a->numarator * b->numarator;
    c->numitor = a->numitor * b->numitor;
    simplifica(c);
}

```

10.35 Să se scrie o funcție care împarte două date de tip RATIONAL.

FUNCȚIA BX35

```

void divfr(RATIONAL *a,RATIONAL *b,RATIONAL *c)
/* c = a/b */
{
    c->numarator = a->numarator * b->numitor;
    c->numitor = a->numitor * b->numarator;
    simplifica(c);
}

```

10.36 Să se scrie o funcție care citește numărătorul și numitorul unei date de tip RATIONAL.

FUNCȚIA BX36

```

int citfr(RATIONAL *a)
/* - citește număratorul și numitorul unei fractii;
   - returnează:
      0 - la întâlnirea sfîrșitului de fisier;
      1 - altfel.*/
{
    char t[255];

    for ( ; ; ) {
        printf("număratorul fractiei: ");
        if(gets(t) == 0) return 0;
        if(sscanf(t,"%ld",&a->numarator) == 1) break;
        printf("nu s-a tastat un intreg\n");
    }
    for( ; ; ) {
        printf("numitorul fractiei: ");
        if(gets(t) == 0) return 0;
        if(sscanf(t,"%ld",&a->numitor) == 1 && a->numitor != 0 )
            break;
        printf("nu s-a tastat un intreg nenul\n");
    }
    return 1;
}

```

)

10.37 Să se scrie un program care citește componentele a trei date de tip RATIONAL, f1, f2 și f3, calculează și afișează sub formă de fracție valoarea expresiei:

$$(f1 - f2) * f3$$

PROGRAMUL BX37

```

#include <stdio.h>
#include <stdlib.h>

typedef struct {
    long numarator;
    long numitor;
} RATIONAL;

#include "bx29.cpp" /* cmmdc */
#include "bx30.cpp" /* emmnc */
#include "bx31.cpp" /* simplifica */
#include "bx32.cpp" /* adfr */
#include "bx33.cpp" /* subfr */
#include "bx34.cpp" /* mulfr */
#include "bx36.cpp" /* citfr */

main() /* citește datele f1,f2 si f3 de tip RATIONAL, calculeaza si
         afișeaza valoarea expresiei:
         (f1-f2)*f3
         sub forma de fractie.*/
{
    RATIONAL f[3],temp;
    char er[] = "s-a tastat EOF\n";
    char *text[] = {"prima fractie\n", "fractia a doua\n",
                    "fractia a treia\n"};
    int i;

    /* citește fractiile: f[0]=f1, f[1]=f2, f[2]=f3 */
    for(i=0; i<=2; i++) {
        printf(text[i]);
        if(citfr(&f[i]) == 0) {
            printf(er);
            exit(1);
        }
    }

    /* temp = f1 - f2 */
    subfr(&f[0],&f[1],&temp);

    /* temp = temp*f3 */
    mulfr(&temp,&f[2],&temp);

    /* scrie rezultatul */
    printf("(f1-f2)*f3:numarator= %ld\n",

```

```

    temp.numerator);
    printf("(f1-f2)*f3: numerator= %ld\n",temp.numerator);
}

```

10.4. Reuniune

Limbajul C ofera utilizatorului posibilitatea de a pastra intr-o zona de memorie date de tipuri diferite. Pana in prezent am vazut ca unei date i se aloca o zona de memorie potrivit tipului datei respective si in zona alocata ei se pot pastra numai date de acel tip. De exemplu, daca avem in vedere declaratia:

```
long x;
```

atunci pentru *x* se aloca 32 de biți și in zona respectiva se păstrează intregi reprezentații prin complement față de 2.

Se pot ivi situații in care in momente diferite ale execuției, am dori ca in aceeași zona de memorie sa putem pastra date de tipuri diferite. De exemplu, daca după un timp nu mai este nevoie de variabila *x*, zona alocată lui *x* ar putea fi utilizata in alte scopuri, pentru a pastra o data de un alt tip: *char*, *int* sau chiar *float*. Astfel de *reutilizări* ale zonelor de memorie pot conduce la economisirea memoriei.

Aceasta este posibil "grupind" imprecună datele care dorim sa fie alocate in aceeași zona de memorie. Pentru a realiza o astfel de grupare se folosesc o construcție similară cu cea pentru *structuri*, diferența constind in aceea ca se schimba cuvintul *struct* cu *union*. In rest, toate formulele intilnite in cazul structurilor rămân valabile. Tipul introdus prin *union* este un tip *definit* de utilizator ca și cel definit prin *struct*. O astfel de data grupată o vom numi *reuniune*.

Exemplu:

```
1. union a {
    int x;
    long y;
    double r;
    char c;
} p;
```

p este o reuniune de tipul *a*.

Componentele *x,y,r* și *c* ale lui *p* le referim ca și in cazul structurilor prin:

p.x, p.y, p.r și p.c

Ele sint alocate in aceeași zonă de memorie și de aceea, la un moment dat al execuției, numai una din aceste componente este definită(alocată).

Pentru *p* se aloca o zonă de memorie suficientă pentru a pastra data care necesită numarul maxim de octeți, deci se vor aloca 8 octeți necesari pentru a putea pastra componenta *r* de tip *double*.

Să observăm ca daca inlocuim cuvintul *union* prin *struct*, atunci pentru *p* se

aloca:

- 2 octeți pentru *x*;
- 4 octeți pentru *y*;
- 8 octeți pentru *r*;
- 1 octet pentru *c*,

deci in total 15 octeți. Aceasta deoarece, in cazul unei structuri, componentele ei sunt definite simultan și deci trebuie să fie alocate in zone diferite de memorie.

```
2. typedef union {
    char nume[70];
    int nrmat;
    long cod;
} ZC;
ZC sir;
```

sir este o reuniune de tip *ZC* pentru care s-au alocat 70 de octeți. În această zonă se pot pastra siruri de caractere la care ne putem referi prin:

```
sir.nume;
sir.nume[0];
sir.nume[1];
etc.
```

sau intregi de tip *int* sau *long*. La aceștia ne referim prin:

sir.nrmat

și

sir.cod.

Fie:

*ZC *p;*

atunci putem realiza o atribuire de forma:

p = &sir;

Prin intermediul lui *p* ne putem referi la componente reuniunii *sir* astfel:

```
p -> nume;
p -> nume[0];
p -> nume[1]
etc.
p -> nrmat
```

și

p -> cod.

In legătură cu reuniunile, pot apărea probleme la utilizarea lor, deoarece programatorul trebuie să cunoască, in fiecare moment al execuției, ce componentă a reuniunii este prezentă in zona alocată ei.

Fie reuniunea:

```
union {
    int i;
    float f;
    double d;
} zc;
```

Dacă la un moment dat, în zona alocată datei *zc* se păstrează un întreg de tip *int* (de exemplu: se face atribuirea *zc.i* = 10) și se fac referiri la componenta *d*:

```
if( zc.d > 0 )
```

atunci rezultatul comparării va fi eronat. Această eroare nu poate fi semnalată de compilator și nici la execuție și de aceea, astfel de utilizări pot fi evitate numai de programator.

Pentru a înlătura eforile de acest fel, se recomandă utilizarea unui indicator care să definească tipul datei păstrate în fiecare moment în zona alocată reuniunii respective. De exemplu, în cazul reuniunii *zc* de mai sus este necesar un indicator care să aibă trei valori corespunzătoare celor trei tipuri ale componentelor lui *zc* (*int*, *float* și *double*). Aceste valori le numim sugestiv prin constante simbolice:

```
#define INTREG 1
#define FSIMPLU 2
#define FDUBLU 3
```

Adăugăm la reuniunea de mai sus un indicator care are una din aceste valori, în funcție de componentă prezentă în zona reuniunii. Se obține tipul utilizator de mai jos:

```
struct tszc {
    int tipcrt; /* indicator care defineste tipul curent */
    union {
        int i;
        float f;
        double d;
    } zc;
};
```

Declaram structura de tip *tszc*:

```
struct tszc szc;
```

Structura are două componente:

- | | |
|---------------|---|
| <i>tipcrt</i> | - Este de tip <i>int</i> . |
| <i>zc</i> | - Este o reuniune de componente <i>i</i> , <i>f</i> și <i>d</i> . |

Deci, tot timpul este alocată data *tipcrt* și numai una din componentele *i*, *f*, *d*.

Pentru a păstra o dată de tip *int*, de exemplu valoarea 123, vom folosi atribuirea:

```
szc.zc.i = 123;
```

Alături de această atribuire vom mai utiliza încă una și anume:

```
szc.tipcrt = INTREG;
```

În felul acesta, componenta *tipcrt* ne permite să stabilim faptul că reuniunea *zc* conține o dată de tip *int*.

În mod analog, dacă dorim să păstrăm, în zona respectiva, o dată flotantă de tip *double*, de exemplu valoarea lui *pi*, vom folosi atribuirile:

```
szc.zc.d = 3.14159265;
```

```
szc.tipcrt = FDUBLU;
```

La utilizarea datelor păstrate în acest fel se poate folosi o construcție *if* de formă:

```
if(szc.tipcrt == INTREG)
    se folosește szc.zc.i
else if(szc.tipcrt == FSIMPLU)
    se folosește szc.zc.f
else if(szc.tipcrt == FDUBLU)
    se folosește szc.zc.d
else
    eroare
```

Acaceași utilizare se obține folosind o instrucție *switch*:

```
switch (szc.tipcrt) {
    case INTREG:
        se folosește szc.zc.i
        break;
    case FSIMPLU:
        se folosește szc.zc.f
        break;
    case FDUBLU:
        se folosește szc.zc.d
        break;
    default:
        eroare
}
```

În încheierea acestui paragraf amintim că reuniunile *nu pot* fi inițializate, spre deosebire de structuri.

Exerciții:

10.38 Fie tipul FIG declarat ca mai jos:

```
typedef struct {
    int tip; /* tipul figurii */
    union {
        double raza; /* cere */
```

```

        double lp; /* patrat */
        double ld[2]; /* dreptunghi */
        double lt[3]; /* triunghi */
    } fig;
} FIG;

```

O dată de tip FIG conține elementele unei figuri necesare pentru a calcula aria figurii respective. Figurile avute în vedere sunt:

- cerc;
 - pătrat;
 - dreptunghi
- și
- triunghi.

În cazul primelor două figuri, data de tip FIG conține o valoare de tip *double* care, în cazul cercului reprezintă lungimea razei acestuia, iar în cazul pătratului, lungimea laturii pătratului. În cazul dreptunghiului, data conține două elemente de tip *double*: lungimea și lățimea. În sfîrșit, în cazul triunghiului, data conține trei valori de tip *double* care reprezintă lungimile celor trei laturi ale triunghiului.

Componenta *tip* definește elementele (figura) prezente într-o dată de tip FIG și are valorile:

- | | |
|----|----------------------|
| 0 | - Pentru cerc. |
| 1 | - Pentru pătrat. |
| 2 | - Pentru dreptunghi. |
| 3 | - Pentru triunghi. |
| -1 | - Pentru eroare. |

În locul acestor valori, considerăm constantele simbolice:

```

#define EROARE -1
#define CERC 0
#define PATRAT 1
#define DREPTUNGHI 2
#define TRIUNGHI 3

```

Funcția de mai jos are ca parametru o dată de tip FIG, calculează și returnează aria figurii ale cărei elemente sunt conținute în zona definită de parametru.

Funcția returnează 0 în cazul în care datele sunt eronate.

La calculul ariei unui triunghi se folosește formula lui Heron.

FUNCȚIA BX38

```

#define PI 3.14159265358979

double aria(FIG *p)
/* calculeaza si returneaza aria figurii definite de elementele prezente in
   zona spre care pointeaza p; la eroare returneaza 0 */
{
    double sp,a,b,c;

```

```

switch(p->tip) {
    case CERC:
        return PI*p->fig.raza*p->fig.raza;
    case PATRAT:
        return p->fig.lp*p->fig.lp;
    case DREPTUNGHI:
        return p->fig.ld[0]*p->fig.ld[1];
    case TRIUNGHI:
        sp=(p->fig.lt[0] + p->fig.lt[1] + p->fig.lt[2])/2;
        if((a=sp - p->fig.lt[0]) > 0 &&
           (b=sp - p->fig.lt[1]) > 0 &&
           (c=sp - p->fig.lt[2]) > 0 )
            return sqrt(sp*a*b*c);
        else { /* cele 3 valori nu reprezinta lungimile laturilor unui triunghi */
            printf("a= %g\ tb= %g\ tc= %g\tnu\
                   formeaza un triunghi\n",
                   p->fig.lt[0], p->fig.lt[1],p->fig.lt[2] );
            return 0;
        }
    default:/* eroare */
        return 0;
}

```

- 10.39 Să se scrie o funcție care are ca parametru un pointer spre o dată de tip FIG și care citește și păstrează elementele figurii definite de componenta *tip* a datei de tip FIG.

FUNCȚIA BX39

```

int citfig(FIG *p)
/* - citește elementele figurii definite de p->tip;
   - returnează:
     0 - la inițializarea sfîrșitului de fisier sau la eroare;
     1 - altfel. */
{
    char t[255];

    switch(p->tip) {
        case CERC:
            for( ; ; ) {
                printf("raza= ");
                if(gets(t) == 0) return 0;
                if(sscanf(t,"%lf",&p->fig.raza)== 1 &&
                   p->fig.raza>0)
                    return 1;
                printf("nu s-a tastat un numar pozitiv\n");
            }
        case PATRAT:
            for( ; ; ) {
                printf("latura patratului= ");
                if(gets(t) == 0) return 0;
                if(sscanf(t,"%lf",&p->fig.lp) == 1 &&
                   p->fig.lp > 0)

```

```

        return 1;
    printf("nu s-a tastat un numar pozitiv\n");
}
case DREPTUNGHI:
for( ; ; ) {
    printf("lungimea si latimea pe aceeasi linie: ");
    if(gets(t) == 0) return 0;
    if(sscanf(t,"%lf %lf",&p->fig.ld[0],
              &p->fig.ld[1])==2 && p->fig.ld[0] > 0 &&
       p->fig.ld[1] > 0)
        return 1;
    printf("nu s-au tastat 2 numere pozitive\n");
}
case TRIUNGHI:
for( ; ; ) {
    printf("laturile triunghiului pe \
           aceeasi linie: ");
    if(gets(t) == 0) return 0;
    if(sscanf(t,"%lf %lf %lf",&p->fig.lt[0],
              &p->fig.lt[1],
              &p->fig.lt[2])==3 && p->fig.lt[0]>0 &&
       p->fig.lt[1]>0 && p->fig.lt[2]> 0 )
        return 1;
    printf("nu s-au tastat 3 numere pozitive\n");
}
default:
    return 0;
}

```

- 10.40 Să se scrie un program care citește elementele unei figuri necesare pentru calculul ariei sale. Elementele sunt precedate de o literă mare, care definește figura. Această literă se tastează la început, singură pe o linie.

Programul afișează aria figurii.

Corespondența dintre litere și figuri este următoarea:

- C* - Pentru cerc.
- P* - Pentru pătrat.
- D* - Pentru dreptunghi.
- T* - Pentru triunghi.

PROGRAMUL BX40

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

typedef struct {
    int tip;
    union {
        double raza;
        double lp;
    }
}
```

```

    double ld[2];
    double lt[3];
} FIG;

#define EROARE -1
#define CERC 0
#define PATRAT 1
#define DREPTUNGHI 2
#define TRIUNGHI 3

#include "bx38.cpp" /*arie */
#include "bx39.cpp" /*ciffig */

main() /* - citeste o literă mare care defineste o figura geometrică,
         apoi citeste elementele figurii respective;
         - calculeaza si afiseaza aria acelei figurii.*/
{
    char t[255];
    char er[]="s-a tastat EOF\n";
    char lit[2];
    FIG f;
    double a;

    for( ; ; ) {
        printf("tastati una din literele mari:\n");
        printf("C\nD\nP\nT\n");
        if(gets(t) == 0){
            printf(er);
            exit(1);
        }
        sscanf(t,"%ls",lit);
        switch(lit[0]) {
            case 'C': /*cerc*/
                f.tip = CERC;
                break;
            case 'D': /*dreptunghi*/
                f.tip = DREPTUNGHI;
                break;
            case 'P': /*patrat*/
                f.tip = PATRAT;
                break;
            case 'T': /*triunghi*/
                f.tip = TRIUNGHI;
                break;
            default: /*eroare*/
                printf("nu s-a tastat una din literele\
                       mari C,D,P sau T\n");
                f.tip = EROARE;
        }
        if(f.tip != EROARE) break;
    } /*sfarsit for*/
}
```

```

/* citeste elementele figurii */
if (citfig(&f) == 0) {
    printf(er);
    exit(1);
}

/* calculeaza si afiseaza aria figurii */
if ((a=aria(&f)) == 0) exit(1);
printf("aria figurii= %g\n", a );
)

```

10.41 Fie tipul utilizator introdus prin următoarea declarație:

```

typedef struct d_c {
    int zz;
    int tipluna;
    union {
        int nll;
        char sluna[11];
    } luna;
    int an;
} D_C ;

```

unde:

<i>tipluna</i>	- Poate avea una din valorile: INTREG sau SIR
----------------	--

Dacă *tipluna* are valoarea INTREG, atunci luna se exprimă prin numărul ei care se păstrează ca valoare a variabilei *nll*. În caz contrar, luna este păstrată prin denumirea ei în tabloul *sluna*.

Funcția de față are ca parametru un pointer spre o dată de tip *D_C* și returnează numărul lunii calendaristice. La eroare, returnează valoarea 0.

În cazul în care luna calendaristică se dă prin denumirea ei, se utilizează tabloul de pointeri *tpdl* definit în exercițiul 8.19.

FUNCȚIA BX41

```

int nrluna(D_C *p) /* returneaza numarul lunii sau 0 la eroare */
{
    int i;
    static char *tpdl[] = {"",
        "ianuarie",
        "februarie",
        "martie",
        "aprilie",
        "mai",
        "iunie",
        "iulie",
        "august",
        "septembrie",
        "octombrie",
        "noiembrie",
    }

```

```

        "decembrie"
    );

    if (p->tipluna == INTREG)
        /* luna sc da prin numarul ci */
        return p->luna.nll <1 || p->luna.nll >12 ? 0 : p->luna.nll;

    if (p->tipluna != SIR)
        /* tipluna nu are o valoare corecta */
        return 0;

    /* se cauta denumirea lunii in tabloul tpdl */
    for (i=1; i <13; i++)
        if (strcmp(p->luna.sluna,tpdl[i]) == 0) return i;
    return 0;
}

```

10.42 Funcția din acest exercițiu are ca parametru un pointer spre o dată de tip *D_C*, validează data calendaristică definită de pointerul respectiv și returnează numărul lunii din data respectivă. În cazul în care data este eronată, funcția returnează valoarea zero.

Funcția validează data calendaristică apelind funcția *v_calend* definită în exercițiul 6.5.

FUNCȚIA BX42

```

int vnrluna(D_C *p)
/* - valideaza data calendaristica spre care pointeaza p;
   - returneaza numarul lunii calendaristice si 0 la eroare. */
{
    int nr;

    if((nr=nrluna(p)) == 0) return 0; /* luna eronata */
    if(v_calend(p->zz,nr,p->an)) return nr; /* data calendaristica valida */
    return 0; /* data calendaristica eronata */
}

```

10.43 Să se scrie un program care afișează diferența, în zile, dintre două date calendaristice. Una dintre cele două date se dă sub formă de argument în linia de comandă, iar cealaltă se citește de la tastatură.

Notăm cu *d1* data din linia de comandă și cu *d2* data care se tastează. Programul afișează diferența *d2 -d1*, în valoare absolută.

Data *d1* are formatul:

zi luna an

unde:

<i>zi</i>	- Este un întreg de 1-2 cifre.
<i>luna</i>	- Este denumirea lunii calendaristice.

an - Este intreg de 4 cifre și reprezinta anul din intervalul [1600,4900].

Data *d2* are un format similar cu deosebirea că luna calendaristică se dă prin numărul ei.

Se pot tasta mai multe date *d2* la o aceeași execuție a programului.

PROGRAMUL BX43

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

typedef struct {
    int zz;
    int tipluna;
    union {
        int nll;
        char sluna[11];
    } luna;
    int an;
} D_C;

#define INTREG 1
#define SIR 2
#define EROARE -1

#include "bvi5.cpp" /*v_calend */
#include "bvi6.cpp" /*zi_din_an */
#include "bx41.cpp" /*nrluna */
#include "bx42.cpp" /*vnrluna */
#include "bvi12.cpp" /*pcit_int */
#include "bvi13.cpp" /*pcit_int_lim */

int nrzile[] = { 0,31,28,31,30,31,30,31,31,30,31,30,31,30,31 };

main(int argc, char *argv[])
/* afiseaza numarul de zile dintre doua date calendaristice valide */
{
    char er[]="s-a tastat EOF\n";
    D_C d1,d2;
    int n,n1,n2,t,min,max;

    /* pastreaza data calendaristica din linia de comanda in structura d1 */
    if(argc != 4) {
        printf("numar argumente= %d este eronat\n",argc);
        exit(1);
    }
    if(sscanf(argv[1],"%d",&d1.zz) != 1) {
        printf("ziua din linia de comanda eronata\n");
        exit(1);
    }
    if(sscanf(argv[2],"%10s",d1.luna.sluna) != 1) {
        printf("luna din linia de comanda eronata\n");
        exit(1);
    }
}
```

```
        exit(1);
    }
    if(sscanf(argv[3],"%d",&d1.an) != 1) {
        printf("anul din linia de comanda eronat\n");
        exit(1);
    }
    d1.tipluna = SIR;

    /* valideaza data din d1 */
    if((n=vnrluna(&d1)) == 0 ) {
        printf("data calendaristica din linia de comanda\
               eronata\n");
        printf("zi: %d\luna: %s\tanul: %d\n",d1.zz,
               d1.luna.sluna, d1.an);
        exit(1);
    }

    /* numar zile din an */
    n1=zi_din_an(d1.zz,n,d1.an);

    for( ; ; ) {
        /* citeste o data calendaristica de la tastatura si o pastreaza in structura d2 */
        if(pcit_int_lim("ziua ",1,31,&d2.zz) == 0)
            exit(0); /* s-a tastat EOF */
        if(pcit_int_lim("numarul lunii calendaristice"
                        ,1,12, &d2.luna.nll) == 0 ) {
            printf(er);
            exit(1);
        }
        if(pcit_int_lim("anul ",1600,4900,&d2.an)==0 )
        {
            printf(er);
            exit(1);
        }
        d2.tipluna = INTREG;

        /* valideaza data calendaristica din d2 */
        if(vnrluna(&d2) == 0 ) {
            printf("data calendaristica tastata este eronata\n");
            printf("zi: %d\luna: %d\tanul: %d\n",
                   d2.zz,d2.luna.nll,d2.an);
            continue;
        }

        /* numar zile din an */
        n2=zi_din_an(d2.zz,d2.luna.nll,d2.an);

        /* determina relatiile dintre ani */
        if(d1.an < d2.an) {
            min=d1.an;
            max=d2.an;
        }
        else {
            min = d2.an;
```

```

        max = (d1.an);
    }

/* insumeaza diferența dintre ani în zile */
for (t=0; min < max; min++)
    t += 365 + (min%4==0 && min%100 == 0 || min%400 == 0);
if (min == d1.an) t -= nt - n2; /* t=d1-d2 */
else t -= n2 - nt; /* t=d2-d1 */
if (t < 0) t += 1;

printf("d1= %d/%d/%d\n", d1.aa,
      n, d1.an, d2.aa, d2.luna, n1, d2.an);
printf("d2= %d/%d/%d\n", t);
}

```

10.5. Cimp

Limbajul C permite definirea și prelucrarea datelor pe *biți*. Utilizarea lor poate conduce la economisirea de memorie. Într-adevar, adesea avem nevoie de date care au numai două valori, zero sau unu. O astfel de date poate fi pastrată pe un singur bit. De aceea, pentru astfel de date, nu se justifică să alocăm un octet sau chiar doi. În general, nu este util ca date de valori mici să fie pastrate pe octeți sau pe 16 biți, mai ales atunci cind aceste date sunt în numar mare. În acest scop, limbajul C oferă posibilitatea de a declara date care să se *aloce pe biți*.

Un sir de biți adjacenți formează un *cimp*. Un cimp trebuie să se poată păstra într-un cuvint calculator.

Mai multe cimpi pot fi pastrate într-un același cuvint calculator.

Cimpii se grupează formând o structură. O astfel de structură se declară ca o structură obișnuită care are ca și componente cimpi:

```

struct name {
    cimp_1;
    cimp_2;
    ...
    cimp_n;
} nume1, nume2, ..., numem;

```

Un cimp se declară astfel:

tip nume_cimp: lungime_in_bitii

sau

tip: lungime_in_bitii

De obicei, *tip* este cuvintul cheie *unsigned*, ceea ce înseamnă că sirul de biți din cimpul respectiv se interpretează ca fiind un întreg fără semn. Alte posibilități pentru *tip* sunt:

- int;

- unsigned char;

și

- char.

Cimpii se alocă de la biții de ordin inferior ai cuvintului spre cei de ordin superior.

Cimpii cu semn se utilizează pentru a păstra întregi de valori mici prin complement față de doi. De aceea, în acest caz, bitul cel mai semnificativ al cimpului este bit semn.

Dacă un cimp nu se poate aloca în cuvintul curent, el se va aloca în cuvintul următor.

Un cimp fără *name* nu se poate referi. El definește o zonă neutilizată dintr-un cuvint.

Lungimea în biți poate fi egală cu zero. În acest caz, data următoare se alocă în cuvintul următor.

Cimpii se pot referi folosind aceleasi convenții ca și în cazul structurilor obișnuite.

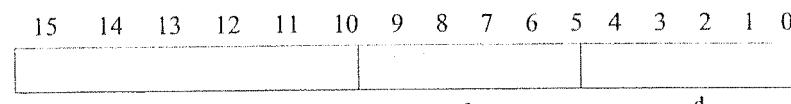
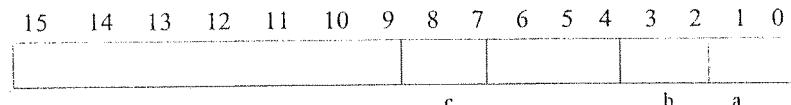
Exemplu:

```

struct {
    unsigned a:2;
    int     b:2;
    unsigned :3;
    unsigned c:2;
    unsigned :0;
    int     d:5;
    unsigned e:5;
} x,y;

```

Pentru *x* se alocă două cuvinte, astfel:



x.a = 1

- Atribuie cimpului *a* al datei *x* valoarea 1, deci bitul 0 devine 1, iar bitul 1 ia valoarea 0.

x.b = -1

- Atribuie cimpului *b*, al datei *x*, valoarea -1.

- Aceasta înseamnă că ambii biți ai lui *b* se fac egali cu 1 (11 este reprezentarea lui -1 pe 2 biți prin complement față de 2).

Nu se pot defini tablouri de cimpuri. De asemenea, operatorul adresă (& unar) nu se poate aplica la un cimp.

Datele pe biți conduc la programe care, de obicei, nu sunt portabile sau au o portabilitate redusă. De aceea, se recomandă utilizarea lor cu precauție.

De asemenea, datele pe biți necesită instrucțiuni suplimentare (deplasări, setări și/sau mascări de biți etc.) față de cazul cind sunt păstrate în mod obișnuit (ca date de tip *int* sau *char*). De aceea, utilizarea lor se justifică numai atunci cind alocarea pe biți conduce la o economie substanțială de memorie față de alocarea pe octeți sau pe cuvinte de 16 biți.

Observație:

Prelucrarea datelor pe biți se poate realiza și fără a defini cimpuri de biți, utilizând operatorii logici pe biți. Utilizarea lor poate conduce însă la un efort de programare suplimentar care poate fi destul de mare. De asemenea, utilizarea operatorilor respectivi poate să nu fie făcută optim sau să conducă la folosirea unor expresii eronate.

Aceasta nu înseamnă că trebuie să renunțăm la utilizarea operatorilor logici pe biți. Există situații cind utilizarea lor permite serierea unor programe mai performante decât dacă se utilizează, în același scopuri, cimpuri de biți.

Exerciții:

10.44 Să se scrie o funcție care stabilește dacă o relație binară este o relație de echivalență.

Fie A și B două mulțimi (disjuncte sau nu, care pot și coincide).

Prin AxB se notează mulțimea perechilor ordonate de forma:

$\langle a, b \rangle$

unde:

a - Este un element al mulțimii A, iar b un element al mulțimii B.

Mulțimea AxB se numește *produsul cartezian* al mulțimilor A și B.

O submulțime a produsului cartezian AxB se numește *relație binară*.

Dacă $A = B$, atunci o submulțime a produsului cartezian AxA definește o relație binară pe mulțimea A.

Fie R o relație binară, adică o submulțime a produsului cartezian AxB . Dacă perechea:

$\langle a, b \rangle$

apartine lui R, atunci se spune că a și b se află în relația R. Notăm acest lucru prin:

(1) aRb

și se obișnuiește să se spună că expresia (1) este *adevărată*. Dacă a și b nu sunt în

relația R, atunci spunem că expresia (1) este *falsa*.

Exemple:

1. Dacă $A = B = N$, unde prin N am notat mulțimea numerelor naturale, atunci sunt definite o serie de relații binare pe mulțimea N. De exemplu, relația *mai mic* este definită pe mulțimea N și ea se notează prin simbolul $<$.

De exemplu, perechea:

$\langle 3, 7 \rangle$

apartine relației mai mic ($<$) și aceasta înseamnă că expresia:

$3 < 7$

este adevarată. În schimb expresia:

$7 < 3$

este falsă.

2. Fie mulțimile:

$A = \{1, 2, 3\}$ și $B = \{2, 5, 7, 8\}$

Atunci:

$AxB = \{\langle 1, 2 \rangle, \langle 1, 5 \rangle, \langle 1, 7 \rangle, \langle 1, 8 \rangle, \langle 2, 2 \rangle, \langle 2, 5 \rangle, \langle 2, 7 \rangle, \langle 2, 8 \rangle, \langle 3, 2 \rangle, \langle 3, 5 \rangle, \langle 3, 7 \rangle, \langle 3, 8 \rangle\}$

Relația R este submulțimea perechilor produsului cartezian cu proprietatea că primul element divide pe al doilea.

Atunci:

$R = \{\langle 1, 2 \rangle, \langle 1, 5 \rangle, \langle 1, 7 \rangle, \langle 1, 8 \rangle, \langle 2, 2 \rangle, \langle 2, 8 \rangle\}$

Dacă notăm cu R1 submulțimea perechilor produsului AxB cu proprietatea că primul element este multiplul celui de-al doilea, atunci relația R1 este o relație vidă.

Fie R o relație definită pe mulțimea A.

Dacă

aRa

este o expresie adevarată pentru orice a din A, atunci se spune că relația R este *reflexivă*.

Relația $=$ definită pe mulțimea numerelor naturale nu este o relație reflexivă. Pe aceeași mulțime se poate defini relația de egalitate $=$.

Perechea:

$\langle a, a \rangle$

este în relația $=$, oricare ar fi a un număr natural, deci relația $=$ este reflexivă.

Relația R definită pe mulțimea A se spune că este *simetrică* dacă din faptul că expresia:

aRb

este adevarată, rezultă că și expresia:

bRa

este adevarata.

Relatia \prec , definita pe multimea numerelor naturale nu este simetrica. In schimb, relatia $=$ este simetrica.

In sfirșit, relatia R este *tranzitiva* daca din faptul ca expresile:

$$aRb \text{ și } bRc$$

sunt adevarate, rezulta ca și expresia:

$$aRc$$

este adevarata.

Relatiile \sim și \approx definite pe multimea numerelor naturale N sunt tranzitive.

Daca expresiile:

$$a < b \text{ și } b < c$$

sunt adevarate, rezulta ca și expresia:

$$a < c$$

este adevarata oricare ar fi a, b, c numere naturale.

O relatia R definita pe multimea A, care este reflexiva, simetrica și tranzitiva, se numeste relatie de *echivalență*.

Relatia \prec nu este o relatie de echivalență nefiind simetrica și nici reflexiva.

Relatia \leq (mai mic egal) este reflexiva dar nu și simetrica, deci nu este o relatie de echivalență.

Relatia \neq (diferit) este o relatie simetrica dar nu este reflexiva.

Un exemplu nebalan de relatie de echivalență este relatiea de asemănare definita pe multimea triunghiurilor plane. Această relație se notează cu simbolul \sim .

Dacă a și b sunt două triunghiuri plane asemenea, atunci din faptul că a este asemenea cu b , rezulta că și b este asemenea cu a . Deci, relația de asemănare este simetrică. Ea este și reflexivă, deoarece un triunghi este totdeauna asemenea cu el însuși.

In sfirșit, dacă a, b, c sunt trei triunghiuri și au loc relațiile:

$$a \sim b \text{ și } b \sim c,$$

atunci are loc și relația:

$$a \sim c.$$

In acest exemplu, precum și in continuare, vom avea in vedere numai relații cu un număr finit de elemente. Relațiile de acest fel se pot reprezenta prin matrici.

Dacă relația R este o submulțime a produsului cartezian $A \times B$, atunci reprezentăm relația respectivă prin matricea a astfel:

- Fiecarui element din mulțimea A i se pune in corespondență o linie in matricea a .

- Fiecarui element din mulțimea B i se pune in corespondență o coloană in matricea a .
- Fie ai elementul mulțimii A căruia ii corespunde linia a i-a din matricea a și bj elementul mulțimii B căruia ii corespunde coloana a j-a din matricea a . Atunci:
 - $a[i][j] = 1$, dacă $ai R bj$
 - $a[i][j] = 0$, altfel.

Exemplu:

Să consideram exemplul 2 de mai sus. Matricea a conține 3 linii și 4 coloane. Relația R se reprezintă prin următoarea matrice:

A	2	5	7	8
	1	1	1	1
	2	1	0	0
	3	0	0	0

B

In cazul in care $A = B$, matricea este pătratică. Relațiile de echivalență se definesc pentru astfel de cazuri. Proprietatea de reflexivitate se exprimă simplu și anume, matricea conține 1 pe diagonala principală.

Intr-adevăr, relația R este reflexivă dacă:

$$ai R ai$$

este adevarata, adica daca

$$a[i][i] = 1, \text{ pentru toate valorile lui } i.$$

Relația R este simetrică dacă matricea de reprezentare a ei este simetrică.

Intr-adevăr, dacă R este simetrică, atunci din faptul că expresia:

$$ai R aj$$

este adevarata, rezulta ca și expresia:

$$aj R ai$$

este adevarata. Aceasta inseamna ca:

$$a[i][j] = a[j][i] = 1$$

iar dacă:

$$a[i][j] = 0, \text{ atunci } a[j][i] = 0$$

deci:

$$a[i][j] = a[j][i]$$

Pentru a verifica tranzitivitatea relației, să considerăm definiția tranzitivității:

R este tranzitivă dacă din faptul că expresiile:

ai R aj și aj R ak

sunt adevărate, rezultă că și expresia:

ai R ak

este adevărată.

Deci, este necesar să se verifice că dacă

$a[i][j] = a[j][k] = 1$

atunci și

$a[i][k] = 1$, pentru toate valorile lui i, j și k.

Dacă există cel puțin un element

$a[i][k] = 0$

în timp ce

$a[i][j] = a[j][k] = 1$

pentru cel puțin un j, atunci relația nu este tranzitivă.

Elementele matricei de reprezentare a unei relații au numai două valori: 0 sau 1.

De aceea, o astfel de matrice se poate păstra pe biți. Aceasta permite realizarea unei economii de memorie importantă față de cazurile în care matricea se păstrează pe octeți sau cuvinte de 16 biți. De exemplu, dacă se utilizează o matrice de ordinul 100, atunci necesarul de memorie va fi:

100×100 biți = $10000 / 8$ octeți = 1250 octeți,

față de 10000 de octeți, cind matricea s-ar păstra pe octeți.

Având în vedere acest fapt, funcția de față utilizează o matrice cu elemente păstrate pe biți, pentru a reprezenta relația de analizat.

Ea returnează valoarea 1 dacă relația este de echivalență și zero în caz contrar.

Matricea relației se păstrează într-un tablou unidimensional de tip *char* și fiecare element al tabloului păstrează 8 elemente consecutive din matricea respectivă.

Fie *a* matricea relației, care este o matrice pătratică de ordinul *n*.

Notăm cu *ta* tabloul unidimensional care păstrează elementele matricei *a* pe biți. Problema care ne interesează este accesul la elementul $a[i][j]$ al matricei *a*, reprezentată prin tabloul *ta* ($i = 0, 1, \dots, n-1$ și $j = 0, 1, \dots, n-1$).

Dacă elementele matricei *a* s-ar păstra pe octeți, atunci folosind relația de liniarizare (vezi exercițiul 5.1), acest element ar avea indicele:

$k = i \cdot n + j$

deci

$a[i][j]$ ar fi elementul $ta[k = i \cdot n + j]$.

Deoarece fiecare element al tabloului *ta* păstrează 8 elemente din matricea *a*, rezultă că valoarea lui *k*, de mai sus, trebuie micșorată de 8 ori.

Avem:

$a[0][0], a[0][1], \dots, a[0][7]$, se păstrează în $ta[0]$,

$a[0][8], a[0][9], \dots, a[0][15]$, se păstrează în $ta[1]$

...

În general, elementul $a[i][j]$ se păstrează în octetul de indice $(i \cdot n + j) / 8$, adică este un bit al elementului:

$ta[(i \cdot n + j) / 8]$

Pozitia bitului se determină astfel:

- dacă $(i \cdot n + j) \% 8$ este zero,
atunci este bitul cel mai semnificativ (bitul 7)
- dacă $(i \cdot n + j) \% 8 = 1$,
atunci este bitul 6;
- în general, dacă $(i \cdot n + j) \% 8 = r$,
atunci este bitul $7-r$.

FUNCTIA BX44

```
unsigned belem(char [], int, int, int);

int relechiv(char ta[], int n)
/* returneaza 1 daca relația păstrată în tabloul ta pe biți este o relație de
echivalență și zero în caz contrar */
{
    int i, j, k;

    /* reflexivitate */
    for(i=0; i < n; i++)
        if(belem(ta, n, i, i) == 0)
            return 0; /* relația nu este reflexiva */

    for( i=0; i < n; i++)
        for(j=0; j < n; j++)
            if(belem(ta, n, i, j) != belem(ta, n, j, i))
                return 0; /* relația nu este simetrică */

    for(i=0; i < n; i++)
        for( j=0; j < n; j++)
            if(belem(ta, n, i, j))
                for(k=0; k < n; k++)
                    if(belem(ta, n, j, k))
                        if(belem(ta, n, i, k) == 0)
                            return 0;
}
```

```

/* relația este tranzitivă */
return 1;
}

unsigned belem(char ta[], int n, int i, int j)
/* returnează valoarea elementului a[i][j];
- elementele matricei patratice a, de ordinul n, sunt pastrate pe biti în
tabloul ta, folosind relația de liniarizare.*/
{
    return ta[(i*n+j)/8]>> 7-(i*n+j)%8&1;
}

```

Observație:

Funcția *belem* returnează valoarea elementului $a[i][j]$. Așa cum s-a indicat mai sus, elementul respectiv este pastrat într-un bit al elementului:

$ta[(i*n+j)/8]$.

Selectarea acestui bit se face deplasind spre dreapta valoarea acestui element cu:

$7 - (i*n+j)\%8$

poziții binare (prin aceasta bitul în cauză devine cel mai puțin semnificativ), iar apoi se face un și *logic pe biți* dintre rezultatul acestei deplasări și 1 (în felul acesta se anulează toți biții exceptând ultimul, care este chiar valoarea lui $a[i][j]$).

Această funcție se poate realiza folosind cimpuri de biți. În acest scop, se folosește o reuniune cu două componente. Una este o structură care definește fiecare bit al unui octet, iar cealaltă este o dată de tip *char*. Elementul $ta[(i*n+j)/8]$ se atribuie datei de tip *char* și apoi se selectează bitul dorit prin acces direct la el.

```

unsigned belem(char ta[], int n, int i, int j)
/* returnează elementul a[i][j] pastrat pe biti în tabloul ta*/
{
    union {
        struct {
            unsigned b0:1;
            unsigned b1:1;
            unsigned b2:1;
            unsigned b3:1;
            unsigned b4:1;
            unsigned b5:1;
            unsigned b6:1;
            unsigned b7:1;
        } b;
        char x;
    } octet;
    /* pastreaza octetul ce conține bitul de selectat în zona alocată reuniunii */
    octet.x = ta[(i*n+j)/8];
}

```

```

/* selectează bitul de ordin 7-(i*n+j)%8 */
switch ( 7-(i*n+j)%8 ) {
    case 0: return octet.b.b0;
    case 1: return octet.b.b1;
    case 2: return octet.b.b2;
    case 3: return octet.b.b3;
    case 4: return octet.b.b4;
    case 5: return octet.b.b5;
    case 6: return octet.b.b6;
    case 7: return octet.b.b7;
}

```

10.45 Să se scrie o funcție care calculează compunerea a două relații.

Fie P și Q două relații reprezentate prin matricele MP și MQ.

Operatorul de compunere a două relații îl notăm prin caracterul punct.

Fie R compunerea celor două relații P și Q:

$$R = P.Q$$

Operatorul de compunere a două relații se definește ca mai jos.

Fie P o submulțime a produsului cartezian:

$$A \times B$$

și Q o submulțime a produsului cartezian:

$$B \times C$$

Atunci R este o submulțime a produsului cartezian:

$$A \times C$$

care se definește astfel:

perechea

$$\langle a, c \rangle$$

apartine relației R, dacă există un element b, în mulțimea B, așa încât perechea:

$$\langle a, b \rangle$$

apartine relației P, iar perechea:

$$\langle b, c \rangle$$

apartine relației Q.

Deci, dacă expresiile:

$$aPb \text{ și } bQc$$

sunt adevărate, atunci este adevărată și expresia:

$$aRc$$

Fie MR matricea care reprezintă relația R. Dacă MP este o matrice de ordinul

m^*n , iar MQ o matrice de ordinul $n*s$, atunci matricea MR este de ordinul $m*s$.
Matricea relației MX ($X = P, Q$ sau R) este păstrată pe biți în tabloul unidimensional tmx ($x = p, q$ sau r).

FUNCȚIA BX45

```

unsigned belem(char [],int,int,int);

void sbelem(char [],int,int,int);

void prodrel(char tmp[],char tmq[],char tmr[],int m,int n,int s)
/* - calculeaza relatia:
   R = P.Q
   - relatia P se reprezinta prin matricea MP;
   - relatia Q se reprezinta prin matricea MQ;
   - relatia R se reprezinta prin matricea MR;
   - matricea MP este de ordinul  $m^*n$  si se păstreaza pe biti in tabloul tmp;
   - matricea MQ este de ordinul  $n*s$  si se păstreaza pe biti in tabloul tmq;
   - matricea MR este de ordinul  $m*s$  si se păstreaza pe biti in tabloul tmr. */
{
    int i,j,k;

    /* se anuleaza toate elementele matricei MR */
    for(i=0; i<=m*s/8; i++) tmr[i] = 0;

    /* se seteaza elementele matricei MR */
    for(j=0; j<n; j++)
        for(i=0; i<m; i++)
            if(belem(tmp,n,i,j))

                /* MP[i][j]=1 */
                for(k=0; k< s; k++ )
                    if(belem(tmq,s,j,k))

                        /* MQ[j][k]=1 */
                        /* elementul MR[i][j] se seteaza la valoarea 1 */
                        sbelem(tmr,s,i,k);
}

void sbelem(char ta[],int n, int i,int j)
/* - seteaza la 1 valoarea elementului a[i][j];
   - matricea a este păstrata in tabloul ta pe biti folosind relația de liniarizare;
   - matricea a are n coloane. */
{
    ta[(i*n+j)/8] |= 1<< 7 -(i*n+j)%8;
}

```

Observație:

Expresia:

$$(1) 1<<7 - (i*n+j)%8$$

deplasează unu spre stînga cu un număr de poziții binare egal cu valoarea

expresiei:

$$(2) 7 - (i*n+j)%8$$

Se observă că dacă $i*n+j$ este multiplu de 8, atunci expresia (2) are valoarea 7 și deci unu se deplasează cu 7 poziții binare spre stînga. Se obține:

10000000

deci, bitul cel mai semnificativ al octetului rezultat devine egal cu 1.

Dacă restul împărțirii lui $i*n+j$ la 8 este 1, atunci expresia (2) are valoarea 6, iar expresia (1) generează valoarea:

1000000

Deci, în acest caz, bitul de ordinul 6 al rezultatului devine egal cu 1.

În mod analog, dacă $(i*n+j)%8 = 2$, bitul de ordinul 5 al rezultatului devine egal cu 1, și aşa mai departe.

Dacă $(i*n+j)%8 = 7$, atunci expresia (2) are valoarea zero, iar expresia (1) are valoarea 1. Deci, bitul de ordinul zero al rezultatului va deveni egal cu 1.

10.46 Să se scrie o funcție care citește indicii elementelor $a[i][j]$ diferite de zero ale matricii a care reprezintă o relație binară. Matricea este de ordinul m^*n și elementele se păstrează pe biți, în tabloul ta unidimensional, folosind relația de liniarizare.

La început funcția citește valorile lui m și n . Apoi se citesc indicii i și j pentru fiecare element nenul al matricei a .

FUNCȚIA BX46

```

void citrel(char ta[], int max,int *m, int *n)
/* - citește pe m și n;
   - citește indicii elementelor a[i][j] nenule ale matricei a ce reprezintă o relație binară;
   - păstrează matricea pe biti în tabloul ta folosind relația de liniarizare.
*/
{
    int i,j;

    do ( /* citește pe m și n */
        *m=*n=0;
        if(pcit_int_lim("numar linii m= ",1,max,m) == 0 ) {
            printf("s-a tastat EOF\n");
            exit(1);
        }
        if(pcit_int_lim("numar coloane n= ",1,max,n) == 0 ) {
            printf("s-a tastat EOF\n");
            exit(1);
        }
        if(*m * *n < max) break;
        printf("produsul m*n= %d\n",*m * *n );
        printf("depaseste valoarea maxima= %d\n",max);
    } while(1);
}

```

```

for(i=0; i<= *m * *n/8; i++) ta[i] = 0;

/* citeste indicii elementelor nenule si seteaza bitii corespunzatori in tabloul ta */
for( ; ; ) {
    if(pcit_int_lim("linia i= ", 1, *m, &i) == 0) return;
    if(pcit_int_lim("coloana j= ", 1, *n, &j) == 0) return;

    /* seteaza elementul a[i-1][j-1] */
    ta[((i-1)* *n + j-1)/8] |= 1 << ((i-1) * *n + j-1) % 8;
}
}

```

- 10.47 Să se scrie un program care calculează și afișază compunerea a două relații definite pe mulțimea A, apoi stabilește dacă relațiile respective sunt de echivalență.

Relațiile se introduc folosind funcția *citrel* definită în exercițiul precedent.

Apoi, se apelează funcția *prodrel* care compune relațiile respective. În final, se stabilește dacă cele trei relații sunt de echivalență.

PROGRAMUL BX47

```

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

#include "bviii2.cpp" /* pcit_int */
#include "bviii3.cpp" /* pcit_int_lim */
#include "bx44.cpp" /* relechiv */
#include "bx45.cpp" /* prodrel */
#include "bx46.cpp" /* citrel */

void afisrel(char *, char [], int );

#define MAX 128

main() /*- calculeaza si afiseaza compunerea a doua relatii;
         - stabileste dacă relațiile sunt de echivalență.*/
{
    char a[MAX * (MAX/8)], b[MAX * (MAX/8)], ab[MAX * (MAX/8)];
    int m, n, p, q;

    /* defineste relația reprezentată prin tabloul a */
    citrel(a, MAX*MAX, &m, &n);

    /* defineste relația reprezentată prin tabloul b */
    citrel(b, MAX*MAX, &p, &q);
    if(!(m==n& n==p & p==q)) {
        printf("se cere ca relatiile sa fie\
               reprezentate prin matrice patratice\n");
        exit(1);
    }
}

```

```

/* se calculeaza compunerea relațiilor a și b:
   ab = a.b */
prodrel(a, b, ab, m, n, p);

/* afiseaza relația reprezentată prin a */
afisrel("relatia a ", a, m);

/* afiseaza relația reprezentată prin b */
afisrel("relatia b ", b, n);

/* afiseaza compunerea relațiilor a și b */
afisrel("relatia ab= a.b ", ab, m);
/* stabilește care dintre relații este de echivalență */
if(relechiv(a, m))
    printf("relatia a este de echivalenta\n");
else
    printf("relatia a nu este de echivalenta\n");
if(relechiv(b, n))
    printf("relatia b este de echivalenta\n");
else
    printf("relatia b nu este de echivalenta\n");
if(relechiv(ab, m))
    printf("relatia ab este de echivalenta\n");
else
    printf("relatia ab nu este de echivalenta\n");
} /* sfirsit main */

void afisrel(char *p, char ta[], int n)
/* afiseaza relația pastrată pe biti prin linierizare în tabloul ta */
{
    int i, j, k, m;

    /* afiseaza un antet */
    printf("\n\n\t\t%s\n", p);
    for(i=0; i< n; i++) {
        printf("linia: %d\n", i+1);
        k=1, m=0;
        for(j=0; j<n; j++, k++) {
            printf("%d ", ta[(i*n+j)/8]>> 7-(i*n+j)%8&1);
            if( k == 30 ) {
                printf("\n");
                k = 0;
            }
            m++;
            if(m%20 == 0 ) {
                printf("actionati o tasta pentru a \
                       continua\n");
                getch();
            }
        }
        printf("actionati o tasta pentru a continua\n");
        getch();
    }
}

```

10.6. Tipul enumerare

Tipul *enumerare* permite programatorului să folosească nume sugestive pentru valori numerice. De exemplu, în locul numărului unei luni calendaristice este mai sugestiv să folosim denumirea lunii respective sau eventual o prescurtare:

- | | |
|------------|---------------------------------------|
| <i>ian</i> | - Pentru ianuarie în locul cifrei 1. |
| <i>feb</i> | - Pentru februarie în locul cifrei 2. |
- și aşa mai departe.

Un alt exemplu se referă la posibilitatea de a utiliza cuvintele FALSE și ADEVARAT pentru valorile 0 respectiv 1. În felul acesta se obține o mai mare claritate în programele sursă, deoarece valorile numerice sunt înlocuite prin sensurile atribuie lor într-un anumit context.

În acest scop se utilizează tipul *enumerare*. Un astfel de tip se declară într-un format asemănător cu cel utilizat în cadrul structurilor. Un prim format general este:

```
enum nume { nume0,nume1,nume2,...,numek } d1,d2,...,dn;
```

unde:

- | | |
|-------------------------|---|
| <i>nume</i> | - Este numele tipului de enumerare introdus prin această declarație. |
| <i>nume0,nume1,...,</i> | - Sunt nume care se vor utiliza în continuare în locul valorilor numerice și anume <i>numei</i> are valoarea <i>i</i> . |
| <i>numek</i> | - Sunt date care se declară de tipul <i>nume</i> . |
| <i>d1,d2,...,dn</i> | - Aceste date sunt similare cu datele de tip <i>int</i> . |

Ca și în cazul structurilor, în declarația de mai sus nu sunt obligatorii toate elementele. Astfel, poate lipsi nume, dar atunci va fi prezent cel puțin *d1*. De asemenea, poate lipsi în totalitate lista *d1,d2,...,dn*, dar atunci va fi prezent *nume*. În acest caz, se vor defini ulterior date de tip *nume* folosind un format de forma:

```
enum nume d1,d2,...,dn;
```

Exemple:

1.

```
enum { ileg,ian,feb,mar,apr,mai,iun,iul,aug,sep,oct,nov,dec } luna;
```

Prin această declarație, numărul lunii poate fi înlocuit prin denumirea prescurtată a lunii respective. De exemplu, o atribuire de forma:

```
luna = 3
```

se poate înlocui cu una mai sugestivă:

```
luna = mar
```

deoarece, conform declarației de mai sus, *mar* are valoarea 3.

În mod analog, o expresie de forma:

```
luna == 7
```

este identică cu expresia:

```
luna == iul
```

Dacă în locul declarației de mai sus s-ar fi utilizat declarația de tip enumerare:

```
enum dl { ileg,ian,feb,mar,apr,mai,iun,iul,aug,sep,oct,nov,dec };
```

atunci putem declara ulterior data *luna* de tip *dl* astfel:

```
enum dl luna;
```

Data *luna* declarată în acest fel este o dată identică cu data *luna* declarată la început.

2. Fie tipul enumerare Boolean declarat astfel:

```
enum Boolean { false, true};
```

Declarăm data *bisect* de tip Boolean:

```
enum Boolean bisect;
```

Atribuirea:

```
bisect = an%4 == 0 && an%100 != an%400 == 0;
```

atribuie variabilei *bisect* valoarea 1 sau 0, după cum anul definit de variabila *an* este bisect sau nu (se presupune că anul aparține intervalului [1600,4900]).

În continuare se pot folosi expresii de forma:

```
bisect == false
```

sau

```
bisect == true
```

3. Fie/fo funcție care returnează numai două valori: 0 sau 1. O astfel de funcție se consideră, de obicei, ca returnează o valoare booleană. Atunci, antetul ei poate fi definit astfel:

```
enum Boolean f( ... )
```

În continuare se pot folosi expresii de forma:

```
f(...) == false
```

sau

```
f(...) == true
```

La declararea tipurilor enumerare se poate folosi cuvântul cheie *typedef*, ca și în cazul tipurilor utilizator definite prin *struct*. În acest caz vom utiliza o declarație de forma:

```
typedef enum nume { nume0,nume1,...numek } nume_tip;
```

În continuare, se pot declara mai simplu date de tipul enumerare *nume* folosind:

```
nume_tip d1,d2,...,dn;
```

În acest caz *nume* este optional și de obicei lipsește. Folosind această convenție, exemplele de mai sus pot fi scrise astfel:

```
typedef enum { ieq, ian, ieb, mar, apr, mai, iun,
                iul, aug, sep, oct, nov, dec } DL;
DL_tuna;
typedef enum { false, true} Boolean;
Boolean bisect;
```

sau antetul funcției f:

```
Boolean f(...)
```

Tipul enumerare poate fi declarat impunind valori altfel decit cele care rezulta implicit. În acest caz, *numei* se va înlocui cu:

```
numei = eci
```

unde:

eci - Este o expresie constantă.

Numele *numei* va avea ca valoare, valoarea expresiei *eci*. Dacă numelui următor nu î se atribuie o valoare, atunci acesta va avea ca valoare, valoarea lui *numei* mărit cu 1.

Folosind aceasta observație, modificăm tipul DL astfel:

```
typedef enum { ian=1,feb,mar,apr,mai,iun,iul,aug,sep,oct,nov,dec } DL;
```

Menționam că datele de acest tip nu sunt supuse la controale din partea compilatorului C și de aceea se pot scrie expresii care să nu corespundă scopului pentru care s-au definit datele respective.

De exemplu, fie:

```
DL d1,d2,d3;
```

Expresiile de mai jos nu sunt interpretate ca eronate de compilator:

```
d3 = d1 + d2;
d3 = d1 * d2;
d3 = d1/d2; etc.
```

Aceasta, deoarece datele de tip enumerare sunt tratate ca simple date de tip *int*.

10.7. Date definite recursiv

Fie declarația:

```
struct nume {
    declaratii
};
```

Declarațiile incluse între acolade definesc componentele datelor structurate de tip *nume*, tip definit prin declarația struct. Aceste declarații pot defini date de diferite tipuri predefinite sau utilizator, dar diferite de tipul *nume*. În schimb, se pot defini pointeri spre date de tipul *nume*. Deci, o declarație de forma:

```
struct nume {
    declaratii
    struct nume *nume1;
    declaratii
};
```

este corectă, în timp ce declarația:

```
struct nume {
    declaratii
    struct nume nume1;
    declaratii
};
```

este eronată.

Un tip utilizator, se spune că este *direct recursiv* dacă el are cel puțin o componentă care este de tip *pointer spre el însuși*.

Exemplu:

```
struct tnod {
    char cuvint[100];
    int nr;
    struct tnod *urm;
};
```

Tipul utilizator *tnod* conține 3 componente:

- tabloul *cuvint* de tip *char* de 100 elemente;
- variabila *nr* de tip *int*;
- pointerul *urm* spre tipul *tnod*.

tnod este un tip direct recursiv deoarece conține un pointer spre el însuși.

În continuare, putem defini obișnuit date de tipul *tnod*.

Evident, tipurile direct recursive pot fi denumite folosind construcția *typedef*.

Tipul de mai sus îl vom numi TNOD folosind declarația:

```
typedef struct tnod {
    char cuvint[100];
    int nr;
    struct tnod *urm;
} TNOD;
```

În continuare putem defini date folosind tipul TNOD:

```
TNOD nod, *p;
```

O altă posibilitate în definirea tipurilor utilizator este acela în care un tip *t1* conține un pointer spre tipul *t2*, iar acesta, la rindul lui, conține un pointer spre tipul *t1*. În acest caz se spune că tipul *t1* este *indirect recursiv*.

Să observăm că la declararea tipului *t1*, se utilizează o declarație de forma:

```
struct t2 *nt2;
```

Aceasta este posibil dacă tipul *t2* este definit înainte de tipul *t1*. Să presupunem că acest lucru are loc. Dar la declararea tipului *t2* constatăm că acesta conține o declarație de forma:

```
struct t1 *nt1;
```

Aceasta la rindul ei poate fi scrisă dacă înaintea declarației tipului *t2* este prezentă declarația care definește tipul *t1*. Se ajunge în felul acesta la o contradicție. Pentru a o elimina, s-a introdus în limbajul C declarația de tip incompletă.

Aceasta are formatul:

```
struct nume;
```

Această declarație poate fi plasată într-un fișier sursă înainte de a defini tipul *nume* sau în orice fișier sursă în care tipul *nume* încă nu este definit. După declarația de tip incompletă, se pot declara pointeri spre tipul respectiv. În felul acesta, tipurile *t1* și *t2* de mai sus, se vor declara astfel:

```
struct t1; /* - declaratie incompleta pentru tipul t1;
            - in continuare se poate declara tipul t2 */
struct t2 {
    declaratii
    struct t1 *nt1;
    declaratii
};
struct t1 { /* declaratia completa a lui t1 */
    declaratii
    struct t2 *nt2;
    declaratii
};
```

Un tip se spune că este *recursiv* dacă el este direct sau indirect recursiv. O dată este *recursivă* dacă ea este o dată de un tip recursiv.

În general, un tip *t* este *indirect recursiv* dacă există un sir de tipuri *ti*:

t1,t2,t3,...,tn

însă înainte tipul *ti* conține cel puțin o componentă care este pointer spre tipul *tj* ($j = i + 1$), pentru $i = 1, 2, \dots, n-1$, iar *t1* și *tn* sunt ambele de tipul *t*.

Datele recursive se utilizează într-o serie de aplicații bazate pe definirea și prelucrarea listelor, arborilor, tabelelor de dispersie etc.

Ele se folosesc cu succes în prelucrarea dinamică a datelor.

În multe aplicații se utilizează date structurate de un anumit tip, care se repetă de un număr variabil de ori, număr care se precizează la fiecare execuție a programului.

În astfel de situații se poate defini un tablou de structuri a cărui număr de elemente să fie egal cu o valoare maximă precizată de utilizator. Acest mod de lucru nu este cel mai indicat atunci cind numărul datelor diferă mult de la o execuție la alta. De asemenea, pot să se ivescă situații în care acest maxim este greu de evaluat, ca să nu mai vorbim de faptul că el poate fi evaluat greșit. De aceea, în astfel de cazuri este mult mai util să se păstreze datele respective în memoria *heap*, rezervându-se memorie pentru ele la execuție, pe măsură ce este nevoie. Prelucrarea datelor respective implică posibilitatea de a trece simplu de la o dată la alta. În cazul tablourilor, aceasta se asigură cu ajutorul indicilor. În noua situație se poate trece de la o dată păstrată în memoria *heap* la o altă dată de același tip, dacă se utilizează un pointer spre tipul comun datelor respective și care să fie componenta a datei respective. Deci, se ajunge la un tip de forma:

```
typedef struct nume {
    declaratii
    struct nume *urm;
    declaratii
} nume_tip;
```

În felul acesta se ajunge la necesitatea de a utiliza date recursive.

Gestiona dinamică a lor mai are încă un avantaj și anume: astfel de date pot fi eliminate ușor din memoria *heap* cind nu este nevoie de ele și zona eliberată în acest fel poate fi realocată altor date de același tip sau în alte scopuri.

Exerciții:

10.48 Să se scrie o funcție care citește un cuvânt și-l păstrează în memoria *heap*. Prin *cuvânt* înțelegem o succesiune de litere mici sau mari.

Funcția returnează adresa de început a zonei din memoria *heap* în care se păstrează cuvântul citit sau zero la întâlnirea sfîrșitului de fișier.

FUNCȚIA BX48

```
char *citcuv()
/* - citește un cuvânt și-l păstrează în memoria heap;
   - returnează pointerul spre cuvântul respectiv sau zero la sfârșit de fișier. */
```

```

{
    int c,i;
    char t[255];
    char *p;

    /* salt peste caractere care nu sunt litere */
    while((c=getchar()) < 'A' || (c > 'Z' && c < 'a') || c > 'z')
        if( c == EOF ) return 0; /* s-a tastat EOF */

    /* se citește cuvintul și se păstrează în t */
    i = 0;
    do {
        t[i] = c;
    } while((c=getchar()) >= 'A' && c<='Z'||c>= 'a' && c <= 'z');
    if( c == EOF ) return 0;
    t[i+1] = '\0';

    /* se păstrează cuvintul în memoria heap */
    if((p = (char *)malloc(i)) == 0) {
        printf("memorie insuficienta\n");
        exit(1);
    }
    strcpy(p,t);
    return p;
}

```

- 10.49 Sa se scrie o funcție care citește cuvintele dintr-un text și determină numărul de apariții al fiecărui cuvint din textul respectiv.

Accesta problemă poate fi rezolvată în mai multe moduri. Prezentăm mai jos o metodă simplă, care însă nu este și eficientă. Metode mai eficiente vor fi prezentate în alte capitole.

Înainte de toate să observăm că problema se poate rezolva folosind un tablou de pointeri spre *char*, dacă putem indica o valoare maximă pentru numărul de cuvinte diferite din text. În acest caz se definește tabloul:

```
char *tp[maxcuv];
```

și un indice *itp* care are ca valoare indicele primului element liber din tablou.

De asemenea, vom utiliza un tablou pentru a număra aparițiile fiecărui cuvint:

```
int nc[maxcuv];
```

Procesul de calcul se realizează astfel:

1. *itp* = 0.
2. Se apelează funcția *citcuv* pentru a citi cuvintul curent:
 $p = \text{citcuv}();$
Dacă *p* = 0, s-a întâlnit sfîrșitul de fișier și procesul de calcul se întrerupe.
Altfel se trece la pasul următor.
3. Se stabilește dacă cuvintul citit (spre care pointează *p*) a fost deja întâlnit în text.

În acest scop se compară cuvintele spre care pointează elementele:

$tp[0], tp[1], \dots, tp[itp-1]$
cu cel spre care pointează *p*.

În caz afirmativ, se trece la pasul 4, altfel la pasul 5.

4. Dacă cuvintul spre care pointează *p* coincide cu cel spre care pointează $tp[j]$, atunci se incrementează $nc[j]$ (numărul de apariții al cuvintului respectiv); apoi se elimină cuvintul, citit la punctul 2, din memoria *heap* și procesul de calcul se reia de la pasul 2.
5. Deoarece cuvintul spre care pointează *p* nu a fost încă întâlnit în textul de intrare, pointerul *p* se păstrează și numărul de apariții al cuvintului respectiv se face egal cu 1:

```
tp[itp] = p;
nc[itp] = 1;
```

apoi se mărește *itp* cu o unitate:

```
itp = itp + 1;
```

procesul de calcul se reia de la pasul 2.

Funcția utilizează un tablou de structuri de tipul T declarat astfel:

```
typedef struct {
    char *tp;
    int nc;
} T;
```

Ea are un parametru care este un tablou de tipul T. Un alt parametru al ei este *maxcuv*.

Funcția returnează numărul cuvintelor distincte citite (*itp*).

FUNCȚIA BX49

```

int freqcuv(T t[], int maxcuv)
/* - citește un text și determină frecvența de apariție a fiecarui cuvint citit;
 - se admit cel mult maxcuv cuvinte distincte;
 - funcția returnează numarul cuvintelor distincte întâlnite în text */
{
    char *p;
    int itp;
    int i;

    for(itp = 0; itp < maxcuv; ) {

        /* citește cuvintul curent */
        if((p = citcuv()) == 0)

            /* s-a întâlnit sfîrșitul de fisier */
            return itp;
    }
}

```

```

/* se stabilește dacă cuvintul citit a fost deja întlnit */
for(i = 0; i < itp; i++)
    if(strcmp(t[i].tp,p) == 0) break;
if( i < itp ) {

    /* cuvint deja întlnit */
    t[i].nc++;
    free(p);
}
else { /* cuvint nou */
    t[itp].tp = p;
    t[itp++].nc = 1;
}
return itp;
}

```

10.50 Să se scrie un program care afișează frecvența de apariție a cuvintelor unui text.

PROGRAMUL BX50

```

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>
#include <alloc.h>

typedef struct {
    char *tp;
    int nc;
} T;

#include "bx48.cpp"
#include "bx49.cpp"

#define MAXCUV 100

main() /* afiseaza frecventa de aparitie a cuvintelor unui text */
{
    T tab[MAXCUV];
    int ncuv,i;

    ncuv = freqcuv(tab,MAXCUV);
    for(i=0; i < ncuv; i++) {
        printf("%s\t%d\n",tab[i].tp,tab[i].nc);
        if((i+1)%23 == 0) {
            printf("actionati o tasta pentru a continua\n");
            getch();
        }
    }
}

```

10.51 Să se modifice funcția *freqcuv* definită în exercițiul 10.49, în aşa fel încit

în loc de tabloul de tip T să se utilizeze date de tip recursiv.

În acest caz vom folosi tipul recursiv:

```

typedef struct tr {
    char *tp;
    int nc;
    struct tr *urm;
} TR;

```

Pentru fiecare cuvint se va păstra, în memoria heap, o dată de tip TR. Pointerului *urm* îi se atribuie adresa de început a zonei din memoria heap în care se păstrează cuvintul următor citit din textul de intrare. El are valoarea zero pentru data corespunzătoare ultimului cuvint citit.

Pentru a gestiona simplu datele păstrate în memoria heap, va fi util să existe două variabile de tip pointer spre TR, una să pointeze spre prima dată păstrată în memoria heap, iar cealaltă spre ultima. Vom numi *prim* și *ultim* aceste variabile.

Funcția realizează pașii de mai jos:

1. *prim* = *ultim* = 0;
2. Se apelează funcția *citcuv* pentru a citi cuvintul curent:
 $p = \text{citcuv}();$
Dacă $p = 0$, atunci se revine din funcție cu valoarea variabilei *prim*; altfel se trece la pasul următor.
3. Se stabilește dacă cuvintul citit (spre care pointează *p*) a fost deja întlnit în text.

În acest scop se compara cuvintele spre care pointează pointerul *tp* din datele de tip TR păstrate în memoria heap, cu cel spre care pointează *p*.

În caz afirmativ se incrementează număratorul cuvintului respectiv, se eliberează zona ocupată de cuvintul curent citit și procesul se reia de la pasul 2.

Altfel, se rezervă zonă pentru o dată de tip TR și fie *q* adresa ei de început. Pointerului *urm*, corespunzător ultimei date de tip TR păstrate în memoria heap, îi se atribuie valoarea lui *q*.

Apoi *nc* = 1 și *urm* = 0 pentru data spre care pointează *q*.

Cum acesta devine ultima dată păstrată în memoria heap, se face *ultim=q* și apoi se revine la pasul 2.

FUNCȚIA BX51

```

TR *pfreqcuv()
/* citeste un text si păstreaza in memoria heap date de tip TR pentru cuvintele distincte din text,
determina frecventa acestora si returneaza pointerul spre data corespunzatoare primului
cuvint din text */
{
    TR *prim, *ultim,*q;
    char *p;

    prim = ultim = 0;
}

```

```

tor( ; ; ) {
    /* citește cuvintul curent */
    if ((p = citouv()) == 0)

        /* s-a intilnit sfîrșitul de fisier */
        return prim;

    /* stabilește dacă cuvintul curent citit a fost deja intilnit în text */
    q = prim; /* căutarea se face începînd cu primul cuvînt */
    while ( q ) { /* dacă q != 0 înseanîă ca pointeaza spre o
                    data din memoria heap */
        if (strcmp( q->tp, p ) == 0)
            break; /* cuvintul a mai fost intilnit în text */

    /* - se trece la data corespunzatoare cuvintului urmator citit din textul de intrare;
       - adresa acestei date este valoarea pointerului urm al datei spre care pointeaza q;
       - aceasta se poate exprima cu ajutorul expresiei:
           q->urm;
       - aceasta adresa se atribuie lui q și în felul acesta se poate relua ciclul */

        q = q->urm;
    }
    if (q) { /* dacă q != 0, cuvintul a mai fost intilnit în textul de intrare */
        q->nc++; /* se incrementează numărul de aparitii al cuvintului */
        free(p); /* se eliberează zona ocupată de cuvint */
    }
    else { /* cuvintul este nou */
        /* se rezerva zona pentru o data de tip TR */
        q = (TR *) malloc(sizeof(TR));
        if (q == 0) { /* nu se poate aloca memoria heap */
            printf("memorie insuficienta\n");
            exit(1);
        }

        /* se pastrează pointerul spre cuvintul nou */
        q->tp = p;

        /* nc = 1 */
        q->nc = 1;
        if (ultim) /* dacă ultim != 0, cuvintul curent nu este
                     primul cuvint citit */
        /* ultim pointează spre data din memoria heap corespunzatoare
           cuvintului precedent pastrat, deci
           ultim->urm = q
        */
        ultim->urm = q;
    }
    ultim->urm = q;

    /* q->urm = 0, deoarece data spre care pointeaza q corespunde
       ultimului cuvint citit și pastrat în memoria heap;
       - din același motive
           ultim = q
    */
}

```

```

q->urm = 0;
ultim = q;
if (prim == 0) /* cuvintul curent este primul cuvint citit */
    prim = q;
} /* sfîrșit else */
} /* sfîrșit for */
}

```

10.52 Să se scrie un program care citește și afișează frecvența de apariție a cuvintelor din textul citit, folosind funcția *pfreqcuv* definită în exercițiul precedent.

În programul de față nu mai este necesar să ne limităm la 100 de cuvinte, cum am făcut în programul din exercițiul 10.50. Datele se păstrează în memoria heap pe măsură ce este nevoie de ele.

PROGRAMUL BX52

```

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>
#include <alloc.h>

typedef struct tr {
    char *tp;
    int nc;
    struct tr *urm;
} TR;

#include "bx48.cpp" /* citouv */
#include "bx51.cpp" /* pfreqcuv */

main() /* afișează frecvența de apariție a cuvintelor unui text */
{
    TR *p;
    int i;

    p = pfreqcuv();
    i = 1;

    while( p ) {
        printf("%s\t%d\n", p->tp, p->nc);
        if(i%23 == 0) {
            printf("actionati o tasta pentru a continua\n");
            getch();
        }

        i++;
        p = p->urm;
    }
}

```

Observație:

Datele de tip TR, păstrate în memoria *heap*, sunt structuri recursive care formează o mulțime ordonată. Există o *primă* structură și fiecare structură are un element următor, exceptând ultima. De aceea, între elementele acestei mulțimi de structuri este stabilită o relație de ordine. Relația respectivă se definește printr-un pointer care este componentă a tipului comun al acestor structuri.

O astfel de mulțime se spune că formează o *listă simplu înlățuită*.

11. LISTE

Lista este o mulțime *dinamică*, înțelegind prin aceasta că ea are un număr variabil de elemente. La început lista este o mulțime vidă. În procesul execuției programului se pot adăuga elemente noi listei și totodată pot fi eliminate diferite elemente din listă de care nu mai este nevoie.

Elementele unei liste sunt date de un același tip. În forma cea mai generală, tipul comun elementelor unei liste este un tip utilizator. Deci, elementele unei liste sunt structuri de un același tip. Acest fapt ne conduce la ideea de a organiza o listă sub formă de tablou. Acest lucru este posibil, dar de obicei nu este eficient, deoarece lista are o natură dinamică, ori tablourile în limbajul C nu sunt dinamice. La compilare este necesar să se definească un maxim pentru numărul elementelor unui astfel de tablou, maxim care uneori nu se poate stabili dinainte. Se poate adopta o soluție de compromis dacă se păstrează elementele listei în memoria *heap*, iar într-o memorie statică sau pe stivă se alocă un tablou de pointeri spre elementele listei. În acest caz, elementele tabloului de pointeri ocupă fiecare 16 biți sau 32 de biți dacă pointerii respectivi sunt de tip *far*. Tabloul de pointeri se va dimensiona la un număr de elemente suficient de mare. Avantajul păstrării elementelor listei în memoria *heap* rezultă din faptul că, supradimensionarea tabloului de pointeri nu conduce la un consum prea mare de memorie deoarece elementele tabloului sunt pointeri și nu însăși elementele listei. În același timp, memoria *heap* este gestionată optim, deoarece ea conține, în fiecare moment al execuției programului, numai elementele necesare din listă. Cu toate acestea, există situații în care este dificil a evalua numărul maxim al elementelor unei liste, precum și cazuri cind numărul lor diferă mult de la execuție la execuție. De aceea, apare problema de a organiza altfel mulțimile de tip listă. Se obișnuiește să se ordoneze elementele unei liste folosind pointeri care intră în compunerea elementelor listei. Datorită acestor pointeri, elementele listei devin structuri recursive. Listele organizate în acest fel se numesc *liste înlățuite*.

În concluzie, o mulțime dinamică de structuri recursive de același tip, pentru care sunt definite una sau mai multe relații de ordine cu ajutorul unor pointeri din compunerea structurilor respective, se numește *listă înlățuită*.

Elementele unei liste, de obicei, se numesc *noduri*.

Dacă între nodurile unei liste există o singură relație de ordine, atunci lista se numește *simplu înlățuită*. În mod analog, lista este *dublu înlățuită* dacă între nodurile ei sunt definite două relații de ordine.

În general, vom spune că o listă este *n-inlățuită* dacă între nodurile ei sunt definite *n* relații de ordine.

În legătură cu listele înlățuite se au în vedere unele operații de interes general:

- a. crearea unei liste înlățuite;
- b. accesul la un nod oarecare al unei liste înlățuite;

- c. inserarea unui nod intr-o lista înlanțuită;
- d. ștergerea unui nod dintr-o lista înlanțuită;
- e. ștergerea unei liste înlanțuite.

11.1. Lista simplu înlanțuită

Așa cum s-a arătat mai sus, intre nodurile unei liste simplu înlanțuite este definită o singură relație de ordonare. De obicei aceasta relație este cea de succesor, adică fiecare nod conține un pointer a cărui valoare reprezintă adresa nodului *următor* din lista. În mod analog se poate defini relația de *precedent*. În cele ce urmează ne vom mărgini numai la liste simplu înlanțuite pentru care nodurile satisfac relația de succesor.

Intr-o astfel de listă există totdeauna un nod și numai unul care nu mai are următor (succesor), precum și un nod care nu este următorul (succesorul) nici unui alt nod. Aceste noduri formează *capetele* listei simplu înlanțuite.

Pentru a gestiona nodurile unei liste simplu înlanțuite, vom utiliza doi pointeri spre cele două capete. Numim *prim* pointerul spre nodul care nu este următorul nici unui nod al listei și cu *ultim* pointerul spre nodul care nu are următor (succesor) în listă. Acești pointeri vor fi utilizati în toate funcțiile care le vom avea în vedere în prelucrarea listelor simplu înlanțuite. Ei pot fi definiți fie ca variabile globale, fie ca parametri pentru funcțiile de prelucrare a listei. Se alege alternativa cu variabile globale în cazul în care nu se gestionează mai multe liste simplu înlanțuite în același program. Dimpotrivă, pointerii respectivi se vor transfera prin parametri în cazul în care programul gestionează mai multe liste simplu înlanțuite.

În cele ce urmează vom considera cazul în care pointerii *prim* și *ultim* sunt variabile globale.

Tipul unui nod într-o listă simplu înlanțuită se poate defini folosind o declarație de forma:

```
typedef struct tnod {
    declarării
    struct tnod *urm;
} TNOD;
```

Pointerii *prim* și *ultim* se declară în afara oricărei funcții prin:

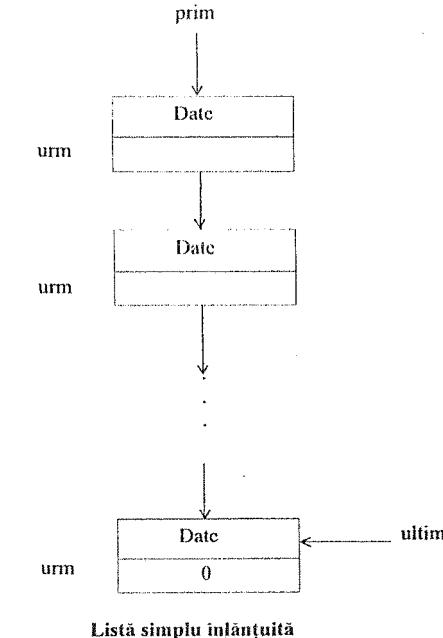
```
TNOD *prim,*ultim;
```

De obicei, ei vor fi declarati înaintea definirii funcției *main* a programului.

Pointerul *urm* definește relația de succesor pentru nodurile listei. Pentru fiecare nod, el are ca valoarea adresa nodului următor din listă. Excepție de la această regulă o constituie nodul spre care pointează variabila *ultim*. În acest caz *urm* are valoarea zero (pointerul nul).

În paragrafele următoare vom construi funcții care să realizeze operațiile a-e,

amintite mai sus, pentru listele simplu înlanțuite.



11.1.1. Crearea unei liste simplu înlanțuite

La crearea unei liste simplu înlanțuite se realizează următoarele:

1. Se inițializează pointerii *prim* și *ultim* cu valoarea zero, deoarece lista început este vidă.
2. Se rezervă zonă de memorie în memoria *heap* pentru nodul curent.
3. Se încarcă nodul curent cu datele curente, dacă există și apoi se trece la pasul 4.
4. Aftfel lista este creată și se revine din funcție.
5. Se atribuie pointerului *ultim* → *urm* adresa din memoria *heap* a nodului curent, dacă lista nu este vidă. Aftfel se atribuie lui *prim* această adresă.
6. Se atribuie lui *ultim* adresa nodului curent.
7. *ultim* → *urm* = 0
7. Procesul se reia de la punctul 2 de mai sus pentru a adăuga un nod nou la listă.

Pentru a încărca datele curente în nod (punctul 3) vom apela o funcție care este specifică aplicației. Această funcție poate să aibă un nume dinainte precizat sau să fie transferată printr-un parametru la funcția de creare a listei. Acest parametru va fi un pointer spre funcția respectivă. Soluția cu parametru se alege de obicei cind programul prelucrăza mai multe liste. În cazul de față am presupus că programul prelucrăza o singură listă și de aceea în loc să utilizăm un parametru pointer, vom considera că datele se încarcă prin funcția de nume *incnod*. Ea returnează una din următoarele trei valori: 0, 1 și -1 și anume:

- 0 - La eroare.
- 1 - La încărcare normală a datelor în nodul curent.
- 1 - Cind nu mai sunt date de încărcat în nod.

De exemplu, dacă funcția *incnod* citește datele pe care le încarcă în nod dintr-un fișier, atunci ea va returna valoarea -1 la întâlnirea sfîrșitului de fișier (nu mai sunt date de încărcat în nod).

Funcția *incnod* are un singur parametru și anume adresa din memoria *heap* rezervată pentru nodul curent, deci acest parametru este un pointer spre tipul comun nodurilor structurii, adică spre TNOD.

Din cele de mai sus rezultă că funcția *incnod* are prototipul:

```
int incnod(TNOD *p);
```

O altă funcție specifică aplicației concrete și care se apelează la crearea listei este cea care eliberează zona de memorie rezervată nodului curent. Vom numi *elibnod* această funcție. Ea are prototipul:

```
void elibnod(TNOD *p);
```

Această funcție se apelează după un apel al funcției *incnod* în care aceasta nu a încărcat date în nodul curent (s-a revenit din ea cu valoarea 0 sau -1). În acest caz există zonă de memorie rezervată în memoria *heap* pentru nodul curent, dar la apelul funcției *incnod* nu s-au încărcat date ori s-au încărcat date parțial. De aceea, în acest caz vom elibera zona de memorie rezervată apelând *elibnod*.

Funcția pentru crearea listei simplu înlănțuite o numim *crelist*. Ea returnează una din valorile 0 sau -1 și anume:

- 0 - La eroare.
- 1 - La crearea normală a listei.

Tinind seama de cele de mai sus, definim funcția *crelist* astfel:

```
int crelist() /*- creaza o lista simplu înlanțuită;
               - returneaza:
                 0 - la eroare;
                 -1 - creare normală.*/

{
    extern TNOD *prim, *ultim;
    int i, n;
```

```
TNOD *p;
n = sizeof(TNOD);
prim = ultim = 0; /* initial lista este vida */

/* se rezerva n octeti pentru nod în memoria heap */
for (;;) {
    if ((p = (TNOD *) malloc(n)) == 0) {
        printf("memorie insuficientă la crearea listei\n");
        exit(1);
    }

    /* se încarcă date în nod */
    if ((i = incnod(p)) != 1) {
        elibnod(p); /* se eliberează zona rezervată pentru nod
                       deoarece nu s-au încărcat date în nod */
        return i;
    }

    /* se înlanțează nodul în lista */
    if (ultim != 0)
        /* lista nu este vida */
        ultim->urm = p;
    else
        /* lista vida */
        prim = p;

    ultim = p;
    ultim->urm = 0;
} /* sfîrșit for */
} /* sfîrșit crelist */
```

Așa cum s-a arătat înainte, o listă poate fi organizată păstrind elementele ei în memoria *heap*, iar adresele lor intr-o tabelă de pointeri.

Dăm mai jos o variantă a funcției *crelist* care să creeze o listă după acest principiu. În acest caz, tabloul de pointeri va fi global și are un număr maxim de elemente.

Numim *ttnod* acest tablou. De asemenea, vom nota cu *itpnod* variabila globală care are ca valoare indicele primului element al tabloului *ttnod* care este liber. Cu alte cuvinte, *ttnod[itpnod - 1]* este pointerul spre ultimul nod adăugat la listă. Inițial *itpnod* are valoarea zero.

Tipul nodurilor nu mai este necesar să fie recursiv, deoarece în acest caz ordonarea se realizează prin indici:

- primul nod are indicele zero;
- al doilea nod are indicele 1;
- ...
- ultimul nod are indicele *itpnod*-1.

De aceea, în locul tipului TNOD vom folosi tipul:

```

typedef struct {
    declaratii
} TTNOD;

```

Tabloul *tmod* se declara astfel:

```

TTNOD *tmod[MAX];

```

unde:

MAX

- Este o constantă simbolică în prealabil definită printr-o construcție `#define` și valoarea ei se va alege așa încit să nu fie depășita de numărul elementelor listei.

Variabila *tmod* este de tip *int*:

int *tmod*;

Indicăm mai jos funcția pentru crearea listei cu utilizarea tabloului *tmod*. Ea utilizează funcțiile *incnod* și *elibnod* ca în cazul funcției *crelist*, fiind analogă cu aceasta.

```

int tpcrelist () /*- creaaza o lista folosind un tablou de pointeri
                    care contine adresele nodurilor listei;
                    - functia returneaza:
                      0 - la eroare;
                      -1 - la creare normala. */
{
    extern TTNOD *tmod[];
    extern int tmod;
    int i,n;
    TTNOD *p;

    n=sizeof(TTNOD);
    for(itmod=0;itmod<MAX;itmod++){
        if((p=(TTNOD *)malloc(n))==0){
            printf("memorie insuficienta\n");
            exit(1);
        }

        /* se incarca date in nod */
        if((i=incnod(p))!=1){
            elibnod(p);
            return i;
        }
        tmod[itmod]=p; /* pastreaza adresa nodului in tabloul de pointeri */
    } /* sfarsit for */
}

```

11.1.2. Accesul la un nod al unei liste simplu înlățuite

Puteam avea acces la nodurile unei liste simplu înlățuite începînd cu nodul spre care pointeaza variabila globală *prim* și trecînd apoi pe rînd de la un nod la altul, folosind pointerul *urm*.

Există cazuri în care dorim acces la un nod anumit al listei. În acest caz, este necesar să definim modul de identificare al nodului la care dorim să avem acces. Un mod simplu este acela de a indica numărul de ordine al nodului la care se dorește acces, de exemplu, se dorește acces la al *n*-lea nod al listei. Cum lista este o mulțime dinamică, acest mod de a defini accesul la un nod al ei nu este totdeauna cel mai nimerit. O altă metodă este aceea, de a avea o dată componentă a nodurilor, care să aibă valori diferite, pentru noduri diferite. În acest caz se poate defini accesul la nodul din listă pentru care data respectivă are o valoare dată. O astfel de dată, care este componentă a nodurilor unei liste și are valori distincte pentru noduri diferite ale unei liste se numește *cheie*.

Cheia poate fi o dată de un tip oricare.

În cazul de față vom considera cheia de tip *int*. Cititorul poate modifica simplu funcția de mai jos pentru a utiliza chei de alte tipuri: *long*, *char*, *double* etc.

Funcția returnează pointerul spre nodul căutat sau zero în cazul în care lista nu conține un nod a cărui cheie să aibă valoarea indicată de parametrul ei.

Nodurile listei au tipul definit astfel:

```

typedef struct tnod{
    declaratii
    int cheie;
    declaratii
    struct tnod *urm;
} TNOD;

```

Puteam acum defini funcția de căutare ca mai jos:

```

TNOD *cnei(int c)
/* - cauta un nod al listei pentru care cheie=c;
   - returneaza pointerul spre nodul determinat in acest fel sau zero daca nu exista nici
     un nod asa incit cheia sa aiba valoarea c. */
{
    extern TNOD *prim;
    TNOD *p;

    p=prim; /* cautarea incepe cu nodul spre care pointeaza variabila prim */
    while(p!=0){
        if(p->cheie==c)
            return p; /* s-a gasit un nod pentru care cheie=c */
        p=p->urm; /* se trece la nodul urmator din lista */
    }
    /* in acest punct se ajunge cind nu exista un nod in lista cu proprietatea ca cheie=c */
    return 0;
}

```

O funcție similară se poate construi și pentru cazul în care la implementarea listei se utilizează tabloul de pointeri *tmod* (vezi paragraful precedent).

În acest caz tipul nodurilor listei se declară astfel:

```

typedef struct {
    declaratii
    int cheie;
    declaratii
} TTNOD;

Funcția caută nodul din listă parcurgind-o cu ajutorul indicelui care variază de la zero pînă la itpnod-1 inclusiv.

TTNOD *tpcncl(int c)
/*- cauta un nod al listei pentru care cheie=c;
 - returnaza pointerul spre nodul respectiv sau zero daca nu există în
 lista un astfel de nod */
{
    extern TTNOD *tpnod[];
    extern int itpnod;
    int i;

    for(i=0;i<itpnod;i++)
        if(tpnod[i] == cheie==c) return tpnod[i];
    return 0;
}

```

Ambele funcții parcurg elementele listei, nod cu nod, fie pînă la întîlnirea nodului căutat, fie pînă la sfîrșitul listei dacă nu există în listă un nod a căruia cheie să aibă valoarea cerută. Această metodă de căutare se numește *căutare liniară*. Ea este o metodă foarte simplă de căutare dar nu este eficientă și de aceea se aplică numai la liste cu un număr relativ mic de noduri.

O metodă mult mai eficientă este metoda *căutării binare*. Această metodă poate fi aplicată simplu la listele implementate cu ajutorul tabloului de pointeri *tpnod*. În acest scop lista trebuie întîi sortată, de exemplu crescător, în raport cu valorile cheii. Sortarea se poate realiza procedind ca în exercițiul 8.18. în care cuvintele citite de la intrarea standard se păstrează în memoria *heap* și în paralel cu aceasta se definește un tablou cu adresele la care sunt păstrate aceste cuvinte. Apoi se realizează sortarea lor (ordonarea în ordine alfabetică) folosind metoda bulelor.

Pentru o eficiență mai bună se poate utiliza o altă metodă de sortare (sortare *shell*, sortare rapidă, sortare cu ajutorul arborilor etc.).

11.1.3. Inserarea unui nod într-o listă simplu înlățuită

Într-o listă simplu înlățuită se pot face inserări de noduri în diferite poziții. Amintim cîteva posibilități care intervin mai frecvent:

- a. inserare înaintea primului nod;
- b. inserare înaintea unui nod precizat printr-o cheie;
- c. inserare după un nod precizat printr-o cheie;
- d. inserare după ultimul nod al listei (adăugarea unui nod nou la listă).

În cele ce urmează se definesc funcții pentru cazurile a-d indicate mai sus, atât pentru liste simplu înlățuite, cât și variantele lor cînd lista se implementează cu ajutorul tabloului de pointeri *tpnod*.

Tipurile TNOD și TTNOD se consideră declarate ca în paragrafele precedente.

11.1.3.1. Inserarea unui nod într-o listă simplu înlățuită înaintea primului ei nod

Primul nod al unei liste simplu înlățuite este nodul spre care nu pointează pointerul *urm* al nici unui nod al listei, adică este nodul care nu este succesorul nici unui nod din listă. Amintim că spre acest nod pointează variabila *prim* dacă lista nu este vidă.

În cazul utilizării tabloului *tpnod*, primul nod al listei este nodul spre care pointează elementul:

tpnod[0]

Funcția de inserare returnază pointerul spre nodul inserat în listă sau zero în cazul în care nu se poate realiza inserarea. De altfel, toate funcțiile de inserare returneză aceste valori.

```

TNOD *iniprim() /* inserează nodul curent înaintea primului nod al listei */
{
    extern TNOD *prim,*ultim;
    TNOD *p;
    int n;
    n=sizeof(TNOD);

    /* rezerva memorie heap și încarcă datele în zona respectiva */
    if (((p=(TNOD *)malloc(n))!=0)&& (!inenod(p)==1)) {
        if (prim==0) { /* lista vida */

            /* lista se compune numai din nodul curent */
            prim=ultim=p;
            p->urm=0; /* nu există succesor */
        }
        else{
            p->urm=prim; /* primul nod al listei devine succesorul
            nodului care se inseră */
            prim=p; /* nodul inserat nu este succesorul nici unui alt
            nod și de aceea prim pointează spre el */
        }
        return p;
    }
    if (p==0){
        printf("memorie insuficientă\n");
        exit(1);
    }
    /* s-a rezervat memorie pentru nod dar nu s-au încărcat date în nod */
    elibnod(p);
}

```

```

        return 0;
    }
}

```

Inserarea în cazul utilizării tabloului *ttnod* implică eliberarea elementului *ttnod[0]*. În acest scop se fac atribuirile:

ttnod[i]=ttnod[i-1] pentru *i=itpnod, itpnod-1, ..., 2, 1*

Se observă că funcția de inserare înaintea primului nod pentru liste, care folosește tabloul de pointeri *ttnod* nu mai este așa de eficientă ca funcția *iniprim* definită mai sus.

```

TTNOD *tpiniprim() /* insereaza un nod inaintea primului nod al listei */
{
    extern TTNOD *ttnod[];
    extern int itpnod;
    int n;
    int i;

    if(itpnod >= MAX){
        printf("lista are prea multe elemente\n");
        return 0;
    }
    n = sizeof(TTNOD);
    if(((p=(TTNOD *)malloc(n))!=0)&&(incnod(p)==1)){
        /* deplaseaza valorile elementelor tabloului ttnod */
        for(i=itpnod++; i>0; i--) ttnod[i]=ttnod[i-1];
        ttnod[0]=p;
        return p;
    }
    if(p==0){
        printf("memorie insuficienta\n");
        exit(1);
    }
    elibnod(p);
    return 0;
}

```

11.1.3.2. Inserarea unui nod într-o listă simplu înlățuită înaintea unui nod precizat printr-o cheie

Vom presupune că cheia este de tip *int* ca și în cazul funcției *cnci* definită în paragraful 11.1.2.

```

TNOD *inici(int c)
/* - insereaza un nod înaintea unui nod precizat printr-o cheie numerică;
   - returneaza pointerul spre nodul inserat sau zero daca inserarea nu are loc. */
{
    extern TNOD *prim;
    TNOD *q, *q1, *p;
    int n;

    n = sizeof(TNOD);

```

```

/* - cauta nodul de cheie=c;
   - la terminarea ciclului:
       q - pointeaza spre nodul respectiv;
       q1 - pointeaza spre nodul precedent celui spre care pointeaza q. */
q1=0;
q=prim;
while(q){
    if(q->cheie==c) break; /* s-a gasit nodul cautat */
    q1=q;
    q=q->urm;
}
if(q==0){ /* nu exista in lista un nod de cheie=c */
    printf("nu exista in lista un nod de cheie=%d\n", c);
    return 0;
}

/* se rezerva zona pentru nod si se incarca datele nodului in zona respectiva */
if(((p=(TNOD *)malloc(n))!=0)&&(incnod(p)==1)){
    if(q==prim){

        /*cheie=c pentru primul nod si inserarea se face inaintea primului nod*/
        p->urm=prim;
        prim=p;
    }
    else{

        /*nodul spre care pointeaza p se insereaza intre nodul spre care
           pointeaza q1 si nodul spre care pointeaza q */
        q1->urm=p; /* succesorul nodului spre care pointeaza
                       q1 este nodul spre care pointeaza p */
        p->urm=q; /* succesorul nodului spre care pointeaza p
                      este nodul spre care pointeaza q */
    }
    return p;
}

/* nu s-a facut inserarea ceruta */
if(p==0){
    printf("memorie insuficienta\n");
    exit(1);
}
elibnod(p);
return 0;
}

```

În continuare definim funcția care utilizează tabloul de pointeri *ttnod*.

```

TTNOD *tpinici(int c)
/* - insereaza un nod înaintea unui nod precizat printr-o cheie numerică;
   - returneaza pointerul spre nodul inserat sau zero daca nu are loc inserarea. */
{
    extern TTNOD *ttnod;
    extern int itpnod;
    int i, k;

```

```

/*daca itpnod >= MAX nu exista loc liber in tabloul de pointeri */
if(itpnod >= MAX)
    printf("lista are prea multe elemente\n");
    return 0;
}

/* cauta nodul de cheie=c */
for(i=0;i<itpnod;i++)
    if(tpnod[i] == cheie==c)
        break; /* s-a gasit nodul cautat */
if(i==itpnod) { /* nu exista in lista un nod de cheie=c */
    printf("nu exista in lista un nod de cheie=%d\n",c);
    return 0;
}
n = sizeof(TTNOD);

/* rezerva zona pentru nod si se incarca datele nodului in zona respectiva */
if(((p=(TTNOD *)malloc(n))!=0)&&(incnod(p)==1)) {

    /* deplaseaza valorile elementelor tabloului tpnod:
       tpnod[k]=tpnod[k-1], pentru k=itpnod,itpnod-1,...,i+1. */
    for(k=itpnod++;k>i;k--) tpnod[k]=tpnod[k-1];

    /* tpnod[i] devine egal cu adresa nodului care se inseraza */
    tpnod[i]=p;
    return p;
}

/* nu s-a facut inserarea */
if(p==0){
    printf("memorie insuficienta\n");
    exit(1);
}
elibnod(p);
return 0;
}

```

11.1.3.3. Inserarea unui nod într-o listă simplu înlățuită după un nod precizat printr-o cheie

Vom presupune că, cheia este de tip *int*.

```

TNOD *indci(int c)
/* - inseraza un nod dupa un nod precizat printr-o cheie numerică;
   - returneaza pointerul spre nodul inserat sau zero daca inserarea nu are loc. */
{
    extern TNOD *prim,*ultim;
    TNOD *p,*q;
    int n;

    /* cauta nodul de cheie=c */
    for(q=prim;q;q=q->urm)
        if(q->cheie==c) break;
    if(q==0) {

```

```

        printf("nu exista in lista un nod de cheie=%d\n",c);
        return 0;
    }

    /* se rezerva zona pentru nod si se incarca datele in nod */
    n = sizeof(TNOD);
    if(((p=(TNOD *)malloc(n))!=0)&&(incnod(p)==1)) {

        /* se inseraza nodul spre care pointeaza p dupa nodul spre care pointeaza q */
        p->urm=q->urm; /* nodul succesor nodului spre care pointeaza q devine
                           succesor nodului spre care pointeaza p */
        q->urm=p; /* succesorul nodului spre care pointeaza q
                     devine nodul spre care pointeaza p */
        if(ultim==q) /* - nodul spre care pointeaza q nu a avut succesor;
                       - in acest moment nodul spre care pointeaza p nu are succesor. */
            ultim=p;
        return p;
    }

    /* nu s-a reusit inserarea nodului */
    if(p==0) {
        printf("memorie insuficienta\n");
        exit(1);
    }
    elibnod(p);
    return 0;
}

```

Functia de inserare pentru liste implementate cu ajutorul tabloului de pointeri *tpnod* care realizeaza inserarea dupa un nod de cheie data este asemănatoare cu functia *tpinici* definită în paragraful precedent. În acest caz, se modifică instrucțiunea *for* pentru deplasarea elementelor tabloului *tpnod* și instrucțiunea de atribuire următoare ei:

```

for(k=itpnod++;k>i+1;k--) tpnod[k]=tpnod[k-1];
tpnod[i+1]=p;

```

11.1.3.4. Adăugarea unui nod la o listă simplu înlățuită

Adăugarea unui nod la o lista simplu înlățuită înseamnă inserarea lui după nodul spre care pointează variabila *ultim*. Aceasta înseamnă că după inserarea nodului, variabila *ultim* pointează spre nodul respectiv, acesta devenind nodul din listă care nu are succesor.

```

TNOD *adauga()
/*- adauga un nod la o lista simplu inlantuita;
   - returneaza pointerul spre nodul inserat sau zero daca nu se realizeaza inserarea. */
{
    extern TNOD *prim,*ultim;
    TNOD *p;
    int n;

```

```

/* se rezerva zona de memorie pentru nod si se incarca datele in zona respectiva */
nsizeof (TNOD);
if ((p=(TNOD *)malloc (n))!=0) && (incnod (p)==1) {
    if (prim==0) /* lista este vida */
        prim=ultim=p;
    else{
        ultim->urm=p; /* succesorul nodului spre care pointeaza ultim
                           devine nodul spre care pointeaza p */
        ultim=p; /* acesta devine nodul spre care pointeaza ultim */
    }
    p->urm=0; /* nodul spre care pointeaza p nu are succesor */
    return p;
}

/* nu s-a reusit inserarea nodului in lista */
if (p==0){
    printf("memorie insuficienta\n");
    exit(1);
}
elibnod(p);
return 0;
}

```

Definim mai jos funcția de adăugare a unui nod la o listă implementată cu ajutorul tabloului *ttnod*.

```

TTNOD *tpadauga()
/* adauga un nod la o lista;
 - returneaza pointerul la nodul inserat sau zero daca inserarea nu se realizeaza. */
{
    extern TTNOD *ttnod[];
    extern int itpnod;
    int n;

    if (itpnod >= MAX){
        printf("lista are prea multe elemente\n");
        return 0;
    }
    nsizeof (TTNOD);
    if (((p=TTNOD *)malloc (n))!=0)&&(incnod(p)==1)){
        ttnod[itpnod++]=p;
        return p;
    }
    if (p==0){
        printf("memorie insuficienta\n");
        exit(1);
    }
    elibnod (p);
    return 0;
}

```

11.1.4. Ștergerea unui nod dintr-o listă simplu înlățuită

Dintr-o listă simplu înlățuită se pot șterge noduri. Aceasta se poate realiza în mai multe moduri. În cele ce urmează avem în vedere următoarele cazuri:

- ștergerea primului nod al listei simplu înlățuite;
- ștergerea unui nod precizat printr-o cheie;
- ștergerea ultimului nod al listei simplu înlățuite.

Funcțiile de ștergere utilizează funcția *elibnod*. Această funcție eliberează zona de memorie alocată nodului care se șterge, precum și eventualele zone de memorie alocate suplimentar prin intermediul funcției *incnod* pentru a păstra diferite componente ale unui nod (de exemplu componente de tip sir de caractere).

Alături de funcțiile obișnuite de ștergere, se definesc astfel de funcții și pentru listele implementate cu ajutorul tabloului *ttnod*.

11.1.4.1. Ștergerea primului nod al unei liste simplu înlățuite

```

void spn() /* sterge primul nod din lista */
{
    extern TNOD *prim,*ultim;
    TNOD *p;

    if (prim==0) /* lista vida */
        return;
    p=prim;
    prim=prim->urm;
    elibnod(p);
    if (prim==0) /* lista a devenit vida */
        ultim=0;
}

```

În cazul în care lista este implementată cu ajutorul tabloului *ttnod*, se elimină nodul spre care pointează elementul *ttnod[0]*, apoi se deplasează elementele tabloului *ttnod* folosind atribuirile:

```

ttnod[i]=ttnod[i+1], pentru i=0, 1, ..., itpnod-2.
void tpspn() /* sterge primul nod din lista */
{
    extern TTNOD *ttnod[];
    extern int itpnod;
    int i;

    elibnod(ttnod[0]);
    for(i=0;i<itpnod-1,i++) ttnod[i]=ttnod[i+1];
    itpnod--;
}

```

11.1.4.2. Sterge un nod precizat printr-o cheie, dintr-o listă simplu înlățuită

Vom considera că, cheia este de tip *int*. În acest caz tipul TNOD se definește ca în paragraful 11.1.2.

```
void snci(int c) /* sterge nodul pentru care cheie=c */
{
    extern TNOD *prim,*ultim;
    TNOD *q,*q1;

    q1=0;
    q=prim;

    /*- se cauta nodul de cheie=c;
     - la terminarea ciclului q pointeaza spre nodul respectiv sau
      q=0 daca nu exista un astfel de nod;
     - q1 pointeaza spre nodul al carui succesor este nodul spre care
      pointeaza q sau q1=0 daca q=prim. */
    while(q){
        if(q->cheie==c) break; /* s-a gasit nodul cautat */
        q1=q;
        q=q->urm;
    }
    if(q==0) { /* nu exista in lista un nod pentru care cheie=c */
        printf("lista nu contine un nod de cheie=%d\n",c);
        return;
    }

    /* se sterge nodul spre care pointeaza q */
    if(q==prim){ /* se sterge primul nod din lista */
        prim=prim->urm;
        elibnod(q);
        if(prim==0) ultim=0; /* lista a devenit vida */
    }
    else{
        q1->urm=q->urm; /*succesorul nodului spre care pointeaza
                           q1 devine succesorul nodului spre care pointeaza q */
        if(q==ultim) /* se sterge ultimul nod, deci nodul spre care
                       pointeaza q1 devine ultimul nod al listei */
            ultim=q1;
        elibnod(q);
    }
}
```

În cazul în care lista este implementată cu ajutorul tabloului *ttnod*, se caută nodul pentru care cheie=c. Fie acesta nodul a cărui adresă este dată de elementul *ttnod[i]*.

După eliminarea nodului respectiv, se realizează atribuirile:

ttnod[k]=ttnod[k+1], pentru k=i, i+1, ..., itpnod-2.

```
void tpsnci(int c) /* sterge nodul de cheie=c */
{
    extern TTNOD *ttnod[];
    extern int itpnod;
    int i;

    for(i=0;i<itpnod;i++)
        if(ttnod[i] == cheie=c) break;
    if(i==itpnod){
        printf("nu exista un nod de cheie=%d\n",c);
        return;
    }
    elibnod(ttnod[i]);
    for(;i<itpnod-1;i++) ttnod[i]=ttnod[i+1];
    itpnod--;
}
```

11.1.4.3. Stergerea ultimului nod dintr-o listă simplu înlățuită

```
void sun() /* sterge ultimul nod din lista */
{
    extern TNOD *prim,*ultim;
    TNOD *q,*q1;

    q1=0;
    q=prim;
    if(q==0) return; /* listavida */
    while(q!=ultim){ /* se parcurge lista pînă se ajunge la ultimul nod al ei */
        q1=q;
        q=q->urm;
    }
    if(q==prim) /* lista contine un singur nod care se sterge;
                  lista devine vida; */
        prim=ultim=0;
    else{
        /* - nodul spre care pointeaza q1 are ca succesor nodul spre care
           pointeaza q și acesta este ultimul nod al listei;
           - cum nodul spre care pointeaza q se sterge, nodul spre care pointeaza
           q1 devine ultimul, deci q1->urm=0 și ultim=q1. */
        q1->urm=0;
        ultim=q1;
    }
    elibnod(q);
}
```

În cazul în care lista se implementează folosind tabloul *ttnod*, operația de stergere a ultimului nod este mult mai eficientă, deoarece nu necesită parcurgerea nodurilor listei.

```
void tpsum() /* sterge ultimul nod al listei */
{
    extern TTNOD *ttnod[];
    extern int itpnod;
```

```

    if(itpnod==0) return; /* lista vida */
    elibnod(--itpnod);
}

```

Observație:

Funcțiile definite în paragrafele precedente realizează operațiile cele mai frecvent utilizate asupra listelor. Unele dintre aceste funcții necesită parcurgerea nodurilor listei, parțial sau în totalitate. Alte funcții nu necesită astfel de parcurgeri. Funcțiile care parcurg nodurile unei liste au o eficiență scăzută în comparație cu cele care nu fac astfel de parcurgeri. De aceea, se recomandă pe cât posibil utilizarea funcțiilor care nu necesită parcurgerea nodurilor ei. Astfel de funcții sunt:

- | | |
|----------------|---|
| <i>iniprim</i> | - Inserează nodul curent înaintea primului nod al listei (11.1.3.1.). |
| <i>adauga</i> | - Adaugă un nod la o listă (11.1.3.4.). |
| <i>spn</i> | - Ștergerea primului nod al listei (11.1.4.1.). |

În cazul listelor implementate cu ajutorul tabloului *ttnod*, cele mai eficiente funcții sunt cele care nu necesită instrucțiuni ciclice care să se execute asupra elementelor tabloului *ttnod*. Astfel de funcții sunt:

- | | |
|-----------------|---------------------------------|
| <i>tpadauga</i> | - Adaugă un nod la o listă. |
| <i>tpsun</i> | - Șterge ultimul nod al listei. |

De aceste observații se ține, uneori, seama la alegerea modului de implementare a listelor în aplicații.

11.1.5. Ștergerea unei liste simplu înlățuite

Se utilizează funcția *elibnod* pentru fiecare nod al listei.

```

void streglist() /* sterge o lista simplu inlantuita */
{
    extern TNOD *prim,*ultim;
    TNOD *p;

    while(prim){
        p=prim;
        prim=prim->urm;
        elibnod(p);
    }
    ultim=0;
}

```

Mai jos definim funcția de ștergere a listei implementate cu ajutorul tabloului *ttnod*.

```

void tpsterglis() /* sterge lista implementata cu tablou de pointeri */
{
    extern TTNOD *ttnod[];
    extern int itpnod;
}

```

```

int i;

for(i=0;i<itpnod;i++) elibnod(ttnod[i]);
itpnod=0;
}

```

Exerciții:

11.1 Se consideră tipul utilizator:

```

typedef struct tnod{
    char *cuvant;
    int frecventa;
    struct tnod *urm;
}TNOD

```

Se cere să se scrie funcția *incnod* care încarcă datele curente într-un nod de tip TNOD.

Prin cuvînt se înțelege un sir de litere mici sau mari, ca în exercițiul 10.48. În acest exercițiu se definește funcția *citcuv* care citește un cuvînt de la intrarea standard și-l păstrează în memoria *heap*. Funcția respectivă returnează adresa de început a zonei în care se păstrează cuvîntul citit sau zero în caz că se întâlnește sfîrșitul de fișier.

Funcția de față apeleză funcția *citcuv* și atrbuie adresa returnată de ea pointerului *cuvant* din nodul curent. De asemenea, se atrbuie valoarea 1, variabilei *frecventa*.

Funcția *incnod* returnează valoarea -1 dacă *citcuv* returnează valoarea zero și 1 altfel.

FUNCȚIA BXI1

```

int incnod(TNOD *p)
/* încarcă datele curente în nodul spre care pointează p */
{
    if((p->cuvant = citcuv()) == 0) return -1;
    p->frecventa = 1;
    return 1;
}

```

11.2 Să se scrie funcția *elibnod* care eliberează zonele din memoria *heap* ocupate de nodul de tip TNOD definit în exercițiul precedent.

FUNCȚIA BXI2

```

void elibnod(TNOD *p)
/* eliberează zonele din memoria heap ocupate de nodul spre care pointează p */
{
    free(p->cuvant);
    free(p);
}

```

11.3 Să se scrie funcția *adauga*, care permite adăugarea unui nod de tip TNOD la o listă simplu înlățuită, după ultimul nod al listei. Tipul TNOD este cel

definit în exercițiul 11.1.

Să observăm că funcția *adauga* definită în paragraful 11.1.3.4. este dependență numai de pointerul *urm* din tipul nodului. De aceea, funcția *adauga* de mai jos este identică cu cea din paragraful amintit mai sus.

Funcția de față apelează funcțiile *incnod* și *elibnod* definite în exercițiile precedente.

FUNCȚIA BXI3

```
TNOD *adauga()
/* - adauga un nod la o lista simplu înlanțuită;
 - returnaza pointerul spre nodul adăugat sau zero daca nu s-a
 realizat adaugarea.*/
{
    extern TNOD *prim,*ultim;
    TNOD *p;
    int n;

    n = sizeof(TNOD);
    if((p = (TNOD *)malloc(n)) != 0) && (incnod(p) == 1)) {
        if(prim == 0) prim = ultim = p;
        else {
            ultim->urm = p;
            ultim = p;
        }
        p->urm = 0;
        return p;
    }
    if( p == 0 ) {
        printf("memorie insuficientă\n");
        exit(1);
    }
    elibnod(p);
    return 0;
}
```

11.4 Fie o listă simplu înlanțuită ale cărei noduri au tipul TNOD definit în exercițiul 11.1.

Să se scrie o funcție care caută în lista respectivă, nodul pentru care pointerul *cuvant* are ca valoare adresa unui cuvînt dat.

Cu alte cuvinte, pointerul *cuvant* joacă rol de cheie și se cere să se găsească nodul a cărui cheie pointeză spre un cuvînt dat.

Această funcție este asemănătoare cu funcția *cnci* definită în paragraful 11.1.2.

FUNCȚIA BXI4

```
TNOD *cncs(char *c)
/* - cauta un nod al listei pentru care cuvîntul spre care pointeaza cuvant
 este identic cu cel spre care pointeaza c;
 - returneaza pointerul spre nodul determinat sau zero daca nu există un astfel de nod */
```

```
{
    extern TNOD *prim;
    TNOD *p;

    for(p = prim; p ; p = p->urm)
        if(strcmp(p->cuvant,c) == 0) return p;
    return 0;
}
```

11.5 Să se scrie o funcție care sterge ultimul nod al unei liste simplu înlanțuite ale cărei noduri au tipul TNOD definit în exercițiul 11.1.

Această funcție este definită în paragraful 11.1.4.3 și ea este dependență numai de pointerul *urm* din tipul nodurilor.

Ea apelează funcția *elibnod* definită în exercițiul 11.2.

FUNCȚIA BXI5

```
void sun() /* sterge ultimul nod din lista */
{
    extern TNOD *prim,*ultim;
    TNOD *ql,*q;

    ql = 0;
    if(prim == 0) return;
    for(q = prim; q && q != ultim; q=q->urm) ql = q;
    if(q == prim) prim = ultim = 0;
    else {
        ql->urm = 0;
        ultim = ql;
    }
    elibnod(q);
}
```

11.6 Să se scrie un program care citește cuvîntele dintr-un text și afișează numărul de apariții al fiecărui cuvînt din textul respectiv.

Cuvîntul se definește ca o succesiune de litere mici și/sau mari. Textul se termină prin sfîrșitul de fișier.

Această problemă a fost rezolvată în exercițiul 10.50. Problema s-a rezolvat utilizind un tablou de pointeri ale cărui elemente sunt adrese pentru date păstrate în memoria *heap*. Acest tablou este similar cu tabloul *tpmod* introdus în acest capitol și utilizat la implementarea listelor.

În exercițiul de față programul se realizează utilizând o lista simplu înlanțuită ale cărei noduri au tipul TNOD definit în exercițiul 11.1.

Deși programul din exercițiul 10.50 este mai eficient din punct de vedere al timpului de execuție decât cel de față, el are o limitare cu privire la numărul de cuvînte diferite din text.

În exercițiul 10.50 s-a limitat maximul cuvîntelor diferite la 100. Evident, această limită poate fi schimbată simplu, modificind valoarea lui MAX.

În programul de față nu mai este necesar să definim acest maxim. El se

execută astfel:

1. La întâlnirea unui cuvînt se construiește un nod pentru cuvîntul respectiv. Acesta conține pointerul spre cuvînt, care este pastrat în memoria *heap*; frecvența de apariție a cuvîntului se face egală cu 1.
2. Nodul construit la punctul 1 se adaugă la lista simplu înlănuțită care se construiește.
3. Se cauta în lista un nod care să corespunda cuvîntului curent. Dacă există un astfel de nod și acesta nu este ultimul nod al listei, atunci se mărește frecvența din nodul respectiv și apoi se șterge ultimul nod al listei (cel adăugat la punctul 2) deoarece cuvîntul există deja în listă. După pasul 3 se revine la pasul 1 și ciclul continuă pînă cînd nu mai sunt cuvînte de citit (s-a ajuns la sfîrșitul de fișier). În acest moment se listează cuvîntele și frecvența lor de apariție, corespunzătoare nodurilor listei.

PROGRAMUL BXI6

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <string.h>

typedef struct tnod {
    char *cuvant;
    int frecventa;
    struct tnod *urm;
} TNOD;

#include "bx48.cpp" /* citeuv */
#include "bx11.cpp" /* incnod */
#include "bx12.cpp" /* elbnod */
#include "bx13.cpp" /* adauga */
#include "bx14.cpp" /* cnec */
#include "bx15.cpp" /* sun */

TNOD *prim, *ultim;
main() /* citește un text și afisează frecvența cuvîntelor din text */
{
    TNOD *p, *q;

    prim = ultim = 0; /* la început lista este vida */
    while((p = adauga()) != 0)

        /* s-a adăugat la lista un nod corespunzător ultimului cuvînt citit */
        if((q=cnec(p->cuvant)) != ultim) {

            /* - cuvîntul există într-un nod care nu este ultimul nod al listei;
             - deci există deja în lista;
             - se mărește frecvența lui și se șterge ultimul nod al listei. */
            q->frecventa++;
            sun();
        }
}
```

```
    }

    /* listează cuvîntele și frecvența lor */
    for(p = prim; p ; p = p->urm)
        printf("cuvîntul: %-5ls are frecvența: %d\n",
               p->cuvant, p->frecventa);
}
```

- 11.7 Să se scrie un program care citește cuvîntele dintr-un text și scrie numărul de apariții al fiecărui cuvînt în ordinea alfabetică a cuvîntelor respective.

Acest program este asemănător cu cel precedent. Diferența constă în faptul că înainte de a lista cuvîntele și frecvența lor, se face oordonare a nodurilor listei în aşa fel încît cuvîntele corespunzătoare nodurilor să fie ordonate alfabetic. Aceasta se realizează cu ajutorul funcției *ordlist* definită mai jos.

Ordonarea nodurilor se face pe baza schimbării înlănuțirii acestora.

Se utilizează metoda de sortare a bulelor. Potrivit acestei metode, se compară cuvîntele corespunzătoare a două noduri vecine. Dacă ele nu sunt în ordine alfabetică, atunci se schimbă ordinea de înlănuțire în listă a nodurilor respective.

PROGRAMUL BXI7

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <string.h>

typedef struct tnod {
    char *cuvant;
    int frecventa;
    struct tnod *urm;
} TNOD;

#include "bx48.cpp" /* citeuv */
#include "bx11.cpp" /* incnod */
#include "bx12.cpp" /* elbnod */
#include "bx13.cpp" /* adauga */
#include "bx14.cpp" /* cnec */
#include "bx15.cpp" /* sun */

void ordlist()
/* ordonează nodurile listei de tip TNOD în ordinea alfabetica a cuvîntelor corespunzătoare */
{
    extern TNOD *prim, *ultim;
    TNOD *p, *p1, *q;
    int ind;

    if(prim == 0) return; /* lista vida */
    ind = 1;
    while ( ind ) { /* ! */

        /* citîmp ind nu este zero, trebuie să se parcurgă lista și să se inverseze înlănuîrile
           nodurilor vecine care îi corespund cuvînte care nu sunt în ordine alfabetica */

```

```

ind = 0;
p = prim; /* p pointeaza spre nodul curent */
p1 = 0; /* p1 pointeaza spre nodul precedent */
q = p -> urm; /* q pointeaza spre nodul urmator */
while( q != 0 ) /* exista nod urmator */
    if(strcmp(p->cuvant, q->cuvant) > 0) {

        /* - cuvintele corespunzatoare nodului curent si nodului urmator nu sunt
           in ordine alfabetica;
        - se inverseaza inlantuirile lor. */
        if(p == prim) /* nodul curent nu are precedent */
            prim = q; /* q devine primul nod al listei */
        else
            p1 -> urm = q; /* q devine urmatorul lui p1 */
        p -> urm = q -> urm; /* urmatorul lui p devine urmatorul lui q */
        q -> urm = p; /* urmatorul lui q devine p */
        p1 = q; /* precedentul lui p devine q */
        q = p -> urm; /* q devine urmatorul lui p */
        if( q == 0 ) ultim = p; /* p nu mai are urmator */
        ind = 1; /* s-a facut o permutare, deci procesul de
                 ordonare nu este terminat */
    } /* sfarsit if */
    else { /* - nu se schimba inlantuirile nodurilor deoarece
              cuvintele corespunzatoare sint in ordine alfabetica;
              - se avanseaza in lista la perechea urmatoare de noduri. */
        p1 = p;
        p = q;
        q = q -> urm;
    } /* sfarsit else */
} /* sfarsit while */
} /* sfarsit ordlist */

TNOD *prim,*ultim;

main() /* citeste un text si afiseaza frecventa cuvintelor din textul
       respectiv in ordine alfabetica */
{
    TNOD *p;
    prim = ultim = 0;

    /* creeaza lista */
    while( adauga() != 0 )
        if((p = cncts(ultim -> cuvant)) != ultim) {
            p ->frecventa++;
            sun();
        }

    /* sortare */
    ordlist();

    /* listare */
    for( p=prim; p; p = p->urm)

```

```

        printf("cuvintul: %s are frecventa: %d\n",
               p->cuvant, p->frecventa);
    }
}

```

11.2. Stive și cozi

În exercițiul 7.1 s-a implementat o stivă cu ajutorul unui tablou. În general, o stivă se implementează printr-o listă simplu înlanțuită.

O *stivă* este o listă simplu înlanțuită gestionată conform principiului LIFO (Last In First Out).

Conform acestui principiu, ultimul nod pus în stivă este primul care este scos din stivă.

Stiva, ca și lista are două capete, *baza* stivei și *vîrful* stivei.

Asupra unei stive se definesc cîteva operații, dintre care cele mai importante sunt:

1. punе un element pe stivă (*push*);
2. scoate un element din stivă (*pop*);
3. șterge (videază) stiva (*clear*).

Primele două operații se realizează în vîrful stivei. Astfel, dacă se scoate un element din stivă, atunci acesta este cel din vîrful stivei și în continuare, cel pus anterior lui pe stivă ajunge în vîrful stivei.

Dacă un element se punе pe stivă, atunci acesta se punе în vîrful stivei.

Pentru a implementa o stivă printr-o listă simplu înlanțuită va trebui să identificăm baza și vîrful stivei cu capetele listei simplu înlanțuite. Există două posibilități:

- a. nodul spre care pointează variabila *prim* este baza stivei, iar nodul spre care pointează variabila *ultim* este vîrful stivei;
- b. nodul spre care pointează variabila *prim* este vîrful stivei, iar nodul spre care pointează variabila *ultim* este baza stivei.

În cazul *a*, funcțiile *push* și *pop* se indentifică prin funcțiile *adauga* și respectiv *sun*, definite în paragrafele 11.1.3.4 și respectiv 11.1.4.3.

Dacă funcția *adauga* este eficientă, în schimb funcția *sun* nu este eficientă.

În cazul *b*, funcțiile *push* și *pop* se indentifică prin funcțiile *iniprim* și respectiv *spn*, definite în paragrafele 11.1.3.1. și respectiv 11.1.4.1.

În acest caz ambele funcții sunt eficiente. De aceea, se recomandă implementarea stivei printr-o listă simplu înlanțuită conform cazului *b* indicat mai sus.

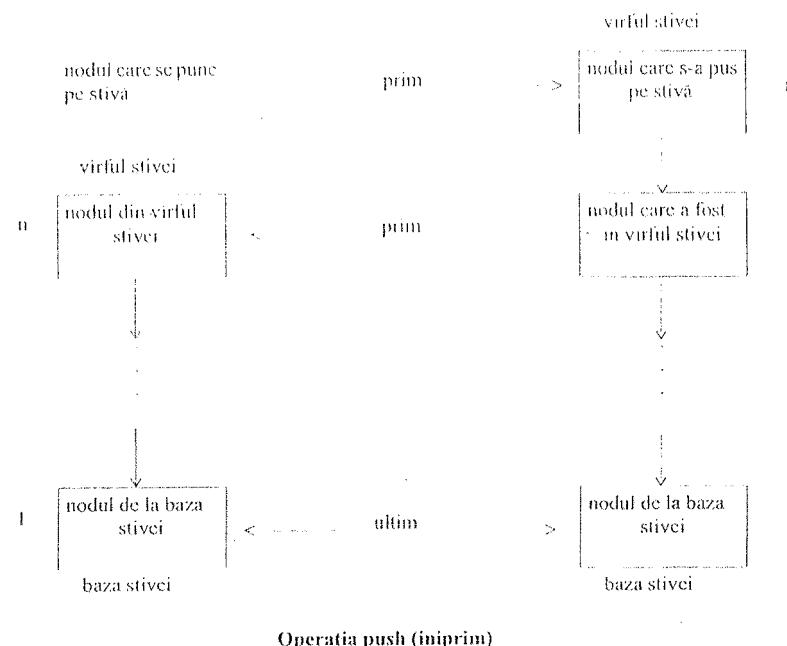
Cele două operații *push* și *pop* pot fi schematizate ca în figura de mai jos.

În sfârșit, să amintim că vidarea stivei se realizează cu ajutorul funcției *sterglis* definită în paragraful 11.1.5.

În concluzie, o stivă se poate implementa printr-o listă simplu înlanțuită,

pentru care se pot utiliza funcțiile:

- | | |
|-----------------|--------------------------------------|
| <i>iniprim</i> | - Realizează operația <i>push</i> . |
| <i>spn</i> | - Realizează operația <i>pop</i> . |
| <i>sterglis</i> | - Realizează operația <i>clear</i> . |



Stivele pot fi implementate ca în exercițiul 7.1., folosind liste care la rindul lor se implementează cu ajutorul tabloului de pointeri *tptnod* (vezi paragraful 11.1).

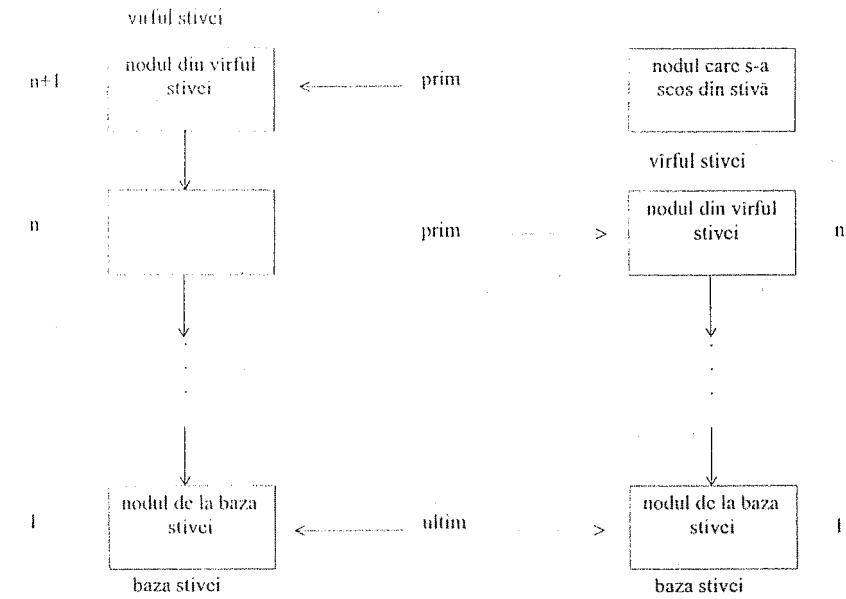
În acest caz, la baza stivei se află nodul spre care pointează elementul *tptnod[0]*, iar în virful stivei se află nodul spre care pointează elementul *tptnod[tptnod-1]*.

În acest caz se vor utiliza funcțiile definite în paragraful 11.1, astfel:

- | | |
|--------------------|--------------------------------------|
| <i>tptadauga</i> | - Realizează operația <i>push</i> . |
| <i>tpsun</i> | - Realizează operația <i>pop</i> . |
| <i>tpsterglist</i> | - Realizează operația <i>clear</i> . |

De obicei, în acest caz se mai definesc două operații asupra stivei:

isempty și *isfull*



Prima se definește printr-o funcție care returnează valoarea 1 dacă stiva este vidă și zero în caz contrar. Cea de a doua returnează valoarea 1 dacă stiva este plină și zero în caz contrar. Păstrind denumirile de mai sus, aceste funcții se definesc simplu astfel:

```
typedef enum {false,true} Boolean;
Boolean isempty() /* returneaza true daca stiva este vida si false altfel */
{
    extern int itpnod;
    return itpnod==0;
}

Boolean isfull() /* returneaza true daca stiva este plina si false altfel */
{
    extern int itpnod;
    return itpnod >= MAX;
}
```

Un alt principiu de gestiune a listelor simplu înlanțuite este principiul FIFO (First In-First Out).

Conform acestui principiu, primul element introdus în listă este și primul care este scos din listă.

Despre o listă gestionată în acest fel se spune că formează o *coadă*.

Cele două capete ale listei simplu înlanțuite care implementează o coadă sunt și capetele cozii. Asupra cozilor se definesc trei operații, ca și asupra stivelor:

- a. pune un element in coadă;
- b. scoate un element din coadă;
- c. ștergerea (vidarea) unei cozi.

Pentru a respecta principiul FIFO, vom pune un element în coadă folosind funcția *adauga* și vom scoate un element din coadă folosind funcția *spn*. Deci, la un capăt al cozii se pun elemente în coadă, iar din celălalt capăt se scoat elementele din coadă.

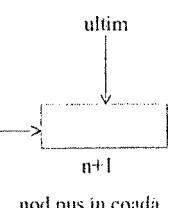
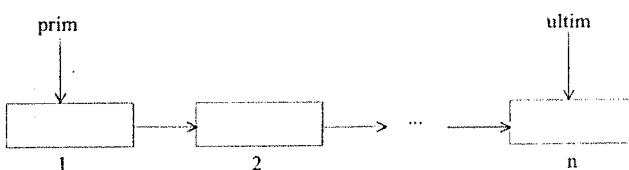
Ambele funcții, *adauga* și *spn* sunt funcții eficiente.

Ștergerea unei liste se realizează cu ajutorul funcției *sterglis*.

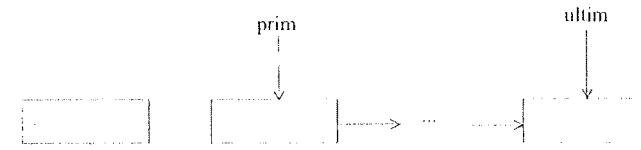
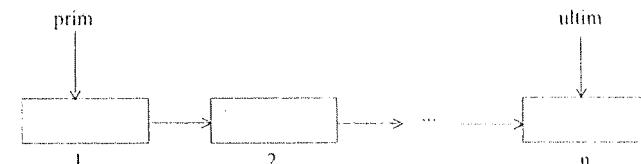
Primele două funcții sunt schematizate în figura următoare.

În concluzie, *coada* este o listă simplu înălțuită pentru care se pot utiliza funcțiile:

<i>adauga</i>	- Realizează operația de adăugare a unui nod la coadă.
<i>spn</i>	- Realizează operația de scoatere a unui nod din coadă.
<i>sterglis</i>	- Realizează ștergerea nodurilor existente în coadă.



Operația de punere în coadă



Operația de scoatere din coadă

Cozile definite ca mai sus corespund celor din viața de toate zilele și din această cauză ele se folosesc frecvent în probleme de simulare a fenomenelor reale.

Stivele se utilizează la descrierea proceselor recursive, inversarea ordinii elementelor unei mulțimi ordonate etc.

Exerciții:

11.8 Într-o gară se consideră un tren de marfă ale căruia vagoane sunt inventariate într-o listă, în ordinea vagoanelor. Lista conține, pentru fiecare vagon, următoarele date:

- codul vagonului (9 cifre);
- codul conținutului vagonului (9 cifre);
- adresa expeditorului (4 cifre);
- adresa destinatarului (4 cifre).

Deoarece în gară se inversează poziția vagoanelor, se cere listarea datelor despre vagoanele respective în nouă lor ordine.

În acest scop se crează o stivă în care se pastrează datele fiecarui vagon. Datele corespunzătoare unui vagon constituie un element al stivei, adică un nod al listei simplu înălțuite. După ce datele au fost puse pe stivă, ele se scoad de acolo și se listeză. În acest mod se obține lista vagoanelor în ordine inversă celei inițiale.

Stiva este o listă simplu înălțuită pe care o gestionăm folosind funcțiile:

iniprim și *spn*.

Aceste funcții sunt generale și ele sunt definite în paragrafele 11.1.3.1. și

respectiv 11.1.4.1.

Functia *inprim* apeleaza functia *incnod* si *elibnod*, iar functia *spn* numai functia *elibnod*. Aceste doua functii (*incnod* si *elibnod*) sunt specifice aplicatiei.

Functia principală apeleaza repetat functia *inprim* pentru a pune datele pe stiva, apoi le scoate de pe stiva și le listează, pînă cînd aceasta devine vida.

PROGRAMUL BX18

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

typedef struct tnod {
    long cvag;
    long cmarfa;
    int exp;
    int dest;
    struct tnod *urm;
} TNOD;

#include "bvi12.cpp" /* pcit_int */
#include "bvi13.cpp" /* pcit_int_lim */

int incnod(TNOD *p) /* incarca un nod cu datele despre vagoane */
{
    char t[255];
    char er[] = "s-a tastat EOF in pozitie rea\n";
    long cod;
    int icod;

    /* citeste cod vagon */
    for ( ; ; ) {
        printf("cod vagon: ");
        if(gets(t) == 0) return -1; /* nu mai sint date */
        if(sscanf(t,"%ld",&cod)==1 && cod >= 0 &&
           cod <= 999999999)
            break;
        printf("cod vagon eronat\n");
    }
    p->cvag = cod;

    /* citeste cod marfa */
    for ( ; ; ) {
        printf("cod marfa: ");
        if(gets(t) == 0) {
            printf(er);
            return 0;
        }
        if(sscanf(t,"%ld",&cod)==1 && cod >= 0 &&
           cod <= 999999999)
            break;
        printf("cod marfa eronat\n");
    }
}
```

```
p->cmarfa = cod;
/* citeste cod expeditor */
if(pcit_int_lim("cod expeditor: ",0,9999,&icod) == 0) {
    printf(er);
    return 0;
}
p->exp = icod;

/* citeste cod destinatar */
if(pcit_int_lim("cod destinatar: ",0,9999,&icod) == 0) {
    printf(er);
    return 0;
}
p->dest = icod;
return 1;
} /* sfîrșit incnod */

void elibnod(TNOD *p) /* elibereaza nodul spre care pointeaza p */
{
    free(p);
} /* sfîrșit elibnod */

TNOD *inprim() /* insereaza nodul curent inaintea primului nod al listei - push */
{
    extern TNOD *prim,*ultim;
    TNOD *p;
    int n;

    n = sizeof(TNOD);
    if((p = (TNOD *)malloc(n)) != 0) && (incnod(p) == 1)) {
        if(prim == 0) {
            prim = ultim = p;
            p->urm = 0;
        }
        else {
            p->urm = prim;
            prim = p;
        }
        return p;
    }
    if(p == 0) {
        printf("memorie insuficienta\n");
        exit(1);
    }
    elibnod(p);
    return 0;
} /* sfîrșit inprim */

void spn() /* sterge primul nod din lista - pop */
{
    extern TNOD *prim, *ultim;
    TNOD *p;

    if(prim == 0) return;
```

```

p = prim;
prim = prim -> urm;
elibnod(p);
if(prim == 0)
    ultim = 0;
} /* sfirsit spn */

TNOD *prim, *ultim;

main() /* listaza inventarul vagoanelor in ordinea inversa citirii lor */
{
    prim = ultim = 0; /* la inceput stiva este vida */

    /* se creeaza stiva apelind iniprim pina cind acesta returneaza valoarea zero */
    while(iniprim() != 0)
        ;

    /* - se listeaza elementul din virful stivei si apoi se sterge din stiva;
       - se repeta pina cind stiva devine vida. */
    while(prim != 0) {
        printf("cod vagon: %ld\tcontinut: %ld\n",
               prim -> cvag, prim -> cmarfa);
        printf("expeditor: %d\tdestinatar: %d\n",
               prim -> exp, prim -> dest);
        spn();
    }
} /* sfirsit main */

```

11.9 Cozile se folosesc adesea la simularea fenomenelor reale pe calculator. Un exemplu simplu de simulare cu ajutorul cozilor este descris mai jos.

Presupunem o agenție C.E.C. mică cu un singur ghișeu, care se deschide la ora 8. Agenția se închide la ora 16, dar publicul aflat la coadă este deservit în continuare. Cu alte cuvinte, după ora 16 publicul nu mai are voie să intre în agenție ca să se așeze la coada de la ghișeu. Întrucât s-a constatat că de obicei coada este destul de mare la ora închiderii, se ridică problema oportunității deschiderii a încă unui ghișeu la agenția respectivă. În acest scop se realizează o simulare pe calculator a situației existente care să stabilească o medie a orelor suplimentare efectuate zilnic pe o perioadă de un an.

Pentru a simula acest fenomen se au în vedere operațiile efectuate la ghișeu, după cum urmează:

Operatie	Timp de execuție
1 - depunere	5 minute
2 - restituire fără confirmare	7 minute
3 - depunere pe un carnet nou	10 minute
4 - restituire cu confirmare	20 minute
5 - lichidare	25 minute

Operațiile care se efectuează nu sunt uniform repartizate. În medie, operațiile 1 și 2 se execută cel mai frecvent iar operația 5 cel mai rar. În medie, s-a constatat

că operațiile 3 și 4 se solicită triplu față de operația 5, iar operațiile 1 și 2 sunt de 7 ori mai solicitate decât operația 5. Având în vedere acest fapt, considerăm o metodă de simulare a operațiilor care se execută la ghișeu bazată pe numere pseudoaleatoare. Dacă se ia în seamă operația 5, ea unitate, înseamnă că pentru operațiile 1 și 2 vom alege ponderea 7, iar pentru operațiile 3 și 4 ponderea 3:

Operatie	Pondere
operația 1	7
operația 2	7
operația 3	3
operația 4	3
operația 5	1
Total	21

Se vor genera numere pseudoaleatoare situate în intervalul [1,21], iar operația se definește astfel:

- Fie r un număr pseudoaleator din intervalul [1,21]:
- dacă $1 \leq r \leq 7$, atunci se realizează operația 1;
 - dacă $8 \leq r \leq 14$, atunci se realizează operația 2;
 - dacă $15 \leq r \leq 17$, atunci se realizează operația 3;
 - dacă $18 \leq r \leq 20$, atunci se realizează operația 4;
 - dacă $r = 21$, atunci se realizează operația 5.

Numerele pseudoaleatoare pot fi generate folosind funcțiile *random* și *srand* din biblioteca limbajelor C și C++.

Funcția *random* are prototipul:

int random(int n);

Ea returnează un număr pseudoaleator aflat în intervalul [0,n] (număr natural mai mic decât n).

Numerele pseudoaleatoare se generează printr-un proces de calcul iterativ. Acest proces pornește cu o valoare inițială care poate fi definită de programator apelând funcția *srand*. Această valoare inițială se numește *sămîntă* sirului de numere pseudoaleatoare.

Funcția *srand* are prototipul:

void srand(unsigned n);

unde:

n - Este valoarea la care se setează sămîntă după apelul funcției *srand*.

Ambele funcții au prototipul în fișierul *stdlib.h*.

Programul de simulare a problemei indicate mai sus construiește o coadă de așteptare cu persoanele care sosesc la agenție în intervalul de timp indicat mai sus.

Apelind funcția *random* se determină operația solicitată de fiecare persoană aflată la coada. Se scoad elementele din coada respectând principiul FIFO și procesul continua pînă la ora 16 cînd se sisteză operația de adăugare. Se determină timpul necesar pentru realizarea operațiilor solicitate de persoanele aflate la coada.

Rămîne de precizat modul în care vin persoanele la agenție. Vom presupune că intervalul de timp între două persoane care vin la agenție este aleator și că acesta este de maximum 15 minute. În acest caz se poate apela funcția *random* cu parametrul 15 și valoarea:

random(15)+1

va reprezenta intervalul la care sosesc persoanele următoare la agenție.

Programul folosește o variabilă globală *timp crt** a cărei valoare este numărul minutelor scurse de la ora 8 și pînă în momentul în care a sosit ultima persoană la agenție.

Aveam 8 ore de lucru la ghișeu, deci *timp crt* $\leq 8 \cdot 60 = 480$ minute.

O alta variabilă care numără minutele rezultate din deservirea persoanelor care s-au aflat la coada la ghișeu este *timpghiseu*.

Valorile acestor variabile satisfac relația:

timpghiseu \leq *timp crt*

Cînd o persoană se aşează la coada, atunci *timp crt* se mărește cu timpul dat de expresia:

random(15)+1.

Cînd o persoană se scoate din coadă (a fost deservită), *timpghiseu* se mărește cu timpul necesar pentru operația solicitată de persoana respectivă, operație care se definește cu ajutorul expresiei:

random(21)+1.

Deoarece *timp crt* definește timpul curent (numărul de minute care s-au socotit începînd cu ora 8 și pînă în momentul în care ultima persoană s-a așezat la coadă), rezultă că persoana din față este deservită (scoasă din coadă) numai dacă:

timpghiseu + timp necesar pentru operația ci \leq *timp crt*

Altfel se adaugă o nouă persoană la coada dacă *timp crt* ≤ 480 .

De asemenea, persoana din față cozii este deservită, dacă *timp crt* > 480 .

PROGRAMUL BXI9

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <conio.h>
typedef struct tnod {
    int timpoperatie;
    struct tnod *urm;
}
```

```
} TNOD;
#define MAXTIMP 480
#define INTERVAL 15

int incnod(TNOD *p)
/* - încarcă nodul curent dacă timpul curent nu depășește pe cel admis și returnează 1;
   - altfel returnează -1. */
{
    extern int timp crt;
    int r;

    /* determină timpul, în minute, la care sosesc persoane la agenție */
    timp crt += random(INTERVAL) + 1;
    if(timp crt > MAXTIMP) return -1; /* agenție inchisă */

    /* determină operația și pastrează în nod timpul necesar operațiilor respective */
    r = random(21) + 1;
    if(r <= 7) r = 5;
    else
        if(r <= 14) r = 7;
        else
            if(r <= 17) r = 10;
            else
                if(r <= 20) r = 20;
                else r = 25;
    p->timpoperatie = r;
    return 1;
} /* sfîrșit incnod */

void elibnod( TNOD *p)
/* eliberează zona ocupată de nodul listei spre care pointează p */
{
    free(p);
} /* sfîrșit elibnod */

#include "bxi3.cpp" /* adaugă */

void spn() /* sterge primul nod al listei(cozii) */
{
    extern TNOD *prim,*ultim;
    TNOD *p;

    if( prim == 0 ) return;
    p = prim;
    prim = prim->urm;
    elibnod(p);
    if(prim == 0) ultim = 0;
} /* sfîrșit spn */

TNOD *prim,*ultim;
int timp crt;
main()
/* simulează cozile de la o agenție CEC pe o perioadă de 1 an */
```

```

{
    int timpghiseu;
    int i;

    for(i=1; i <= 360; i++){
        srand(i*10); /* seteaza saminta sirului pseudoaleator */

        /* initializari la deschiderea agentiei */
        timpcert = timpghiseu= 0;
        prim = ultim = 0; /* coada este vida */
        while( adauga() ) /* ultima persoana sosită la agenzie se pune la coada */

            /* - se deservesc persoanele aflate la coada;
             - trebuie ca:
                1 - prim != 0 (altfel nu există coada);
                2 - timpghiseu <= timpcert */
            while ( prim != 0 && (timpghiseu <= timpcert) ) {
                /* se deserveste prima persoana din coada */
                timpghiseu += prim -> timpoperatie;
                /* se elimină din coada persoana deservită */
                spn();
            }

            /* - se închide agenzie;
             - se deservesc în continuare persoanele aflate la coada la ghiseu. */
            while( prim != 0 ) {
                timpghiseu += prim -> timpoperatie;
                spn();
            }

            /* se afisează timpul suplimentar în minute */
            if(timpghiseu - 480 > 0){
                printf("timp peste ora 16: %d minute\n",
                      timpghiseu - 480);

                if(i%22==0){
                    printf("Actionati o tasta pentru a continua\n");
                    getch();
                }
            }
        }
    } /* sfârșit main */
}

```

11.3. Listă circulară simplu înălțuită

În paragraful 11.1 s-a definit lista simplu înălțuită ca o mulțime ordonată de noduri, fiecare nod conținând un pointer spre un alt nod al listei, numit următorul nodului respectiv, exceptând un singur nod, care nu mai are următor. Acest nod, care nu mai are următor constituie un capăt al listei simplu înălțuite. Tot în paragraful amintim mai sus, am convenit că spre acest nod să poarte variabila *ultim*. De asemenea, lista simplu înălțuită conține un nod care nu este următorul

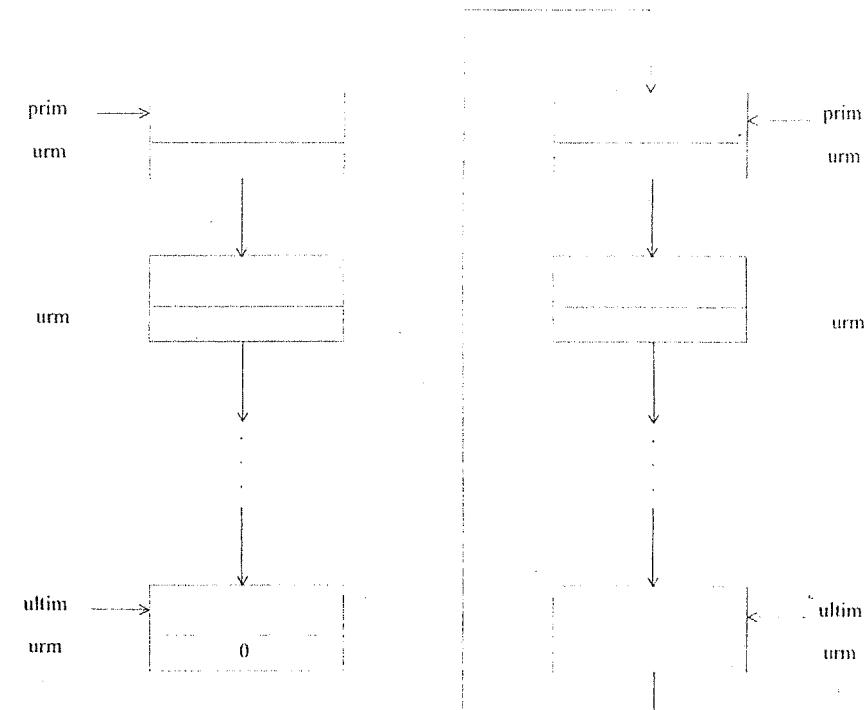
nici unui alt nod al ei. Acest nod constituie celalalt capăt al listei și spre el poartă variabila *prim*.

Pointerul prezent în fiecare nod al listei care definește ordinea nodurilor a fost numit *urm*. Conform celor spuse mai sus, *ultim -> urm=0*. Dacă într-o listă simplu înălțuită schimbăm valoarea expresiei *ultim -> urm* facind:

ultim -> urm=prim

atunci lista simplu înălțuită devine o *listă simplu înălțuită circulară*. În continuare prin *listă circulară* vom înțelege o listă circulară simplu înălțuită.

Procesul de transformare a listei simplu înălțuite în listă circulară este schematizat în figura de mai jos.



Listă simplu înălțuită

Listă circulară

Transformarea unei liste simplu înălțuite în listă circulară

Într-o listă circulară toate nodurile sunt echivalente: fiecare nod are un următor și fiecare nod este următorul unui nod. Într-o astfel de listă nu mai sunt capete și de aceea nu mai sunt necesare variabilele *prim* și *ultim*. Gestiona-

nodurilor listei circulare se realizează folosind o variabilă globală care pointează spre un nod oarecare al listei. Numim *ptrnod* această variabilă. Ea se declară astfel:

```
TNOD *ptrnod;
```

unde:

TNOD - Este tipul comun nodurilor listei.

Asupra listelor circulare se definesc aceleasi operații ca și asupra listelor simplu înlanțuite.

Listele circulare au o serie de aplicații dintre care amintim:

- operații cu numere întregi care au un număr mare de cifre;
- operații asupra polinoamelor de una sau mai multe variabile;
- alocarea dinamică a memoriei.

Menționăm că funcțiile *malloc*, *free*, precum și celelalte utilizate la gestionarea dinamică a memoriei utilizează o zonă de memorie organizată ca o listă circulară.

Memoria liberă este o listă circulară de noduri, fiecare cu o zonă de memorie liberă. Un nod conține dimensiunea lui, un pointer spre nodul următor din listă și spațiul liber care se pune la dispoziția utilizatorului. Alocarea se face astfel:

La un apel a lui *malloc* se parcurge nodurile listei pînă cînd se găsește o zonă de memorie de dimensiune cel puțin egală cu cea cerută la apel. Dacă zona este mai mare, atunci ea se divide și partea neutilizată se înlanțuează cu celelalte blocuri ale listei. Cînd nu se găsește o zonă de memorie corespunzătoare, atunci se încearcă obținerea ei printr-un apel la sistemul de operare.

Eliberarea unei zone de memorie prin intermediul funcției *free* va înlanțui zona respectivă la lista circulară. Dacă două noduri cu zone de memorie liberă ocupă zone contigute, atunci ele se concatenază formind o singură zonă liberă de dimensiune egală cu suma dimensiunilor lor. De aceea, se recomandă utilizarea funcției *free* pentru a elibera zone de memorie de indată ce nu mai este nevoie de ele. În felul acesta se poate preîmpinge divizarea excesivă a zonei gestionată dinamic prin funcțiile de felul lui *malloc* și *free*.

Mai jos definim funcții pentru a realiza operațiile de bază asupra listelor circulare.

11.3.1. Crearea unei liste circulare

La crearea unei liste circulare, ca și la crearea unei liste simplu înlanțuite, se utilizează funcțiile *incnod* și *elibnod*.

Prima se apelează pentru a încărca datele curente într-un nod al listei, iar cea de a doua pentru a elibera zonele de memorie alocate pentru un nod.

Amintim că funcția *incnod* returnează:

- | | |
|----------|--------------------------|
| <i>0</i> | - La eroare. |
| <i>1</i> | - La încărcarea normală. |

- 1 - Nu mai sunt date de încărcat în nod.

Funcția de creare a listei circulare returnează:

- | | |
|-----------|-----------------------------------|
| <i>0</i> | - La eroare. |
| <i>-1</i> | - La crearea fără erori a listei. |

```
int ccrelist() /* - creaza o lista circulara;
                  - returneaza:
                    0 - la eroare;
                    -1 - la creare normala. */
{
    extern TNOD *ptrnod;
    int i, n;
    TNOD *p;

    n = sizeof(TNOD);
    ptrnod = 0; /* lista este vida la inceput */
    while(((p=(TNOD *)malloc(n))!=0) && (i=incnod(p))==1))

        /* s-a rezervat zona pentru nod si s-au incarcat date in zona respectiva */
        if(ptrnod==0) /* listavida */
            ptrnod=p;
            ptrnod->urm=p;
        }
        else{ /* nodul curent se insereaza dupa cel spre care pointeaza ptrnod */
            p->urm=ptrnod->urm;
            ptrnod->urm=p;
            ptrnod=p; /* ptrnod pointeaza spre ultimul nod inserat in lista */
        } /* sfîrșit else */

        /* sfîrșit while: p=0 sau incnod nu a returnat valoarea 1 */
        if(p==0){ /* nu s-a rezervat zona pentru nod */
            printf("memorie insuficienta\n");
            exit(1);
        }
        /* - s-a rezervat zona pentru nod dar incnod nu a reusit sa incarce date in
           zona respectiva returnind o valoare diferita de unu, deci zero sau -1;
           - valoarea returnata de incnod va fi returnata si de functia ccrelist. */

        elibnod(p); /* elibereaza zona de memorie rezervata pentru nod
                      si care n-a mai fost incarcata cu date */
        return i; /* i are ca valoare valoarea returnata de incnod */
}
```

Observație:

Funcția *ccrelist* a fost scrisă mai compact decit funcția *crelist* utilizând expresia:

$((p=(TNOD *)malloc(n))!=0)&&((i=incnod(p))==1)$

Care se evaluatează astfel:

Se apelează funcția *malloc* pentru a rezerva *n* octeți în memoria *heap*.

În cazul în care se poate rezerva o zonă de n octeți, lui p î se atribuie o valoare diferită de zero și deci primul operand al operatorului $\&\&$ are valoarea adevărat. În acest caz se evaluează operandul al doilea al operatorului $\&\&$, care apelează funcția *incnod* pentru a încărca datele curente în zona a cărei adresă de început este valoarea lui p .

În cazul în care nu se pot rezerva n octeți în memoria *heap*, lui p î se atribuie valoarea zero și deci primul operand al operatorului $\&\&$ este fals.

În acest caz întreaga expresie este falsă și deci nu se mai evaluează cel de al doilea operand al operatorului $\&\&$.

De asemenea, expresia este falsă și în cazul în care lui p î s-a atribuit o valoare diferită de zero, dar *incnod* a returnat o valoare diferită de unu.

11.3.2. Accesul la un nod al unei liste circulare

Ca în cazul listelor simplu înlanțuite și în cazul listelor circulare putem căuta un nod după o cheie sau mai multe chei. În cazul listelor circulare, căutarea va începe cu nodul spre care pointează variabila globală *ptrnod*.

Mai jos, se definește funcția *ccnci* care este analogă funcției *cnci* definită în cazul listelor simplu înlanțuite. Ea căută un nod după o cheie numerică și returnează una din valorile:

- pointerul spre nodul căutat;
- 0 dacă nu există un astfel de nod.

Amintim că în acest caz tipul nodurilor listei se declară astfel:

```
typedef struct tnod {
    declaratii
    int cheie;
    declaratii
} TNOD;

TNOD *ccnci(int c)
/* - cauta nodul de cheie=c;
   - returneaza pointerul spre nodul respectiv sau 0 daca nu exista un astfel de nod. */
{
    extern TNOD *ptrnod;
    TNOD *p;

    p=ptrnod;
    if(p==0) /* lista vida */
        return 0;
    do{
        if(p->cheie==c) return p;
        p=p->urm;
    }while(p!=ptrnod);
    return 0;
}
```

11.3.3. Inserarea unui nod într-o listă circulară

Considerăm două operații de inserare a unui nod într-o listă simplu înlanțuită:

- inserarea unui nod înaintea unuia precizat printr-o cheie numerică de tip *int*;
- inserarea unui nod după unul precizat printr-o cheie numerică de tip *int*.

Cititorul poate defini și alte funcții de inserare similare cu cele prezentate mai jos.

11.3.3.1. Inserarea unui nod înaintea unuia precizat printr-o cheie de tip *int*

Nodurile listei au tipul *TNOD* indicat în paragraful 11.3.2.

```
TNOD *ciniici(int c)
/* - inserarea unui nod într-o lista circulară înaintea unuia nod precizat printr-o cheie de tip int;
   - returneaza pointerul spre nodul inserat sau zero daca inserarea nu are loc. */
{
    extern TNOD *ptrnod;
    TNOD *p,*q,*ql;
    int n;

    if(ptrnod==0) return 0; /* lista vida */
    q=ptrnod;
    do{
        ql=q;
        q=q->urm;
        if(q->cheie==c)
            break; /* s-a gasit nodul înaintea caruia se va face inserarea */
    }while(q!=ptrnod);
    if(q->cheie!=c){
        printf("nu exista un nod de cheie=%d\n",c);
        return 0; /* nu s-a facut nici o inserare */
    }

    /* rezerva zona pentru nod și încarcă datele în nodul respectiv */
    n=sizeof(TNOD);
    if((p=(TNOD *)malloc(n))!=0)&&(incnod(p)==1)){
        ql->urm=p;
        p->urm=q;
        return p;
    }
    if(p==0){
        printf("memorie insuficienta\n");
        exit(1);
    }
    elibnод(p);
    return 0;
}
```

11.3.3.2. Inserarea unui nod după un nod precizat printr-o cheie de tip int

Tipul TNOD se definește ca în paragraful 11.3.2.

```
TNOD *cindci(int c)
/* inseraza un nod intr-o lista circulara dupa un nod precizat printr-o cheie de tip int */
{
    extern TNOD *ptrnod;
    TNOD *p,*q;
    int n;

    if(ptrnod==0) return 0; /* lista vida */
    q=ptrnod;
    do{
        if(q->cheie==c) break;
        q=q->urm;
    }while(q!=ptrnod);
    if(q->cheie!=c){
        printf("nu exista un nod de cheie =%d\n",c);
        return 0;
    }

    /* se face inserarea dupa nodul spre care pointeaza q */
    n=sizeof(TNOD);
    if(((p=(TNOD *)malloc(n))!=0)&&(incnod(p)==1)){
        p->urm=q->urm;
        q->urm=p;
        return p;
    }
    if(p==0){
        printf("memorie insuficienta\n");
        exit(1);
    }
    elibnod(p);
    return 0;
}
```

11.3.4. Ștergerea unui nod dintr-o listă circulară

Dăm mai jos o funcție care permite ștergerea dintr-o listă circulară a unui nod precizat printr-o cheie de tip *int*.

În cazul în care variabila *ptrnod* pointează chiar spre nodul care se șterge, convenim ca *ptrnod* să pointeze spre nodul precedent celui șters, dacă lista n-a devenit vidă. În acest ultim caz, lui *ptrnod* îi se atribuie valoarea zero.

Tipul TNOD se definește ca în cazul paragrafului precedent.

După modelul funcției de mai jos se pot defini și alte funcții pentru a șterge noduri dintr-o listă circulară.

```
void csnci(int c) /* sterge nodul pentru care cheie=c */
{
    extern TNOD *ptrnod;
```

```
TNOD *p,*p1;

if(ptrnod==0) return; /* listavida */
p=ptrnod;
do{
    p1=p;
    p=p->urm;
    if(p->cheie==c) break;
}while(p!=ptrnod);
if(p->cheie!=c){
    printf("lista nu contine un nod de cheie=%d\n",c);
    return;
}
if(p==p->urm){ /* lista are un singur nod */
    ptrnod=0;
}
else{
    p1->urm=p->urm;
    if(p==ptrnod) /* se sterge nodul spre care pointeaza ptrnod */
        ptrnod=p1;
}
elibnod(p);
}
```

11.3.5. Ștergerea unei liste circulare

```
void csterglist() /* sterge o lista circulara */
{
    extern TNOD *ptrnod;
    TNOD *p,*p1;

    if((p=ptrnod)==0) return; /* lista vida */
    do{
        p1=p;
        p=p->urm;
        elibnod(p1);
    }while(p!=ptrnod);
    ptrnod=0;
}
```

Exerciții:

11.10 Se consideră tipul TNOD declarat ca mai jos:

```
typedef struct tnod {
    char *cuv;
    struct tnod *urm;
} TNOD;
```

Acest tip se utilizează în toate exercițiile de la acest paragraf.

Mai jos, definim funcția care încarcă datele într-un nod de tipul TNOD.

FUNCȚIA BXI10

```
int incnod(TNOD *p)
/*- incarca datele in nodul spre care pointeaza p;
 - returneaza:
    -1 la intilnirea sfîrșitului de fisier;
    1 altfel.*/
{
    char t[255];

    p -> cuv = 0; /* initializarea cu pointerul nul */
    printf("tastati pe un rind cuvantul curent\n");
    if(gets(t) == 0) return -1; /* s-a tastat EOF */
    /* rezerva zona pentru rindul citit */
    if((p -> cuv = (char *)malloc(strlen(t) + 1)) == 0) {
        printf("memorie insuficienta\n");
        exit(1);
    }

    /* pastreaza rindul citit in memoria heap */
    strcpy(p -> cuv, t);
    return 1;
}
```

11.11 Să se definească funcția *elibnod* care eliberează zonele de memorie ocupate de un nod de tip TNOD.

FUNCȚIA BXI11

```
void elibnod(TNOD *p)
/* elibereaza zonele de memorie ocupate de nodul spre care pointeaza p */
{
    free(p -> cuv);
    free(p);
}
```

11.12 Să se scrie funcția *ccrelist* care crează o listă circulară ale cărei noduri sunt de tipul TNOD definit în exercițiul 11.10.

FUNCȚIA BXI12

```
int ccrelist()
/* - creaza o lista circulara;
 - returneaza:
    0 la eroare;
    -1 altfel.*/
{
    extern TNOD *ptrnod;
    int i,n;
    TNOD *p;

    n = sizeof(TNOD);
    ptrnod = 0;
    while(((p = (TNOD *)malloc(n)) != 0) &&
          ((i = incnod(p)) == 1))
}
```

```
if(ptrnod == 0) {
    ptrnod = p;
    ptrnod -> urm = p;
}
else {
    p -> urm = ptrnod -> urm;
    ptrnod -> urm = p;
    ptrnod = p;
}
if( p == 0 ) {
    printf("memorie insuficienta\n");
    exit(1);
}
elibnod(p);
return i;
}
```

11.13 Să se scrie o funcție care caută un nod al listei circulare create prin funcția *ccrelist*, nod pentru care pointerul *cuv* pointează spre un sir de caractere dat.

Funcția returnează pointerul spre nodul respectiv sau zero dacă nu există un astfel de nod.

FUNCȚIA BXI13

```
TNOD *ccrelist( char *c)
/* - cauta nodul pentru care cuv si c pointeaza spre acelasi sir de caractere;
 - returneaza:
    pointerul spre nodul respectiv sau zero daca nu exista un astfel de nod.*/
{
    extern TNOD *ptrnod;
    TNOD *p;

    p = ptrnod;
    if(ptrnod == 0) return 0; /* lista vida */
    do {
        if(strcmp( p -> cuv, c) == 0) return p;
        p = p -> urm;
    } while ( p != ptrnod );
    return 0;
}
```

11.14 Fie lista circulară creată cu ajutorul funcției *ccrelist* definită în exercițiul 11.12., fie *pnod* pointerul spre un nod al listei circulare pentru care:

pnod -> cuv

pointează spre un sir dat și $n > 1$ un întreg de tip *int*. Se cere nodul din lista obținut în urma eliminării nodurilor din listă în felul următor:

1. Se pornește cu nodul imediat următor nodului care conține pointerul spre sirul dat și se elimină din listă al n -lea nod care urmează după acest nod.
2. Se execută pasul 1 continuind cu nodul imediat următor celui șters,

pina cind lista se reduce la un singur nod.

Nodul la care s-a redus lista este cel căutat.

Aceasta problema se da adesea ca exemplu pentru utilizarea listelor circulare. O varianta a acestei probleme este aşa numita problema a lui *Josephus*. Ea se formuleaza ca mai jos.

O ceteate este aparata de un numar de soldați care își dă seamă ca au nevoie de ajutorare pentru a rezista în fața dușmanului care îi ataca. Se pune problema de a alege pe unul dintre ei care să plece după ajutor.

Alegerea se face așezind soldații în cerc și tragind la sorti numele soldatului de la care să înceapă numaratoarea. De asemenea, se trage la sorti un număr întreg $n \geq 1$. Se numara, în sensul acelor ceasornicului, începînd cu soldatul următor celui al carui nume a fost tras la sorti și al n -lea soldat este seos din cerc. Se continua numaratoarea în același fel începînd cu soldatul care urmează după cel seos din cerc. În sfîrșit acesta, după un numar finit de pași, cercul se reduce la un singur soldat caruia îl revine sarcina să plece după ajutorare.

Problema lui Josephus este o varianta a formulării descrise prin punctele 1-2 de mai sus, dacă se consideră ca pointerul *cuv* pointează spre numele unui soldat. La punctul 1 se precizează că se pornește cu un nod care urmează imediat nodului care conține pointerul spre un sir dat. Acest sir dat, este chiar numele soldatului tras la sorti, care apare în problema lui Josephus.

Programul de mai jos rezolvă aceasta problema conform urmatorilor pași:

- Citește numărul n indicat în formularea problemei.
- Citește un sir de caractere care definește nodul de la care începe numaratoarea.
- Crează lista circulară care trebuie să conțină un nod pentru care *cuv* pointează spre un sir identic cu cel citit la punctul b.
- Se elimină nodurile listei circulare conform pașilor 1 și 2 indicați mai sus.
- Se afișează cuvintul din nodul rămas în lista.

PROGRAMUL BXI14

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <alloc.h>

typedef struct tnod {
    char *cuv;
    struct tnod *urm;
} TNOD;

#include "bviii2.cpp" /* pcit_int */
#include "bviii3.cpp" /* pcit_int_lim */
#include "bxii0.cpp" /* incnod */
#include "bxii1.cpp" /* elbnod */
#include "bxii2.cpp" /* ccrelist */
```

```
#include "bxii3.cpp" /* cnes */ */

#define MAXN 1000
TNOD *ptrnod;

main() /* - crează o lista circulară și elimină nodurile ei pornind de la un nod dat
        și eliminând tot al n-lea nod pînă cind lista se reduce la un singur nod;
        - în final se listează sirul spre care pointează cuv al nodului
        la care s-a redus lista. */
{
    char t[255];
    int i,n;
    char er[]="s-a tastat EOF\n";
    TNOD *p,*p1;

    /* citeste pe n */
    if (pcit_int_lim("n= ",2,MAXN,&n) == 0 ) {
        printf(er);
        exit(1);
    }

    /* citeste un sir de caractere care va defini nodul din lista pentru pornirea
       numarării nodurilor */
    printf("sirul pentru pornirea numararii\n");
    if (gets(t) == 0 ) {
        printf(er);
        exit(1);
    }

    /* crează lista circulară */
    printf("tastati sirurile care intra în compunerea listei\n");
    printf("cite un sir pe un rind\n");
    printf("la sfîrșit se tasteaza Ctrl-Z\n");
    ccrelist();

    /* se determină nodul pentru care cuv pointează spre un sir identic cu cel
       pastrat în tabloul t */
    if ((p = ccncs(t)) == 0 ) {

        /* nu există un nod pentru care sirul spre care pointează cuv să coincida
           cu cel pastrat în t */
        printf("nu se poate determina nodul de la
               care să se înceapă numararea\n");
        exit(1);
    }

    /* elimină nodurile din lista pînă se ajunge la un singur nod */
    p=p->urm; /* numaratoarea începe cu nodul urmator celui pentru care cuv pointează
                  spre un sir identic cu cel pastrat în t */
    while (ptrnod != ptnod->urm) {

        /* lista conține mai mult de un nod */

```

```

jýý*          /* se cauta al n-lea nod incepind cu cel spre care pointeaza p */
/* se cauta al n-lea nod incepind cu cel spre care pointeaza p */
for( i=1; i<n; i++ ) {
    p1 = p;
    p = p -> urm;
}
/* - se sterge nodul spre care pointeaza p;
   - p1 pointeaza spre nodul precedent nodului spre care pointeaza p. */
p1 ->urm = p -> urm;
if(ptrnod == p) ptrnod = p1;
free( p -> cuv);
free(p);
p = p1 -> urm; /* numaratoarea continua incepind cu
                    nodul urmator celui sters */
}

/* lista s-a redus la un singur nod */
printf("sirul cautat: \n");
printf("%s\n", ptrnod -> cuv );
}

```

11.4. Listā dubļu īnlāntuitā

Atât listele simplu înlăntuite, cât și listele circulare, conduc adesea la parcurgeri neefective ale lor. De exemplu, ștergerea ultimului nod al unei liste simplu înlăntuite implică parcurgerea listei respective (vezi funcția *sm*).

Astfel de parcurgeri pot fi uneori evitate folosind liste dublu înlanțuite. Într-o astfel de listă fiecare nod conține doi pointeri: unul spre nodul *următor* și unul spre nodul *precedent*.

In paragraful de față, vom presupune că nodurile listelor dublu înăncapte au tipul definit ca mai jos:

```

typedef struct tnod {
    declaratii
        struct tnod *prec;
        struct tnod *urm;
} TNOD;

```

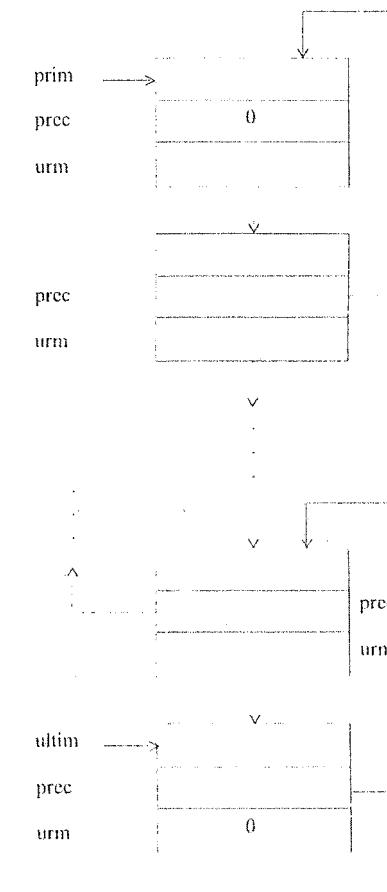
Pentru a gestiona o listă dublu înlanțuită vom utiliza variabilele globale *prim* și *ultim*, ca în cazul listelor simplu înlanțuite. Variabila *prim* pointează spre nodul pentru care *prim-> prec=0*. Pentru restul nodurilor, pointerul *prec* al unui nod pointează spre nodul precedent al listei.

Variabila *ultim* poinează spre un nod pentru care ultim \rightarrow urm=0. Pentru restul nodurilor, pointerul *urm* al unui nod poinează spre nodul următor al listei.

In concluzie, fiecare nod al listei are un nod precedent definit prin pointerul *prec* si un urmator definit prin pointerul *urm*.

O excepție de la această regulă o constituie nodurile spre care pointează variabilele *prim* și *ultim*. Nodul spre care pointează *prim* nu are precedent, iar nodul spre care pointează *ultim* nu are următor. Aceste noduri constituie capetele listei dublu înlántuite.

În figura de mai jos se dă o schemă pentru listele dublu înlántuite.



Listá dublu inlāntuit

În legătura cu listele dublu înlanțuite se pot defini aceleasi operații ca și în cazul listelor simplu înlanțuite:

- a. crearea unei liste dublu înăntuită;
 - b. accesul la un element al unei liste dublu înăntuită;
 - c. inserarea unui nod într-o listă dublu înăntuită;
 - d. stergerea unui nod dintr-o lista dublu înăntuită;

- e. ștergerea unei liste dublu înălțuite.

Operațiile de la punctele *b* și *c* se realizează ca în cazul listelor simplu înălțuite și de aceea ele nu vor mai fi considerate în paragrafele care urmează.

11.4.1. Crearea unei liste dublu înălțuite

Definim mai jos funcția *dcrelist* utilizată pentru a crea o listă dublu înălțuită. Ea este analogă cu funcția *crelist* definită în paragraful 11.1.1.

Funcțiile *incnod* și *elibnod* au aceeași semnificație și utilizare ca în cazul funcției *crelist*.

```
int dcrelist() /* - creaza o lista dublu inlantuita;
                  - returnaza:
                      0 - la eroare;
                      -1 - creare normala.*/
{
    extern TNOD *prim,*ultim;
    int i,n;
    TNOD *p;

    n=sizeof(TNOD);
    prim=ultim=0;
    while(((p=(TNOD *)malloc(n))!=0) && ((i=incnod(p))==1))
        if(prim==0){
            prim=ultim=p;
            p->prec=p->urm=0;
        }
        else{
            ultim->urm=p;
            p->prec=ultim;
            p->urm=0;
            ultim=p;
        }
    if(p==0){
        printf("memorie insuficienta\n");
        exit(1);
    }
    elibnod(p);
    return i;
}
```

11.4.2. Inserarea unui nod într-o listă dublu înălțuită

Într-o listă dublu înălțuită se pot face inserări în diferite poziții. În acest paragraf vom considera cîteva posibilități ca în cazul listelor simplu înălțuite:

- a. inserare înaintea primului nod al listei (nodul spre care pointează variabila *prim* este primul nod al listei);
- b. inserare înaintea unui nod precizat printr-o cheie;

- c. inserare după un nod precizat printr-o cheie;
- d. inserare după ultimul nod al listei (nodul spre care pointează variabila *ultim* este ultimul nod al listei).

11.4.2.1. Inserarea unui nod într-o listă dublu înălțuită înaintea primului nod al ei

```
TNOD *diniprim() /* - inseraza nodul curent inaintea primului nod al listei;
                     - returnaza pointerul spre nodul inserat. */
{
    extern TNOD *prim,*ultim;
    TNOD *p;
    int n;

    n=sizeof(TNOD);
    if(((p=(TNOD *)malloc(n))!=0)&&(incnod(p)==1)){
        if(prim==0){
            prim=ultim=p;
            p->prec = p->urm=0;
        }
        else{
            p->urm = prim;
            p->prec=0;
            prim->prec = p;
            prim=p;
        }
        return p;
    }
    if(p==0){
        printf("memorie insuficienta\n");
        exit(1);
    }
    elibnod(p);
    return 0;
}
```

11.4.2.2. Inserarea unui nod într-o listă dublu înălțuită înaintea unui nod precizat printr-o cheie

Vom presupune că, cheia este de tip *int*. Tipul TNOD se declară astfel:

```
typedef struct tnod {
    declarati
    int cheie;
    declarati
    struct tnod *prec;
    struct tnod *urm;
} TNOD;
```

Funcția de inserare apelează funcția *dnci* care caută într-o listă dublu înălțuită un nod de cheie dată. Această funcție este identică cu funcția *cnci*

definită în paragraful 11.1.2.

```
TNOD *dcnici(int c)
/* - cauta un nod al listei pentru care cheie=c;
   - returneaza pointerul spre nodul determinat in acest fel sau zero daca
     nu exista nici un nod pentru care cheie=c. */
{
    extern TNOD *prim;
    TNOD *p;

    for(p=prim;p;p = p->urm)
        if(p->cheie==c) return p;
    return 0;
}

TNOD *dinicci(int c)
/* - insereaza un nod inaintea unui nod precizat printr-o cheie numérica;
   - returneaza pointerul spre nodul inserat sau zero daca nu are loc inserarea. */
{
    extern TNOD *prim;
    TNOD *p,*q;
    int n;

    /* cauta nodul pentru care cheie=c */
    if((q=dcnici(c))==0){
        printf("nu exista in lista un nod de cheie=%d\n",c);
        return 0;
    }
    n = sizeof(TNOD);
    if(((p=(TNOD *)malloc(n))!=0)&&(incnod(p)==1)){
        p->prec=q->prec;
        p->urm=q;
        if(q->prec!=0)
            /* precedentul lui q are ca urmator nodul inserat */
            q->prec->urm=p;
        q->prec=p;
        if(prim==q)
            /* s-a inserat inaintea primului nod */
            prim=p;
        return p;
    }
    if(p==0){
        printf("memorie insuficienta\n");
        exit(1);
    }
    elibnod(p);
    return 0;
}
```

11.4.2.3. Inserarea unui nod într-o listă dublu înlățuită după unul precizat printr-o cheie

Vom presupune că, cheia este de tip *int*, iar tipul TNOD este cel declarat în

paragraful precedent. De asemenea, pentru a localiza nodul de cheie precizată, se va utiliza funcția *denci* definită în același paragraf.

```
TNOD *dindci(int c)
/* - inserează un nod după unul precizat printr-o cheie numerică;
   - returnează pointerul spre nodul inserat sau zero daca inserarea nu are loc. */
{
    extern TNOD *ultim;
    TNOD *p,*q;
    int n;

    if((q=dcnici(c))==0){
        printf("nu exista nodul de cheie=%d\n",c);
        return 0;
    }
    n = sizeof(TNOD);
    if(((p=(TNOD *)malloc(n))!=0)&&(incnod(p)==1)){
        p->prec=q;
        p->urm=q->urm;
        if(q->urm!=0)
            /* urmatorul lui q are ca și precedent nodul inserat */
            q->urm->prec=p;
        q->urm=p;
        if(ultim==q)
            /* s-a inserat după ultimul nod */
            ultim=p;
        return p;
    }
    if(p==0){
        printf("memorie insuficienta\n");
        exit(1);
    }
    elibnod(p);
    return 0;
}
```

11.4.2.4. Inserarea unui nod într-o listă dublu înlățuită după ultimul nod (adăugarea unui nod la o listă dublu înlățuită)

În acest caz tipul nodurilor se declară astfel:

```
typedef struct nod {
    declaratii
    struct nod *prec;
    struct nod *urm;
} TNOD;
```

nefiind necesară prezența unei chei.

```
TNOD *adauga()
/* - adauga un nod la o lista dublu inlantuita;
   - returneaza pointerul spre nodul inserat sau zero daca nu se realizeaza inserarea. */
{
```

```

extern TNOD *prim,*ultim;
TNOD *p;
int n;

n=gizeof(TNOD);
if((p=(TMOD *)malloc(n))!=0)&&(incnod(p)==1)){
    if(prim==0){
        prim=ultimo=p;
        p->prec=p->urm=0;
    }
    else{
        ultimo->urm=p;
        p->prec=ultimo;
        p->urm=0;
        ultimo=p;
    }
    return p;
}
if(p==0){
    printf("memorie insuficienta\n");
    exit(1);
}
elibnod(p);
return 0;
}

```

11.4.3. Stergerea unui nod dintr-o listă dublu înlanțuită

Dintr-o listă dublu înlanțuită se pot șterge noduri. În cele ce urmăza vom avea în vedere următoarele cazuri:

- stergerea primului nod al unei liste dublu înlanțuite;
- stergerea unui nod precizat printr-o cheie;
- stergerea ultimului nod al unei liste dublu înlanțuite.

11.4.3.1. Stergerea primului nod al unei liste dublu înlanțuite

```

void dspn() /* sterge primul nod din lista */
{
    extern TNOD *prim,*ultimo;
    TNOD *p;

    if(prim==0) return;
    p=prim;
    prim=prim->urm;
    elibnod(p);
    if(prim==0) ultimo=0; /* lista a devenit vida */
    else prim->prec=0;
}

```

11.4.3.2. Stergerea unui nod dintr-o listă dublu înlanțuită precizat printr-o cheie

Vom considera că, cheia este de tip *int*. În acest caz tipul TNOD se declară ca în paragraful 11.4.2.2.

Funcția de față apelează funcția *dnci*, definită în același paragraf, pentru a localiza nodul care urmează a fi sters.

```

void dnci(int c) /* sterge nodul de cheie=c */
{
    extern TNOD *prim,*ultimo;
    TNOD *p;

    if(prim==0) /* lista vida */
        return;
    if((p=dnci(c))==0){
        printf("lista nu contine nodul de cheie = %d\n",c);
        return;
    }
    if(prim==p&&ultimo==p){
        /* lista are un singur nod; devine vida */
        prim=ultimo=0;
        elibnod(p);
        return;
    }
    if(prim==p) /* se sterge primul nod din lista */
        prim=prim->urm;
        prim->prec=0;
        elibnod(p);
        return;
    if(ultimo==p) /* se sterge ultimul nod */
        ultimo=ultimo->prec;
        ultimo->urm=0;
        elibnod(p);
        return;
    }

    /* se sterge un nod diferit de capete */

    /* urmatorul nodului care se sterge are ca precedent, precedentul nodului care se sterge */
    p->urm->prec=p->prec;

    /* precedentul nodului care se sterge are ca urmator, urmatorul nodului care se sterge */
    p->prec->urm=p->urm;
    elibnod(p);
}

```

11.4.3.3. Stergerea ultimului nod al unei liste dublu înlanțuite

În acest caz tipul TNOD nu necesită prezența unei chei. Se poate utiliza declarația indicată în paragraful 11.4.2.4.

```

void dsun() /* sterge ultimul nod din lista */
{
    extern TNOD *prim, *ultim;
    TNOD *p;

    if(prim==0) return;
    p=ultim;
    ultim=ultim -> prec;
    if(ultim==0) prim=0; /* lista devine vida */
    else ultim -> urm=0;
    elibnod(p);
}

```

Observații:

1. Funcția *dsun*, spre deosebire de funcția *sun* relativă la listele simplu înlățuite, devine eficientă deoarece ea nu mai necesită parcurgerea nodurilor listei.
2. Funcțiile de inserare înainte și după un nod precizat printr-o cheie numerică întreagă sunt mai simple în cazul listelor dublu înlățuite în comparație cu analogele lor pentru liste simplu înlățuite deoarece ele folosesc funcția *dcnci* pentru localizarea nodului în raport cu care se face inserarea.
3. Funcția de stergere a unui nod precizat printr-o cheie numerică întreagă este mai simplă în cazul listelor dublu înlățuite în comparație cu funcția corespunzătoare pentru liste simplu înlățuite deoarece ea folosește funcția *dcnci* pentru localizarea nodului care se sterge.
4. O listă dublu înlățuită devine o *listă circulară dublu înlățuită* dacă se fac atribuirile:
 - ultim -> urm=prim;
 - și
 - prim -> prec=ultim.

Pentru a gestiona o astfel de listă nu mai sunt necesare variabilele *prim* și *ultim*, lista ne mai avind capete. În locul lor se utilizează un pointer spre un nod arbitrar al listei, ca în cazul listelor circulare simplu înlățuite.

În legătură cu listele circulare dublu înlățuite se ridică același probleme ca și în cazul listelor circulare simplu înlățuite.

Propunem cititorului să construiască funcții pentru a realiza operațiile de bază asupra listelor circulare dublu înlățuite:

- creare;
- acces la un nod de cheie dată;
- inserări de noduri;
- stergeri de noduri;
- stergerea listei.

Exerciții:

- 11.15 Se consideră tipul utilizator:

```

typedef struct tnod {
    char *cuvant;
    int frecventa;
    struct tnod *prec;
    struct tnod *urm;
} TNOD

```

Să se scrie funcția *dadauga*, care permite adăugarea la o listă dublu înlățuită a unui nod de tipul TNOD definit ca mai sus.

Se observă că funcția *dadauga* definită în paragraful 11.4.2.4, este dependentă numai de componentele *prec* și *urm* din TNOD. De aceea, funcția *dadauga* de mai jos este identică cu cea din paragraful 11.4.2.4.

FUNCȚIA BXI15

```

TNOD *dadauga ()
/* - adauga un nod la o lista dublu inlantuita;
   - returneaza pointerul spre nodul inserat sau zero daca nu se realizeaza inserarea. */
{
    extern TNOD *prim, *ultim;
    TNOD *p;
    int n;

    n = sizeof(TNOD);
    if(((p = (TNOD *)malloc(n)) != 0) && (incnod(p) == 1)) {
        if( prim == 0 ) {
            prim = ultim = p;
            p -> prec = p -> urm = 0;
        }
        else {
            ultim -> urm = p;
            p -> prec = ultim;
            p -> urm = 0;
            ultim = p;
        }
        return p;
    }
    if( p == 0 ) {
        printf("memorie insuficienta\n");
        exit(1);
    }
    elibnod(p);
    return 0;
}

```

- 11.16 Să se scrie o funcție care sterge ultimul nod al unei liste dublu înlățuite ale cărei noduri au tipul TNOD definit în exercițiul precedent.

Aceasta funcție este definită în paragraful 11.4.3.3.

FUNCȚIA BXI16

```
void dsun() /* sterge ultimul nod din lista */
{
    extern TNOD *prim, *ultim;
    TNOD *p;

    if(prim == 0) return;
    p = ultim;
    ultim = ultim -> prec;
    if(ultim == 0) prim = 0;
    else ultim -> urm = 0;
    elibnod(p);
}
```

- 11.17 Să se scrie un program care citește cuvintele dintr-un text și afișează numărul de apariții al fiecarui cuvânt din textul respectiv.

Această problemă a mai fost rezolvată în exercițiile 10.50 și 11.6. În exercițiul 11.6 s-a creat o listă simplu înlățuită ale cărei noduri conțin pointeri spre cuvintele diserite din text, precum și frecvența de apariție a acestora în text.

În cazul de față, se utilizează o listă dublu înlățuită. Prin aceasta, eficiența programului este mai mare deoarece funcția *dsun*, care sterge ultimul nod dintr-o listă dublu înlățuită, este mult mai eficientă decât funcția *sun* care realizează același lucru relativ la o listă simplu înlățuită.

Programul de față utilizează o parte din funcțiile apelate de programul BXI6.CPP și anume:

- *citcuv*;
- *incnod*;
- *elibnod*;
- și
- *cnccs*.

Funcțiile *adauga* și *sun* se înlocuiesc cu funcțiile *dadauga* și respectiv *dsun*. De asemenea, TNOD se declară ca în exercițiul 11.15.

PROGRAMUL BXI17

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <string.h>

typedef struct tnod {
    char *cuvant;
    int frecventa;
    struct tnod *prec;
    struct tnod *urm;
} TNOD;

#include "bx48.cpp" /* citcuv */
#include "bxi11.cpp" /* incnod */
```

```
#include "bxi2.cpp" /* elibnod */
#include "bxi15.cpp" /* dadauga */
#include "bxi4.cpp" /* cnccs */
#include "bxi16.cpp" /* dsun */

TNOD *prim, *ultim;

main() /* citește un text și afișează frecvența cuvintelor din textul respectiv */
{
    TNOD *p, *q;

    prim = ultim = 0;
    while((p = dadauga()) != 0)
        if((q = cnccs(p -> cuvant)) != ultim)
            q -> frecventa++;
        dsun();
    }
    for( p = prim; p; p = p -> urm)
        printf("cuvintul: %s are frecvența: %d\n",
               p -> cuvant, p -> frecventa);
}
```

- 11.18 Să se scrie un program care citește cuvintele dintr-un text și scrie numărul de apariție al fiecarui cuvânt, în ordinea alfabetă a cuvintelor respective.

Această problemă a fost rezolvată în exercițiul 11.7. În exercițiul respectiv s-a definit funcția *ordlist* care s-a apelat înainte de a afișa frecvența cuvintelor citite. Ea a modificat înlățuirile nodurilor listei simplu înlățuite create prin citirea cuvintelor textului, în aşa fel încât cuvintele corespunzătoare nodurilor listei să fie ordonate alfabetic.

Programul definit în exercițiul 11.7, diferă de cel definit în exercițiul 11.6, prin prezența funcției *ordlist* și apelul ei înainte de listarea rezultatului.

În exercițiul de față se definește funcția *dordlist* care realizează același lucru ca și funcția *ordlist*, adică modifică înlățuirile nodurilor unei liste dublu înlățuite.

Amintim că funcția *ordlist* parcurge lista analizând ordinea cuvintelor corespunzătoare nodurilor vecine. Dacă două cuvinte, care corespund la noduri vecine, nu sunt în ordine alfabetă, atunci se schimbă înlățuirile nodurilor respective astfel încât ele să fie înlățuite invers în listă.

Inversarea înlățuirilor se realizează ca mai jos.

Presupunem că *p* pointează spre nodul curent din listă și *q=p->urm*, deci *q* pointează spre nodul următor din listă.

Dacă *p->cuvant* pointează spre un cuvânt care este în ordine alfabetă după cuvintul spre care pointează *q->cuvant*, atunci nodurile *p* și *q* se înlățuiesc invers, deci *p* devine următorul lui *q*. Aceasta implică pașii:

1. Dacă *p->prec!=0*, atunci *p->prec->urm=q*, deoarece următorul precedentalui lui *p* devine *q*.
2. Dacă *q->urm!=0*, atunci *q->urm->prec=p*, deoarece precedentul ur-

mătorului lui q devine p .

3. $p \rightarrow \text{urm} = q \rightarrow \text{urm}$, deoarece următorul lui q devine următorul lui p .
4. $q \rightarrow \text{urm} = p$, deoarece p devine următorul lui q .
5. $q \rightarrow \text{prec} = p \rightarrow \text{prec}$, deoarece precedentul lui p devine precedentul lui q .
6. $p \rightarrow \text{prec} = q$, deoarece q devine precedentul lui p .

PROGRAMUL BXI18

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <string.h>

typedef struct tnod {
    char *cuvant;
    int frecventa;
    struct tnod *prec;
    struct tnod *urm;
} TNOD;

#include "bx48.cpp" /* citcuv */
#include "bx11.cpp" /* incnod */
#include "bxi2.cpp" /* elibnod */
#include "bxi15.cpp" /* dadauga */
#include "bxi4.cpp" /* cnec */
#include "bxi16.cpp" /* dsun */

void dordlist()
/* ordonaza nodurile listei de tip TNOD in asa fel incit cuvintele
   corespunzatoare nodurilor sa fie in ordine alfabetica */
{
    extern TNOD *prim,*ultim;
    TNOD *p,*q;
    int ind;
    if(prim == 0) return ;
    ind = 1;
    while(ind) {
        ind = 0;
        for(p=prim; p->urm; ) {
            q = p->urm; /* q este urmatorul lui p */
            if(strcmp(p->cuvant, q->cuvant) > 0){

                /* se inverseaza inlantuirile nodurilor p si q */
                if(p->prec) p->prec->urm=q;
                if(q->urm) q->urm->prec=p;
                p->urm=q->urm;
                q->urm=p;
                q->prec=p->prec;
                p->prec=q;
                if(p == prim) prim = q;
                if(q == ultim) ultim = p;
                ind = 1;
            }
        }
    }
}
```

```
else /* se trece la nodul urmator deoarece cuvintele sunt in ordine alfabetica */
    p = q;
} /* sfirsit for */
} /* sfirsit while */
} /* sfirsit dordlist */

TNOD *prim,*ultim;

main() /* citeste un text si afiseaza frecventa cuvintelor diferite in ordine alfabetica */
{
    TNOD *p,*q;

    prim = ultim = 0;
    while((p = dadauga()) != 0)
        if((q = cnec(p->cuvant)) != ultim) {
            q->frecventa++;
            dsun();
        }
    dordlist();
    for( p = prim; p; p=p->urm)
        printf("cuvintul: %-5ls are frecventa: %d\n",
               p->cuvant, p->frecventa);
}
```

12. ARBORI

Arborii, ca și listele, sunt structuri de date de natură recursivă și dinamică. În multe publicații de specialitate (vezi de exemplu [3]) arborele se definește recursiv.

Prin arbore înțelegem o mulțime finită și nevidată de elemente numite noduri:

$$A = \{A_1, A_2, \dots, A_n\}, n > 0$$

care au următoarele proprietăți:

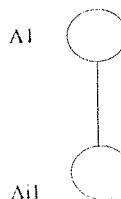
- Există un nod și numai unul care se numește *rădăcina* arborelui.
 - Celelalte noduri formează submulțimi disjuncte ale lui A, care formează fiecare cîte un arbore. Arboarele respectiv î se numesc *subarbori* ai rădăcinii.

Un arbore poate fi reprezentat intr-un plan. Nodurile se reprezintă prin cercuri. Așa de exemplu, dacă A1 este rădăcina arborelui A, atunci figurăm în plan un cerc pe care îl marcăm cu A1.

Fie acum:

$$\{A_{i1}, A_{i2}, \dots, A_{ik}\}$$

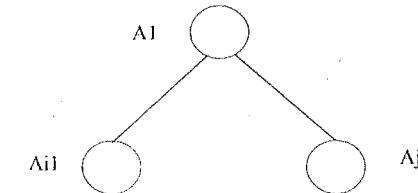
un subarbore a lui A_1 . Deoarece acesta este el insuși un arbore, fie A_{11} rădâcina lui. Lui A_{11} ii corespunde un cerc pe care îl marcăm cu A_{11} . Acest cerc se figurează sub cercul corespunzător lui A_1 . Cele două cercuri se unesc ca în figura de mai jos.



In mod analog, dacă:

$$\{A_{j1}, A_{j2}, \dots, A_{jr}\}$$

este un alt subarbore al rădăcinii A1 și Aj1 este rădăcina acestui subarbore, atunci lui Aj1 ii corespunde un cerc care se trasează sub cercul corespunzător lui A1. De asemenea, cercul marcat cu A1 se unește și cu cel marcat cu Aj1. De obicei, cercurile marcate cu A1 și Aj1 se află pe aceeași orizontală, ca în figura de mai jos.



Acest proces continuă cu toți subbarorii rădăcinii A1. Apoi se continuă cu subbarorii lui A1₁, A1₂ și asa mai departe.

Într-un arbore există noduri cărora nu le mai corespund subarbore. Un astfel de nod se numește *terminal* sau *frunză*.

În legătură cu arborii s-a incetătenit un limbaj conform căruia un nod rădăcină se spune că este un nod *tată*, iar subarborii rădăcinii sunt descendenții acestuia. Rădacinile descendenților unui nod tată sunt *fiii* lui. Astfel, în exemplul de mai sus, A1 este un nod tată, iar A1 și A1 sint fii ai acestuia.

Deci, dacă A este un nod rădăcină și dacă acesta are p subarbori ale căror rădăcini sunt:

B1,B2,...,Bp

atunci A este un nod *tată*, iar B₁, B₂, ..., B_p sunt *fiile* lui

Despre nodurile B_1, B_2, \dots, B_p se spune că sunt *frateli*.

Numărul fiilor unui nod tată este *ordinul* nodului tată respectiv.

În exemplul de mai sus, nodul A are ordinul p . Un nod frunză nu are fiți, deci ordinul lui este egal cu zero.

O altă noțiune legată de arbori este noțiunea de *nivel*. Rădăcina unui arbore are nivelul 1. Dacă un nod are nivelul n , atunci fiile lui au nivelul $n+1$.

Dacă pentru fiecare nod, subarborei săi sunt ordonați (se consideră într-o anumită ordine), atunci arborele se numește *ordonat*. În acest caz se obișnuiește să se spună că rădăcina primului subarbore este *fiul cel mai în vîrstă*, iar rădăcina ultimului subarbore este *fiul cel mai tânăr*.

În multe aplicații practice intîlnim aşa numiții *arbori binari*. Un arbore binar este o mulțime finită de elemente care sau este *vidă* sau conține un element numit *rădăcină*, iar celealte elemente se imparte în două submulțimi disjuncte, care fiecare la rândul ei, este un arbore binar. Una din submulțimi este numită *subarborele stîng* al rădăcinii, iar cealaltă *subarborele drept*. Arborele binar este ordonat, deoarece în fiecare nod, subarborele stîng se consideră că precede subarborele drept. De aici rezultă că un nod al unui arbore binar are cel mult doi fiți și că unul este *fiul stîng*, iar celălalt este *fiul drept*. Fiul stîng este mai în vîrstă decât cel drept. Un nod al unui arbore binar poate să aibă numai un *descendent*.. Acesta poate fi subarborele stîng sau subarborele drept.

Cele două posibilități se consideră distințe. Cu alte cuvinte, dacă doi arbori binari diferă numai prin aceea că nodul A dintr-un arbore are ca descendente numai fiul sting, iar același nod din celălalt arbore are ca descendente numai fiul

drept, cei doi arbori se consideră distincți.

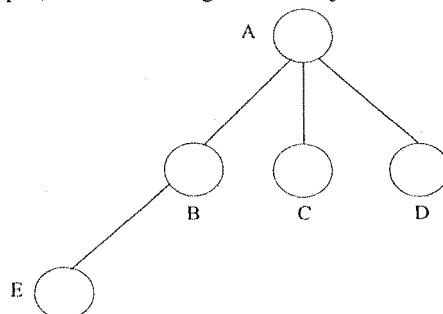
Un arbore binar *nu* se definește ca un caz particular de arbore ordonat. Astfel, un arbore nu este niciodată vid, spre deosebire de un arbore binar care poate fi și vid.

Un arbore ordonat poate fi totdeauna reprezentat printr-un arbore binar (vezi [3]).

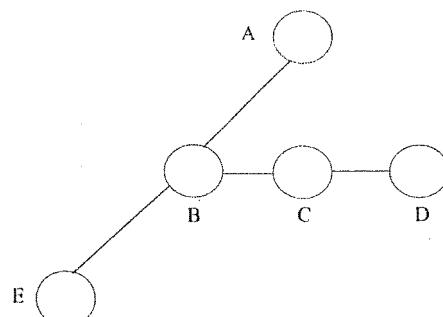
Transformarea se obține destul de simplu și anume:

Se leagă împreună frații descendenți ai unui același nod tată și se suprimă legăturile lor cu nodul tată, exceptând legătura primului dintre ei.

De exemplu, arborele din figura de mai jos:



se transformă în arborele binar:



În arborele transformat:

A are pe B ca fiu stîng;

B are pe C ca fiu drept și pe E ca fiu stîng;

C are pe D ca fiu drept.

Prin regula de mai sus, rezultă că primul fiu al unui nod tată devine fiul stîng (cel mai vîrstnic). Celelalte legături formează subarborei drepti: al doilea fiu devine fiul drept al primului, al treilea fiu devine fiul drept al celui de al doilea și

așa mai departe.

În continuare ne vom ocupa numai de arborii binari.

Un nod al unui arbore binar este o dată structurată de tipul TNOD care se definește în felul următor:

```
typedef struct tnod {  
    declaratii  
    struct tnod *st;  
    struct tnod *dr;  
} TNOD;
```

unde:

- st* - Este pointerul spre fiul stîng al nodului curent.
- dr* - Este pointerul spre fiul drept al aceluiași nod.

Asupra arborilor binari se pot defini mai multe operații dintre care amintim:

- inserarea unui nod frunză într-un arbore binar;
- accesul la un nod al unui arbore;
- parcurgerea unui arbore;
- ștergerea unui arbore.

Operațiile de *inserare* și *accesul* la un nod, au la bază un *criteriu* care să definească locul în arbore al nodului în cauză. Acest criteriu este dependent de problema concretă la care se aplică arborii binari pentru a fi rezolvată. El se definește printr-o funcție pe care o vom numi în continuare funcția *criteriu*. Ea are doi parametri care sunt pointeri spre tipul TNOD. Fie *p1* primul parametru al funcției *criteriu* și *p2* cel de al doilea parametru al ei. Atunci, funcția *criteriu* returnează:

- 1 - Dacă *p2* pointează spre o dată de tip TNOD care poate fi un nod al subarborelui stîng al nodului spre care pointează *p1*.
- 1 - Dacă *p2* pointează spre o dată de tip TNOD care poate fi un nod al subarborelui drept al nodului spre care pointează *p1*.
- 0 - Dacă *p2* pointează spre o dată de tip TNOD care nu poate fi nod al subarborelor nodului spre care pointează *p1*.

Spre exemplificare, să considerăm ca la intrare se află sirul de intregi:

20, 30, 5, 20, 4, 30, 7, 40,

și se pună problema de a determina numărul de apariții al fiecărui.

Acastă problemă se poate rezolva dacă se construiește un arbore binar ale cărui noduri au tipul TNOD definit astfel:

```
typedef struct tnod {  
    int nr;  
    int f;  
    struct tnod *st;
```

```

struct tnod *dr;
} TNOD;

```

Variabila *nr* are ca valoare unul din numerele citite de la intrare, iar *f* reprezintă frecvența lui de apariție în șirul de la intrare.

Într-un nod frunză *st=dr=0* (pointerul nul).

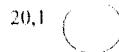
La început se citește valoarea 20 și arborele va conține un singur nod, corespunzător valorii 20:

```

nr=20
f=1
st=dr=0

```

Figurăm arborele de la această fază astfel:



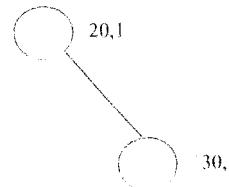
La pasul următor se citește întregul 30 și se construiește data de tip TNOD corespunzătoare lui:

```

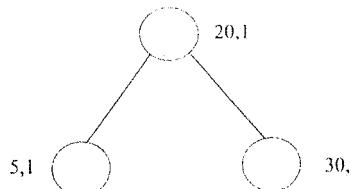
nr=30
f=1
st=dr=0

```

Această dată o vom insera ca fiu drept al radacinii arborelui existent de la pasul precedent. Se obține arborele:



Apoi se citește valoarea 5 și nodul corespunzător acestia se inserează ca fiu stîng al nodului corespunzător valorii 20. Se obține arborele:

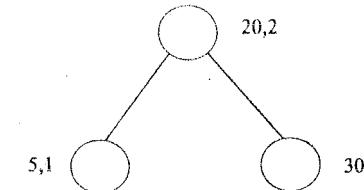


Nodurile se inserează în arbore în aşa fel încit dacă un nod corespunde intregului *n*, atunci subarborele stîng al lui conține noduri care corespund la

valori mai mici decît *n*, iar subarborele drept al aceluiași nod conține noduri care corespund la valori mai mari decît *n*.

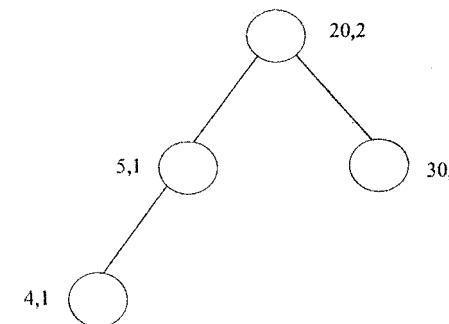
Se observă că arborele de mai sus respectă această regulă. Într-adevăr, rădâcina arborelui corespunde valorii 20, subarborele stîng al ei conține, nodul corespunzător valorii 5 ($5 < 20$), iar subarborele drept al rădăcinii conține nodul corespunzător valorii 30 ($20 < 30$).

În continuare se citește valoarea 20 și cum există deja în arbore un nod corespunzător acestei valori, se incrementează valoarea lui *f* din nodul respectiv. În felul acesta se obține reprezentarea:



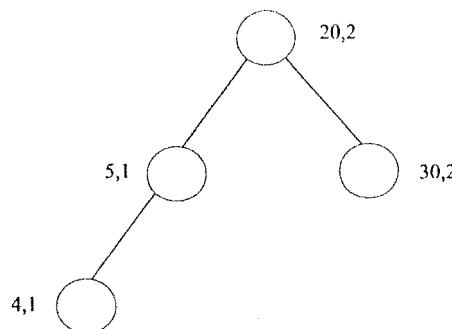
La pasul următor se citește valoarea 4. Se construiește nodul corespunzător valorii 4. Deoarece $4 < 20$, nodul curent trebuie inserat în subarborele stîng. Acesta este format din nodul corespunzător valorii 5.

Cum $4 < 5$, rezultă că nodul corespunzător lui 4 trebuie inserat în subarborele stîng al nodului corespunzător lui 5. Cum nu există un astfel de subarbore, rezultă că nodul corespunzător lui 4 devine fiu stîng al nodului corespunzător lui 5. Se obține reprezentarea:



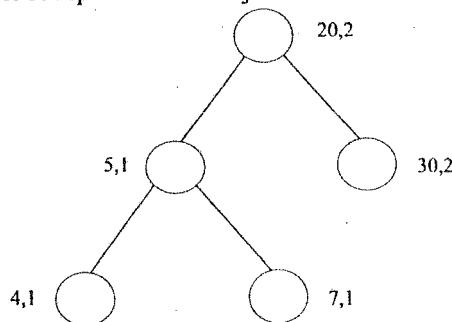
Valoarea următoare citită de la intrare este 30. Ca și în cazurile precedente, se construiește nodul corespunzător valorii citite 30.

Cum $30 > 20$, rezultă că nodul construit curent trebuie să se insereze în subarborele drept al nodului corespunzător lui 20. Acest subarbore este format dintr-un singur nod care corespunde lui 30. Întrucât nodul curent corespunde aceleiași valori, rezultă că el nu se mai inserează în arbore ci pur și simplu se incrementează valoarea lui *f* pentru nodul corespunzător din arbore. Se obține reprezentarea:



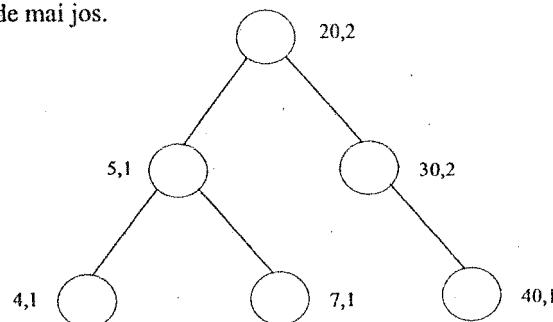
Următorul număr citit este 7. Se construiește nodul corespunzător acestei valori. Deoarece $7 < 20$, rezultă că nodul curent se va insera în subarborele stîng al nodului corespunzător lui 20. Rădăcina acestui subarbore este nodul corespunzător lui 5. Cum $5 < 7$, rezultă că nodul curent se va insera în subarborele drept al nodului corespunzător lui 5. Întrucît nu există un astfel de subarbore, nodul curent se va insera ca fiu drept al nodului corespunzător lui 5.

Arborele se reprezintă ca mai jos.



Ultima valoare citită este 40.

Răționind ca mai sus, se ajunge la concluzia că nodul corespunzător valorii 40 se inserează ca fiu drept al nodului corespunzător lui 30. În final se obține arborele de mai jos.



Se observă că arborele construit respectă regula enunțată mai sus.

Într-adevar, nodul corespunzător lui 20 ($n=20$) are un subarbore stîng ale căruia noduri corespund valorilor: 5, 4 și 7. Aceste valori sunt toate mai mici decît 20. Același nod, are un subarbore drept ale căruia noduri corespund valorilor: 30 și 40. Ambele sunt mai mari decît 20.

Această proprietate are loc și pentru celelalte noduri care nu sunt noduri frunză. De exemplu, nodul corespunzător valorii 5, are ca subarbore stîng nodul corespunzător lui 4 ($4 < 5$), iar ca subarbore drept nodul corespunzător lui 7 ($5 < 7$).

La construirea arborelui din acest exemplu s-a utilizat următorul criteriu pentru determinarea poziției în care să se insereze în arbore nodul curent (nodul corespunzător valorii curent citite):

- $p1$ este pointer spre un nod al arborelui în care se face inserarea (înital $p1$ pointează spre radacina arborelui).
- $p2$ este pointer spre nodul curent (nodul de inserat).
- Dacă $p2 \rightarrow nr < p1 \rightarrow nr$, atunci se va încerca inserarea nodului curent în subarborele stîng al nodului spre care pointează $p1$.
- Dacă $p2 \rightarrow nr > p1 \rightarrow nr$, atunci se va încerca inserarea nodului curent în subarborele drept al nodului spre care pointează $p1$.
- Dacă $p2 \rightarrow nr = p1 \rightarrow nr$, atunci nodul curent nu se mai inseră în arbore deoarece există deja un nod corespunzător valorii curent citite.

În acest caz se incrementează $p1 \rightarrow f$.

Acest criteriu se realizează imediat printr-o funcție care are ca parametri pointerii $p1$ și $p2$ și care returnează valorile:

- 1 - În cazul descriș la punctul c.
- 1 - În cazul descriș la punctul d.
- 0 - În cazul descriș la punctul e.

Funcția se definește ca mai jos:

```

int criteriu(TNODE *p1, TNODE *p2)
{
    if(p2 -> nr < p1 -> nr) return -1;
    if(p2 -> nr > p1 -> nr) return 1;
    return 0;
}

```

Din cele de mai sus rezultă că nodul curent nu se mai inseră în arbore, în situația descrișă de punctul e, situație care în general corespunde cazului cînd funcția *criteriu* returnează valoarea zero. În acest caz nodurile spre care pointează $p1$ și $p2$ le vom considera *equivalente*.

De obicei, la întîlnirea unei perechi de noduri echivalente, nodul din arbore (spre care pointează $p1$) este supus unei prelucrări, iar nodul curent (spre care pointează $p2$) este eliminat. Pentru a realiza o astfel de prelucrare este necesar să se apeleze o funcție care are ca parametri pointerii $p1$ și $p2$ și care returnează un

pointer spre tipul TNOD (de obicei se returneaza valoarea lui *p1*).

Vom numi aceasta functie *echivalenta*. Ea este dependenta de problema concreta, ca si functia *criteriu*.

Pentru exemplul de mai sus, functia *echivalenta* se defineste astfel:

```
TNOD *echivalenta(TNOD *p1, TNOD *p2)
{
    p1->i>1;
    elibnod(p2);
    return p1;
}
```

Functia *elibnod* este o alta functie dependenta de problema concreta si care elibereaza zonele de memorie ocupate de nodul spre care pointeaza parametrul ei. Aceasta functie a fost intilnita in capitolul precedent si s-a utilizat in același scop pentru nodurile din compunerea listelor.

In sfirșit, o alta functie dependenta de problema concreta este functia *incnod*, utilizata si ea in functiile din capitolul precedent.

Ea se apeleaza pentru a incarca datele in nodul curent care urmeaza sa fie inserat in arbore sau pentru a fi prelucrat de functia *echivalenta* daca nu are loc inserarea.

In continuare se definesc functii pentru operatiile asupra arborilor, amintite mai sus. Toate functiile utilizeaza o variabila globala care este un pointer spre radacina arborelui. Numim *prad* acesta variabila. Ea se defineste astfel:

```
TNOD *prad;
```

In cazul in care intr-un program se prelucreaza simultan mai multi arbori, nu se va mai utiliza variabila globala *prad*, interfata dintre functii realizandu-se cu ajutorul unui parametru care este pointer spre tipul TNOD si caruia i se atribuie, la apel, adresa nodului radacina al arborelui prelucrat prin functia apelata.

In paragrafele urmatoare se definesc functii care utilizeaza variabila globala *prad*.

Mentionam ca functiile *incnod*, *criteriu*, *echivalenta* si *elibnod* pot sa aiba si alte denumiri, facind modificarri corespunzatoare in functiile care le apeleză. De asemenea, aceste functii pot fi utilizate prin intermediul parametrilor.

Propunem cititorului sa realizeze functiile din paragrafele acestui capitol in aza fel incit functiile *incnod*, *criteriu*, *echivalenta* si *elibnod* sa fie apelate prin intermediul parametrilor de tip pointer spre functie.

12.1. Inserarea unui nod frunză într-un arbore binar

Arboarele in care se insereaza nodul este definit de variabila globala *prad* care are ca valoare adresa de inceput a zonei de memorie in care se păstreaza radacina arborelui.

In cazul in care arborele este vid, *prad* are valoarea zero. Inserarea nodului se realizeaza conform urmatorilor pași:

1. Se alocă zonă de memorie pentru nodul care urmează să se insereze în arbore. Fie *p* pointerul care are ca valoare adresa de inceput a zonei respective.
2. Se apelează funcția *incnod*, cu parametrul *p*, pentru a încărca datele curente în zona spre care pointează *p*. Dacă *incnod* returnează valoarea 1, se trece la pasul 3. Altfel se revine din funcție cu valoarea zero.
3. Se fac atribuirile:
 $p \rightarrow st = p \rightarrow dr = 0$
deoarece nodul de inserat este nod frunză.
4. $q = prad$.
5. Se determină poziția, în arbore, în care să se facă inserarea. În acest scop se caută nodul care poate fi nod tată pentru nodul curent.
 $i = criteriu(q, p)$.
6. Dacă $i < 0$, se trece la pasul 7. Altfel se trece la pasul 8.
7. Se încearcă inserarea nodului spre care pointează *p* (nodul curent) în subarborele stîng al nodului spre care pointează *q*. Dacă *q* \rightarrow *st* are valoarea zero, atunci nodul spre care pointează *q* nu are subarbore stîng și nodul curent devine fiu stîng al celui spre care pointează *q* (*q* \rightarrow *st* = *p*). Se revine din funcție returnindu-se valoarea lui *p*. Altfel se face atribuirea *q=q* \rightarrow *st* (se trece la fiul stîng al nodului spre care pointează *q*) și se trece la pasul 5.
8. Dacă $i > 0$, se trece la pasul 9. Altfel se trece la pasul 10.
9. Se încearcă inserarea nodului spre care pointează *p* (nodul curent) în subarborele drept al nodului spre care pointează *q*. Dacă *q* \rightarrow *dr* are valoarea zero, atunci nodul spre care pointează *q* nu are subarbore drept și nodul curent devine fiu drept al celui spre care pointează *q* (*q* \rightarrow *dr* = *p*). Se revine din funcție returnindu-se valoarea lui *p*. Altfel se face atribuirea *q=q* \rightarrow *dr* (se trece la fiul drept al nodului spre care pointează *q*) și se trece la pasul 5.
10. Nodul curent nu poate fi inserat în arbore. Se apelează funcția *echivalenta* și se revine din funcție cu valoarea returnată de funcția *echivalenta*.

Definim mai jos funcția care realizează pașii 1-10.

```
TNOD *insnod()
/* - inscreaza un nod in arborele binar spc a carui radacina pointeaza prad;
   - returneaza pointerul spc nodul inserat, pointerul returnat de functia echivalenta sau zero
   daca nu sunt date de incarcat in nod sau la croare. */
```

```

{
    extern TNOD *prad;
    int i;
    int n;
    TNOD *p, *q;

    n = sizeof(TNOD);
    if((p=(TNOD *)malloc(n))!=0) && (incnod(p)==1)) { /* 1 */
        p->st=p->dr=0;
        if(prad==0) { /* arbore vid */
            prad=p; /* nodul curent devine radacina arborelui */
            return p;
        }

        /* se determina pozitia in arbore a nodului si se face inserarea daca este cazul */
        q=prad;
        for(;;){
            if((i=criteriu(q,p))<0)
                if(q->st==0) { /* se insereaza ca fiu stang */
                    q->st=p;
                    return p;
                }
            else { /* se continua cautarea pozitiei in subarborele stang */
                q=q->st;
                continue;
            }
            if(i>0)
                if(q->dr==0) { /* se insereaza ca fiu drept */
                    q->dr=p;
                    return p;
                }
            else { /* se continua cautarea pozitiei in subarborele drept */
                q=q->dr;
                continue;
            }
        }
        /* - nu se face inserarea deoarece nu exista un nod in arbore pentru care
           nodul curent sa poata fi nod fiu (stang sau drept);
           - se apeleaza functia echivalenta si se revine din functie. */
        return echivalenta (q,p);
    } /* sfarsit for */
} /* sfarsit if 1 */

if(p==0){
    printf("memorie insuficienta\n");
    exit(1);
}

/* eroare la incarcarea datelor in nod sau nu mai sunt date */
elibnod(p);
return 0;
} /* sfarsit insnod */
}

```

12.2. Accesul la un nod al unui arbore binar

Accesul la un nod al unui arbore binar presupune existența unui criteriu care să permită localizarea în arbore a nodului respectiv. De obicei, se poate utiliza funcția *criteriu* folosită la crearea arborelui. Cu ajutorul ei se poate realiza accesul la un nod din arbore echivalent cu un nod dat.

```

TNOD *cauta (TNOD *p)
/* - cauta in arborele spre a carui radacina pointeaza prad, nodul
   echivalent cu cel spre care pointeaza p;
   - returneaza pointerul spre nodul determinat sau zero daca nu exista un astfel de nod */
{
    extern TNOD *prad;
    TNOD *q;
    int i;

    if(prad==0) return 0; /* arbore vid */
    for(q=prad;q;)
        if((i=criteriu(q,p))!=0) return q;
        else
            if(i<0)
                q=q->st; /* se cauta in subarborele stang */
            else
                q=q->dr; /* se cauta in subarborele drept */
    return 0; /* nu s-a gasit in arbore un nod echivalent cu cel spre care pointeaza p */
}

```

12.3. Parcurgerea unui arbore binar

Prelucrarea informației păstrată în nodurile unui arbore binar se realizează parcurgând nodurile arborelui respectiv. Parcugerea nodurilor unui arbore binar se poate face în mai multe moduri. Mai jos, indicăm trei moduri de parcugere a arborilor binari:

- in preordine;
- in inordine;
- in postordine.

Parcugerea in preordine înseamnă accesul la radacina și apoi parcugerea celor doi subarborei ai ei, întii subarborele stang, apoi cel drept. Subarboreii, fiind ei însăși arbori binari, se parcug în același mod.

Parcugerea in inordine înseamnă parcugerea mai întii a subarborelui stang, apoi accesul la radacina și în continuare parcugerea subarborelui drept. Cei doi subarborei se parcug în același mod.

Parcugerea in postordine înseamnă parcugerea mai întii a subarborelui stang, apoi a subarborelui drept și în final accesul la radacina arborelui. Cei doi subarborei se parcug în același mod.

Accesul la un nod permite prelucrarea informației conținute în nodul

respectiv. În acest scop se poate apela o funcție care este dependenta de problema concreta care se rezolva cu ajutorul parcurgerii arborelui.

Numim *prelucrare* această funcție. Ea este de prototip:

```
void prelucrare(TNOD *p);
```

unde:

p - Este pointerul spre nodul a carui informație se prelucrarea.

Un exemplu simplu de funcție *prelucrare* este funcția care afișeaza informația conținuta în nodul spre care pointeaza *p*.

Reluind exemplul de la inceputul capitolului, funcția *prelucrare* poate fi definita ca o funcție care afișeaza componentele *nr* și *f* ale nodului spre care pointeaza parametrul ei:

```
void prelucrare(TNOD *p)
{
    printf("numarul=%d aparitii=%d\n", p->nr, p->f);
}
```

Folosind această funcție, să exemplificăm cele trei moduri de parcurgere a arborilor binari cu ajutorul arborelui construit la inceputul capitolului. Conform celor spuse mai sus, accesul la un nod va însemna apelul funcției *prelucrare*, adică afișarea componentelor *nr* și *f* ale nodului respectiv.

a. Parcurgerea în preordine

1. Se are acces la rădâcina arborelui, deci se afișează:
 numarul=20 aparitii=2
2. Se parurge subarborele stîng în preordine.
 Aceasta conduce la următorii pași:
 - 2.1. Se are acces la rădâcina, adică se afișează:
 numarul=5 aparitii=1
 - 2.2. Se parurge subarborele stîng în preordine.
 Aceasta conduce la următorii pași:
 - 2.2.1. Se are acces la radacina, adică se afișează:
 numarul=4 aparitii=1
 - 2.2.2. Se parurge subarborele stîng.
 Acesta este vid de aceea se trece la pasul următor.
 - 2.2.3. Se parurge subarborele drept.
 Acesta este vid (nodul corespunzător lui 4 este frunză) și de aceea se trece la pasul următor.
 - 2.3. Se parurge în preordine subarborele drept corespunzător lui 5.
 Aceasta conduce la următorii pași:
 - 2.3.1. Se are acces la radacina, adică se afișează:
 numarul=7 aparitii=1
 - 2.3.2. Se parurge subarborele stîng.
 Acesta este vid.

Se trece la pasul următor.

2.3.3. Se parurge subarborele drept.

Acesta este vid.

Se trece la pasul următor.

3. Se parurge în preordine subarborele drept corespunzător lui 20.

Aceasta conduce la următorii pași:

3.1. Se are acces la rădâcina, adică se afișează:
 numarul=30 aparitii=2

3.2. Se parurge subarborele stîng.

Acesta este vid.

Se trece la pasul următor.

3.3. Se parurge în preordine subarborele drept corespunzător lui 30.

Aceasta conduce la pașii:

3.3.1. Se are acces la rădâcina, adică se afișează:
 numarul=40 aparitii=1

3.3.2. Se parurge subarborele stîng.

Acesta este vid.

Se trece la pasul următor.

3.3.3. Se parurge subarborele drept.

Acesta este vid.

Procesul de parcurgere al arborelui se încheie.

Se observă că numerele citite au fost afișate în ordinea:

20, 5, 4, 7, 30, 40.

Introducem următoarele notații:

Rnr - Arborele care are ca rădâcina nodul ce corespunde întregului *nr*.

SRnr - Subarborele stîng al arborelui *Rnr*.

DRnr - Subarborele drept al arborelui *Rnr*.

b. Parcurgere în inordine

1. Se parurge subarborele SR20 în inordine.

Aceasta conduce la următorii pași:

1.1. Se parurge subarborele SR5 în inordine.

Aceasta conduce la următorii pași:

1.1.1. Se parurge subarborele SR4.

Acesta este vid.

Se trece la pasul următor.

1.1.2. Se are acces la R4, adică se afișează:

 numarul=4 aparitii=1

1.1.3. Se parurge subarborele DR4.

Acesta este vid.

Se trece la pasul următor.

- 1.2. Se are acces la R5, adică se afișează:
 numarul=5 aparitii=1

- 1.3. Se parurge subarborele DR5 în inordine.

Aceasta conduce la următorii pași:

- 1.3.1. Se parurge subarborele SR7.

Acesta este vid.

Se trece la pasul următor.

- 1.3.2. Se are acces la R7, adică se afișează:
 numarul=7 aparitii=1

- 1.3.3. Se parurge subarborele DR7.

Acesta este vid.

Se trece la pasul următor.

2. Se are acces la R20, adică se afișează:
 numarul=20 aparitii=2

3. Se parurge subarborele DR20 în inordine.

Aceasta conduce la următorii pași:

- 3.1. Se parurge subarborele SR30.

Acesta este vid.

Se trece la pasul următor;

- 3.2. Se are acces la R30, adică se afișează:
 numarul=30 aparitii=2

- 3.3. Se parurge subarborele DR30 în inordine.

Aceasta conduce la următorii pași:

- 3.3.1. Se parurge subarborele SR40.

Acesta este vid.

Se trece la pasul următor.

- 3.3.2. Se are acces la R40, adică se afișează:
 numarul=40 aparitii=1

- 3.3.3. Se parurge subarborele DR40.

Acesta este vid.

Procesul de parurgere al arborelui se încheie.

Numeralele citite au fost afișate în ordine crescătoare:

4, 5, 7, 20, 30, 40

Acest lucru rezultă imediat din modul în care a fost construit arborele și anume; SRnr se compune din noduri care corespund la numere mai mici decât nr , iar DRnr se compune din noduri care corespund la numere mai mari decât nr . Această proprietate are loc pentru orice nod. Conform definiției parurgerii *inordine*, nodurile se afișează în ordinea:

- nodurile lui SRnr;
- nodul Rnr;
- nodurile lui DRnr.

Acest principiu se aplică în continuare la subarborei SRnr și DRnr și își mai departe. În felul acesta se obține o metodă nouă de *sortare*. Ea este mai eficientă decât metoda bulelor în ceea ce privește numarul pașilor.

Sortarea se reduce la construirea arborelui binar și apoi parurgerea acestuia în inordine. Înversind condițiile de inserare a nodurilor în arbore, se va realiza o sortare în ordine descreșătoare a numerelor citite de la intrare. Cu alte cuvinte, ordinea sortării se definește cu ajutorul funcției *criteriu*.

La ora actuală există o serie de algoritmi de sortare care sunt mai eficiente decât acestea și de aceea nu mai insistăm asupra lui (vezi [3]).

c. Parurgere în postordine

1. Se parurge SR20 în postordine.

Aceasta conduce la următorii pași:

- 1.1. Se parurge SR5 în postordine.

Aceasta conduce la următorii pași:

- 1.1.1. Se parurge SR4.

Acesta este vid.

Se trece la pasul următor.

- 1.1.2. Se parurge DR4.

Acesta este vid.

Se trece la pasul următor.

- 1.1.3. Se are acces la R4, adică se afișează:
 numarul=4 aparitii=1

- 1.2. Se parurge DR5 în postordine.

Aceasta conduce la următorii pași:

- 1.2.1. Se parurge SR7.

Acesta este vid.

- 1.2.2. Se parurge DR7.

Acesta este vid.

- 1.2.3. Se are acces la R7, adică se afișează:
 numarul=7 aparitii=1

- 1.3. Se are acces la R5, adică se afișează:
 numarul=5 aparitii=1

2. Se parurge DR20 în postordine.

Aceasta conduce la următorii pași:

- 2.1. Se parurge SR30.

Acesta este vid.

- 2.2. Se parurge DR30 în postordine.

Aceasta conduce la următorii pași:

- 2.2.1. Se parurge SR40.

Acesta este vid.

- 2.2.2. Se parurge DR40.

Acesta este vid.

2.2.3. Se are acces la R40, adică se afișează:

 numarul=40 aparitii=1

2.3. Se are acces la R30, adică se afișează:

 numarul=30 aparitii=2

3. Se are acces la R20, adică se afișează:

 numarul=20 aparitii=2

Procesul de parcurgere a arborelui se încheie.

Numerele citite s-au afișat în ordinea:

4, 7, 5, 40, 30, 20

În următoarele 3 paragrafe se definesc funcții, pentru parcurgerea arborilor în modurile indicate mai sus. Toate cele 3 moduri de parcurgere a arborilor binari au o natură recursivă. Într-adevăr, în oricare din cele 3 moduri se realizează următoarele activități, în fiecare nod al arborelui:

1. Prelucrare nodul N.

2. Parcurgerea în același mod a subarborelui sting al nodului N.

3. Parcurgerea în același mod a subarborelui drept al nodului N.

Diferența dintre cele 3 moduri constă numai în ordinea în care se realizează activitățile 1-3 de mai sus:

1, 2, 3 - În preordine.

2, 1, 3 - În inordine.

2, 3, 1 - În postordine.

Natura recursivă a acestor moduri de parcurgere decurge din faptul că subarboreii fiecărui nod se parcurg în același mod.

Avind în vedere acest fapt, funcțiile de parcurgere a arborilor binari se definesc cel mai simplu prin funcții recursive.

Fiecare din aceste funcții au ca parametru un pointer spre TNOD:

void fparc(TNOD *p);

Apelul unei astfel de funcții, atribuie lui p adresa de început a zonei de memorie în care se păstrează rădăcina arborelui. Rezultă că funcția fparc poate fi apelată printr-o instrucțiune de forma:

fparc(prad);

12.3.1. Parcurgerea arborilor binari în preordine

Numim *preord* funcția care parcurge în preordine un arbore binar. Ea realizează următoarele:

– Dacă pointerul spre rădăcină nu este nul, atunci se execută pașii de mai jos:

1. Se apeleză funcția *prelucrare* cu pointerul spre rădăcină.

2. Fiul sting devine rădăcină și se reapeleză funcția *preord* cu pointerul

spre noua rădăcină.

În felul acesta se va parcurge în preordine subarborele sting.

3. Fiul drept devine rădăcină și se reapeleză funcția *preord* cu pointerul spre noua rădăcină.

În felul acesta se va parcurge în preordine subarborele drept.

– Dacă pointerul spre rădăcină este nul se revine din funcție.

Acești pași se transcriu imediat în limbajul C, ca mai jos:

```
void preord(TNOD *p) /* parcurge arboarele binar în preordine */
{
    if(p!=0) {
        prelucrare(p); /* prelucrareaza radacina */
        preord(p->st); /* parcurge subarborele sting în preordine */
        preord(p->dr); /* parcurge subarborele drept în preordine */
    }
}
```

12.3.2. Parcurgerea arborilor binari în inordine

Numim *inord* funcția care parcurge în inordine un arbore binar. Ea realizează aceeași pași ca și funcția *preord*, dar în altă ordine. Dacă menținem numerația pașilor definiți în paragraful precedent, atunci funcția *inord* realizează pașii respectivi în ordinea:

2, 1, 3

În felul acesta se obține funcția de mai jos:

```
void inord(TNOD *p) /* parcurge arboarele binar în inordine */
{
    if(p!=0) {
        inord(p->st); /* parcurge subarborele sting în inordine */
        prelucrare(p); /* prelucrareaza radacina */
        inord(p->dr); /* parcurge subarborele drept în inordine */
    }
}
```

12.3.3. Parcurgerea arborilor binari în postordine

Numim *postord* funcția care parcurge în postordine un arbore binar. Ea realizează aceeași pași ca și funcțiile *preord* și *inord*, dar în altă ordine și anume:

2, 3, 1

unde prin 1, 2, 3, am notat pașii definiți în paragraful 12.3.

Se obține funcția de mai jos:

```
void postord(TNOD *p) /* parcurge arboarele binar în postordine */
{
    if(p!=0) {
        postord(p->st); /* parcurge subarborele sting în postordine */
    }
}
```

```

postord(p -> dr); /* parcurge subarborele drept in postordine */
prelucrare(p); /* prelucrarea radacina */
}
}

```

12.4. Ștergerea unui arbore binar

Pentru a șterge un arbore binar este necesară parcurgerea lui și ștergerea fiecărui nod al arborelui respectiv.

Ștergerea unui nod se realizează apelând funcția *elibnod*. Arborele se parcurge în *postordine*. Rezultă că funcția care șterge un arbore binar este asemănătoare cu funcția *postord* definită în paragraful 12.3.3. Deosebirea constă în aceea că, pentru a prelucra rădăcina se apelează funcția *elibnod* în locul funcției *prelucrare*.

```

void stergarb(TNOD *p) /* sterge arborele spre a carui radacina pointeaza p */
{
    if(p!=0){
        stergarb(p -> st);
        stergarb(p -> dr);
        elibnod(p);
    }
}

```

Mentionăm că funcția *stergarb* nu atribuie valoarea zero variabilei globale *prad*. Aceasta este necesar să se realizeze în funcția care apelează funcția *stergarb*.

Exerciții:

12.1 Se consideră tipul TNOD definit ca mai jos:

```

typedef struct tnod {
    int nr;
    int f;
    struct tnod *st;
    struct tnod *dr;
} TNOD;

```

Să se scrie funcția *incnod* care are antetul:

```
int incnod(TNOD *p);
```

și care realizează următoarele:

- citește un întreg de tip *int* și-l atribuie variabilei *p->nr*;
- atribuie valoarea 1 variabilei *p->f*;
- returnează:
 - 1 - La întîlnirea sfîrșitului de fișier.
 - 1 - Altfel.

Funcția *incnod* apelează funcția *pcit_int_lim* definită în exercițiul 8.3.

FUNCȚIA BXII1

```

int incnod(TNOD *p) /* incarcă nodul spre care pointeaza p */
{
    int n;

    if(pcit_int_lim ("nr= ", -32768, 32767,&n) == 0 )
        return -1; /* s-a intîlnit EOF */
    p -> nr = n;
    p -> f = 1;
    return 1;
}

```

12.2 Să se scrie funcția *elibnod* care eliberează zona de memorie ocupată de un nod de tipul TNOD definit în exercițiul 12.1.

FUNCȚIA BXII2

```

void elibnod(TNOD *p) /* eliberează zona de memorie ocupată
                        de nodul spre care pointeaza p */
{
    free(p);
}

```

12.3 Să se scrie funcția *echivalenta* de prototip:

*TNOD *echivalenta (TNOD *q,TNOD *p);*

care realizează următoarele:

- eliberează zona de memorie ocupată de nodul spre care pointeaza *p*;
- incrementează valoarea variabilei *q->f*;
- (TNOD este tipul definit în exercițiul 12.1.);
- returnează valoarea pointerului *q*.

FUNCȚIA BXII3

```

TNOD *echivalenta (TNOD *q,TNOD *p)
/* - eliberează zona de memorie ocupată de nodul spre care pointeaza p,
   - incrementează valoarea variabilei q->f;
   - returnează valoarea lui q. */
{
    elibnod(p);
    q -> f++;
    return q;
}

```

12.4 Să se scrie funcția *prelucrare* care afișează valorile componentelor *nr* și *f* dintr-o dată de tip TNOD spre care pointează parametrul ei.

FUNCȚIA BXII4

```

void prelucrare(TNOD *p) /* afiseaza pe p->nr si p->f */
{
    printf("numarul=%d apare %i ori\n",p->nr, p->f);
}

```

- 12.5 Să se scrie funcția *criteriu* definită la începutul acestui capitol pentru datele de tipul TNOD.

FUNCȚIA BXII5

```
int criteriu(TNOD *p1, TNOD *p2)
/* returnaza:
   -1 - daca p2->nr < p1->nr;
    1 - daca p2->nr > p1->nr;
    0 - altfel. */
{
    if(p2->nr < p1->nr)
        /* nodul spre care pointeaza p2 poate fi inserat în subarborele stîng al nodului spre care
           pointeaza p1 dacă subarborele respectiv nu conține un nod echivalent cu cel spre care
           pointează p2 */
        return -1;

    if(p2->nr > p1->nr)
        /* nodul spre care pointeaza p2 poate fi inserat în subarborele drept al nodului spre care
           pointeaza p1 dacă subarborele respectiv nu conține un nod echivalent cu cel spre care
           pointează p2 */
        return 1;

    /* nodurile spre care pointeaza p1 și p2 sunt echivalente */
    return 0;
}
```

- 12.6 Să se scrie o funcție care inserează un nod într-un arbore binar.

Nodurile arborelui au tipul TNOD.

Această funcție nu depinde de structura nodurilor arborelui. Ea a fost definită în paragraful 12.1.

FUNCȚIA BXII6

```
TNOD *insnod()
/* inseră un nod în arborele binar spre același radacina pointeaza prad;
   - returnaza pointerul spre nodul inserat sau pointerul returnat de funcția echivalenta dacă
     nodul de inserat este echivalent cu unul deja aflat în arbore;
   - returnaza zero dacă nu mai sunt date de incarcat sau la eroare. */
{
    extern TNOD *prad;
    int i;
    int n;
    TNOD *p, *q;

    n = sizeof(TNOD);
    if(((p=(TNOD *)malloc(n))!=0) && (incnod(p)==1)) ( /* 1 */
        p->st = p->dr = 0;
        if(prad == 0) ( /* arbore vid */
            prad = p;
            return p;
        )
    )
}
```

```
q = prad;
for ( ; ; ) {
    if((i = criteriu(q,p)) < 0)
        if(q->st == 0) {
            q->st = p;
            return p;
        }
    else {
        q = q->st;
        continue;
    }
    if( i > 0 )
        if(q->dr == 0) {
            q->dr = p;
            return p;
        }
    else {
        q = q->dr;
        continue;
    }
}
return echivalenta(q,p);
} /* sfîrșit for */
} /* sfîrșit if 1 */
if( p == 0 ) {
    printf("memorie insuficientă\n");
    exit(1);
}
elibnod (p);
return 0;
}
```

- 12.7 Să se scrie o funcție care parcurge un arbore binar în preordine.

Această funcție nu depinde de tipul nodurilor arborelui binar și ea a fost definită în paragraful 12.3.1.

FUNCȚIA BXII7

```
void preord(TNOD *p) /* parcurge arborele binar în preordine */
{
    if(p != 0) {
        prelucrare(p);
        preord( p->st );
        preord( p->dr );
    }
}
```

- 12.8 Să se scrie o funcție care parcurge arborele binar în inordine.

Această funcție nu depinde de structura nodurilor arborelui și ea a fost definită în paragraful 12.3.2.

FUNCȚIA BXII8

```
void inord(TNOD *p) /* parurge arborele binar in inordine */
{
    if( p != 0 ) {
        inord( p -> st );
        prelucrare(p);
        inord( p -> dr );
    }
}
```

12.9 Să se scrie o funcție care parurge un arbore binar in postordine.

Această funcție nu depinde de structura nodurilor arborelui și ea a fost definită în paragraful 12.3.3.

FUNCȚIA BXII9

```
void postord(TNOD *p) /* parurge arborele binar in postordine */
{
    if( p != 0 ) {
        postord( p -> st );
        postord( p -> dr );
        prelucrare(p);
    }
}
```

12.10 Să se scrie un program care citește un sir de întregi de tip *int* și afișează frecvența de apariție a fiecărui număr citit. Numerele sunt separate prin caractere albe, iar la sfîrșit se tastează sfîrșitul de fișier.

Rezolvăm această problemă cu ajutorul arborilor binari.

La început se citesc numerele de la intrare și acestea se păstrează în nodurile unui arbore binar de tipul TNOD definit în exercițiul 12.1.

Componenta *nr* are ca valoare numărul citit. La inserarea unui nod în arbore, f=1, deoarece nodul neexistă în prealabil în arbore, înseamnă că nici numărul pe care-l conține nodul respectiv nu a mai fost citit înainte.

Dacă la citirea unui număr se constată că acestuia îi corespunde deja un nod în arbore, atunci se incrementează valoarea lui *f* pentru nodul respectiv aflat în arbore.

După construirea arborelui binar, acesta se va parurge în cele 3 moduri definite în paragraful 12.3., afișindu-se frecvența întregilor citiți.

Se va observa că parurgerea în inordine afișează numerele citite în ordine crescătoare.

PROGRAMUL BXII10

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

typedef struct tnod {
    int nr;
```

```
    int f;
    struct tnod *st;
    struct tnod *dr;
} TNOD;

#include "bviii2.cpp" /* pcit_int */
#include "bviii3.cpp" /*pcit_int_lim */
#include "bxii1.cpp" /* incnod */
#include "bxii2.cpp" /* elibnod */
#include "bxii3.cpp" /* echivalenta */
#include "bxii4.cpp" /* prelucrare */
#include "bxii5.cpp" /* criteriu */
#include "bxii6.cpp" /* insnod */
#include "bxii7.cpp" /* preord */
#include "bxii8.cpp" /* inord */
#include "bxii9.cpp" /* postord */
```

```
TNOD *prad;

main()
/* citește un sir de numere și le păstrează în nodurile unui arbore binar, apoi afișează frecvența de apariție a fiecărui număr citit, parcurgind arborele în preordine, inordine și postordine */
{
    /* construiește arborele binar */
    prad = 0;
    while( insnod() )
        ;

    /* parurge arborele în preordine */
    printf("\n\n\tpreordine\n\n");
    preord(prad);

    /* parurge arborele în inordine */
    printf("\n\n\tinordine\n\n");
    inord(prad);

    /* parurge arborele în postordine */
    printf("\n\n\tpostordine\n\n");
    postord(prad);
}
```

12.11 Se consideră tipul TNOD ca mai jos:

```
typedef struct tnod {
    char *cuvant;
    int frecventa;
    struct tnod *st;
    struct tnod *dr;
} TNOD
```

Să se scrie funcția *echivalenta* pentru nodurile de tipul TNOD definit mai sus, care realizează următoarele:

Fie antetul:

TNOD *echivalenta(TNOD *q,TNOD *p)

- funcția elibereaza zona de memorie ocupata de nodul spre care pointează p;
- incrementeaza valoarea componentei: q->frecventa;
- returneaza valoarea lui q.

Acastă funcție este similară cu funcția definită în exercițiul 12.3.

FUNCȚIA BXII11

```
TNOD *echivalenta (TNOD *q, TNOD *p)
/* - elibereaza zona de memorie ocupata de nodul spre care pointeaza p;
   - incrementeaza pe q->frecventa;
   - returneaza valoarea lui q. */
{
    elibnod (p);
    q->frecventa++;
    return q;
}
```

Observație:

Tipul TNOD definit mai sus este similar cu tipul TNOD definit în exercițiul 11.1. Aceste două tipuri difera numai prin componentele de înlățuire.

În cazul de față se folosesc pointerii *st* și *dr*, iar în exercițiul 11.1 se utilizează pointerul *urm*.

Celealte componente (*cuvant* și *frecventa*) sunt identice pentru cele două tipuri de noduri. De aceea, funcția *elibnod* definită în exercițiul 11.2, poate fi utilizată și pentru nodurile de tipul TNOD definit mai sus.

12.12 Să se scrie funcția *prelucrare* care afișează datele care nu sunt de înlățuire diatr-un nod de tipul TNOD definit în exercițiul 12.11.

FUNCȚIA BXII12

```
void prelucrare(TNOD *p) /* afiseaza p->cuvant si p->frecventa */
{
    static int n = 0;

    printf("cuvintul: %s are frecventa: %d\n",
           p->cuvant, p->frecventa);
    if((n+1)%23 == 0) {
        printf("pentru a continua actionati o tasta\n");
        getch();
    }
    n++;
}
```

12.13 Să se scrie funcția *criteriu* de antet:

```
int criteriu(TNOD *p1,TNOD *p2)
```

și care returnează valorile:

- 1 - dacă p2->cuvant < p1->cuvant;
- 1 - dacă p2->cuvant > p1->cuvant;
- 0 - dacă p2->cuvant = p1->cuvant.

unde:

TNOD - Este tipul definit în exercițiul 12.11.

FUNCȚIA BXII13

```
int criteriu(TNOD *p1, TNOD *p2)
/* returneaza:
   -1 - daca p2->cuvant < p1->cuvant;
   1 - daca p2->cuvant > p1->cuvant;
   0 - altfel. */
{
    int i;

    if((i = strcmp(p2->cuvant,p1->cuvant)) < 0) return -1;
    else
        if(i > 0) return 1;
        else return 0;
}
```

12.14 Sa se scrie un program care citește cuvintele dintr-un text și afișează numărul de apariții al fiecărui cuvint din textul respectiv. Cuvintul se definește ca o succesiune de litere mici și/sau mari. Textul se termină prin sfîrșitul de fișier.

Această problemă a fost rezolvată în capitolul 11, folosind listele simplu înlățuite și dublu înlățuite. Programul de față rezolvă această problemă folosind arborii binari.

Rezolvarea cu ajutorul arborilor este similară cu cea folosită în exercițiul 12.10. Diferența constă în accea că în cazul exercițiului 12.10 se citesc numere de tip *int*, iar în cazul de față se citesc cuvinte.

Programul de față, ca și cel din exercițiul 12.10, apelează o serie de funcții care sunt definite în diverse exerciții. Astfel, funcția *elibnod* este cea definită în exercițiul 11.2. (vezi observația de la exercițiul 12.11.). Funcția *incnod*, citește un cuvint și-l păstrează în memoria *heap*, apoi atrbuie pointerului *cuvant* adresa zonei de început a memoriei în care se păstrează cuvintul respectiv, iar la componenta *frecventa* î se atrbuie valoarea 1. Funcția returnează valoarea -1 la înțlnirea sfîrșitului de fișier și 1 în caz contrar. Această funcție este definită în exercițiul 11.1. Ea apelează funcția *citcuv* definită în exercițiul 10.48.

Celealte funcții apelate de program sunt definite în capitolul de față.

PROGRAMUL BXII14

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>

typedef struct tnod {
    char *cuvant;
    int frecventa;
    struct tnod *st;
    struct tnod *dr;
} TNOD;

#include "bx48.cpp" /* citeuv */
#include "bx11.cpp" /* incnod */
#include "bxi2.cpp" /* elibnod */
#include "bxii11.cpp" /* echivalenta */
#include "bxii12.cpp" /* prlucrare */
#include "bxii13.cpp" /* criteriu */
#include "bxii6.cpp" /* insnod */
#include "bxii8.cpp" /* inord */

TNOD *prad;

main()
/* citeste cuvintele dintr-un text si le afisaza in ordine alfabetica
impreuna cu freeventa de aparitie a lor in text */
{
    /* construieste arborele binar */
    prad = 0;
    while(insnod())
        ;

    /* parcurge arborele in inordine */
    inord(prad);
}
```

Observație:

Problema de față se rezolvă mai eficient folosind arborii binari în locul listelor. Aceasta din cauză că operația de căutare a unui cuvint într-o listă este ineficientă. Într-adevăr, căutarea unui cuvint într-o listă simplu înlanțuită presupune parcurgerea secvențială a nodurilor ei fie pînă la întlnirea nodului ce conține cuvîntul căutat, fie pînă la sfîrșitul listei în cazul în care cuvîntul nu este conținut nici într-un nod al listei.

Folosind arborii, procesul de căutare al unui cuvînt în arbore necesită, de obicei, mai puțini pași. În fiecare nod se realizează o ramificare, deoarece se continuă căutarea fie în subarborele stîng, fie în cel drept, fie se constată că nodul respectiv conține chiar cuvîntul căutat.

În felul acesta se observă că la căutarea unui cuvînt o parte din nodurile

arborelui rămîn neparcuse.

Utilizarea arborilor devine ineficientă dacă la citirea cuvîntelor se obține un arbore *degenerat*. Un arbore binar este degenerat, dacă subarborei lui sunt toți de același fel, de exemplu, toți sunt subarbore stîngi sau toți sunt subarbore drepti. În acest caz, arborele binar este echivalent cu o listă simplu înlanțuită.

În problema de față se obțin arbori degenerați dacă, de exemplu, se citesc cuvîntele în ordine alfabetică. Aceasta este puțin probabil cînd se citesc cuvînte dintr-un text normal.

13. TABELE

Noțiunea de *tabel* o definim ca o *colecție* de elemente identificabile prin *chei* [3].

Elementele colecției sunt date de *același tip*, care în mod frecvent este un tip utilizator.

Prin *cheie* înțelegem, ca și în cazul listelor, o componentă a elementelor tabeliei care are valori *diferite* pentru elemente diferite. În felul acesta, un element din tabel se identifică exact prin valoarea cheii sale.

Elementele unei tabele se mai numesc și *inregistrări*.

Ca exemple de tabele întâlnite frecvent în diferite aplicații amintim: tabele de cuvinte rezervate, tabele de simboluri, tabele de indecsă pentru fișiere etc.

O problema importantă care se pune în legătură cu tabelele este aceea a căutării unei inregistrări de cheie dată.

Tabelele de cuvinte rezervate sunt exemple de tabele *fixe* cu un număr de inregistrări cunoscut dinainte.

Astfel de tabele se întâlnesc în compilatoare și se utilizează pentru a determina dacă un identificator este sau nu un cuvînt rezervat. În acest caz, identificatorul citit din textul de intrare se caută în tabela de cuvînte rezervate pentru a stabili dacă acesta este un cuvînt rezervat sau definit de utilizator. De obicei, procedura de căutare returnează un cod dacă identificatorul cautat s-a aflat în tabela de cuvînte rezervate sau o valoare distinctă de orice cod (de exemplu -1) în cazul în care identificatorul respectiv nu este un cuvînt rezervat.

Tabelele de acest tip se recomandă să fie *ordonate*, deoarece în felul acesta putem aplica metode de căutare eficiente, ca de exemplu *căutarea binară*.

Tabelele de simboluri sunt exemple de tabele *dinamice*, care se construiesc pe măsură ce se constată că inregistrarea curentă nu se află în tabelă. Astfel de tabele necesită alte metode de căutare și care în mod frecvent sunt urmărite de inserarea în tabelă a inregistrării existente.

O tabelă dinamică poate să fie organizată ca o listă simplu sau dublu înlănțuită, dar în acest caz eficiența procedurii de căutare este foarte mică.

O altă posibilitate este de a organiza tabela inserind fiecare inregistrare în nodul unui arbore binar. În acest caz, este nevoie de un *criteriu* pe care să-l satisfacă cheile inregistrărilor tabelei respective. Acest criteriu se utilizează la localizarea în arbore a nodului corespunzător inregistrării curente.

În cazul în care nu există un nod corespunzător inregistrării respective, se inseră în arbore un astfel de nod. În felul acesta, în general, se va mări mult eficiența procedurilor de căutare și inserare a inregistrărilor în arbore. Criteriul care se află la baza căutării într-un arbore binar, de obicei, reduce substanțial numărul nodurilor consultate. Astfel, în fiecare nod are loc una din următoarele 4 posibilități:

- a. Nodul corespunde inregistrării curente. Procesul de căutare se încheie, identificându-se nodul căutat.
- b. Căutarea se continuă în subarborele stîng al nodului curent.
- c. Căutarea se continuă în subarborele drept al nodului curent.
- d. Căutarea se întrerupe deoarece nodul curent este un nod frunză (nu există subarbore pentru nodul respectiv).

În acest caz, inregistrării curente nu-i corespunde nici un nod în arbore; inregistrarea curentă, de obicei, se inseră ca fiu stîng sau drept al nodului la care s-a ajuns.

Se observă că în fiecare nod, procesul de căutare poate continua numai în subarborele stîng sau numai în cel drept al nodului respectiv. În felul acesta, la fiecare pas se exclud de la căutare nodurile unui subarbore.

Amintim că se poate ajunge la o eficiență slabă, comparabilă cu cea obținută la utilizarea listelor, dacă arborele degeneră într-o listă (fiecare nod care nu este o frunză are tot timpul numai descendental stîng sau tot timpul numai descendental drept).

Degenerarea arborelui se obține atunci cind inregistrările se inseră în arbore într-o astfel de ordine încit cheile lor satisfac tot timpul criteriul (sau negația acestuia) utilizat la căutarea în arbore. De obicei, această situație are o probabilitate mică de apariție.

O a treia soluție pentru organizarea tabelelor dinamice o constituie să numitele *tabele de dispersie*. În [3] se descrie, în detaliu, utilizarea tabelelor de acest tip.

În capitolul de față vom prezenta o variantă simplificată de utilizare a tabelelor de dispersie la construirea *tabelelor de simboluri*.

13.1. Tabela de cuvînte rezervate

O tabelă de cuvînte rezervate este o tabelă ordonată de dimensiune fixă, dinainte cunoscută. Pentru a fixa ideile să presupunem că, cuvîntele rezervate sunt cuvînte imprumutate din limba engleză și sunt utilizate într-un limbaj de programare, avind fiecare un înțeles predefinit.

În acest caz cheile, precum și inregistrările, se reduc fiecare la cuvîntele rezervate. Ordinea impusă cheilor va fi cea *alfabetică*.

Tabela poate să fie realizată simplu în limbajul C și anume vom folosi un tablou de pointeri spre caractere și fiecare element al acestuia se inițializează cu adresa de început a zonei de memorie în care se păstrează un cuvînt rezervat. De exemplu, dacă avem în vedere cuvîntele rezervate din limbajul C, atunci tabela de cuvînte rezervate se poate declara ca mai jos:

```
char *tcr[] = {  
    "break",  
    "case",  
    ...
```

```

    "while"
};

```

unde cuvintele rezervate au fost scrise în ordine alfabetică.

Procedura de căutare intr-o astfel de tabelă poate să căutarea binară care a fost utilizată pentru un tablou cu elemente numerice în exercițiul 4.42.

În cazul de față se schimbă numai modul de realizare al comparațiilor dintre elementele tabelei și elementul curent care se căută în tabelă. În exercițiul amintit mai sus, comparația se realizează folosind operatorii relaționali obișnuiți ($==$ și $>$) deoarece se compară numere. În cazul de față, pentru a compara două cuvinte rezervate, se utilizează funcția de bibliotecă `strcmp`. Funcția de căutare returneză:

- indicele cuvintului rezervat dacă acesta a fost găsit în tabelă;
- -1 în caz contrar.

Cu aceste precizări, funcția de căutare în tabela de cuvinte rezervate se poate defini ca mai jos:

```

int cautrez(char *pcrt)
/*- cauta în tabela de cuvinte rezervate cuvintul spre care pointeaza pcrt;
 - returnaza indicele cuvintului în tabela sau -1 daca nu exista.*/
{
    int inf,sup,i,j;

    static char *tcr[]={
        "break",
        "case",
        ...
        "while"
    };

    inf=0;
    sup=sizeof(tcr)/sizeof(char *)-1;
    for(i=(inf+sup)/2;inf <= sup;i=(inf+sup)/2)
        if((j=strcmp(pcrt,tcr[i]))==0)
            /* cuvintul cautat s-a gasit în tabela */
            return i;
        else
            if(j > 0)
                /* cuvintul care se cauta este după cel spre care pointeaza tcr[i],
                   deci limita inferioara poate fi marită la i+1 */
                inf=i+1;
            else
                /* cuvintul care se cauta este înaintea celui spre care pointeaza tcr[i],
                   deci limita superioara poate fi micsorată la i-1 */
                sup=i-1;

    /* cuvintul cautat nu a fost gasit în tabela */
    return -1;
}

```

13.2. Tabela de dispersie

Tabelele de dispersie au la bază o funcție de transformare a cheilor înregistrărilor:

$hf: K \rightarrow H$

unde:

- | | |
|-----|--------------------------------------|
| K | - Este mulțimea cheilor; |
| H | - Este o mulțime de numere naturale. |

În general, funcția hf nu este injectivă, existând două sau mai multe chei pentru care hf are aceeași valoare.

Funcția hf se numește *funcție de dispersie* (în engleză hashing), iar utilizarea ei conduce la o metodă numită *tehnica de dispersie* (technică de hashing).

Prima problemă care se pune în legătură cu tabela de dispersie este aceea de a alege funcția de dispersie.

În primul rînd, se poate presupune că $0 \leq hf(k) \leq M$, pentru orice cheie k .

Aceasta se obține dacă:

$hf(k) = f(k) \text{ mod } M$,

unde:

- | | |
|--------|---|
| $f(k)$ | - Transformă cheia k într-un număr natural. |
|--------|---|

Relația de mai sus definește pe $hf(k)$, egal cu restul împărțirii întregi a lui $f(k)$ prin M . Pentru a obține o repartizare mai uniformă a valorilor funcției hf , numărul M se alege să fie prim (vezi [3]).

Despre două chei ki și kj diferite, se spune că intră în *coliziune*, dacă $hf(ki)=hf(kj)$.

O funcție de dispersie trebuie să satisfacă urmatoarele condiții (vezi [3]):

- valoarea ei să se calculeze simplu și rapid;
- să minimizeze numărul coliziunilor.

Numărul coliziunilor este dependent de cheile înregistrărilor, precum și de funcția hf . Cu cît aceasta realizează o repartizare mai uniformă a valorilor cheilor, cu atât numărul coliziunilor va fi mai mic.

Numărul coliziunilor poate crește dacă numărul M este prea mic. Pe de altă parte, creșterea exagerată a lui M conduce la un consum mare de memorie. De aceea, M trebuie ales printre un compromis între necesarul de memorie și timpul de căutare suplimentar cheltuit pentru rezolvarea coliziunilor.

Funcția $f(k)$ se alege în funcție de natura cheilor. Dacă k este o cheie numerică, atunci se poate alege $f(k)=k$.

În cazul în care cheile nu sunt numerice, atunci există mai multe posibilități de a transforma cheile în numere naturale.

Așa de exemplu, se poate considera o codificare numerică a caracterelor unei astfel de chei și prin aceasta fiecare cheie se transformă într-un sir de coduri

binare, iar acesta din urma poate fi interpretat ca reprezentind un intreg binar fără semn. O astfel de metoda, deși pare simplă, poate conduce la calcule greoale cu numere mari. De aceea se adoptă metode mai simple. Așa de exemplu, se poate proceda la a păstra numai primele 2-3 coduri binare din reprezentările cheilor sau numai ultimele 2-3 coduri sau un număr mic de coduri din mijlocul cheilor.

O metodă simplă, aplicată frecvent, este aceea de a însuma codurile caracterelor din compunerea unei chei. Mai jos, vom utiliza și noi acest procedeu simplu pentru a transforma cheile în numere naturale.

Exemplu:

Fie $M = 127$. Presupunem că dorim să construim o tabelă de simboluri. În acest caz cheile înregistrărilor sunt identificatori (succesiuni de litere și eventual și cifre care, de obicei, încep cu o literă). Vom presupune că se utilizează codul ASCII pentru reprezentarea caracterelor. Fie identificatorul AB1. Atunci

$$f(AB1) = 'A' + 'B' + '1' = 65 + 66 + 49 = 180$$

$$hf(AB1) = 180 \% 127 = 53.$$

Evident, identificatorii AB1, BA1, B1A au aceeași dispersie, deoarece:

$$f(AB1) = f(AB1) = f(BA1) = f(B1A) = 180$$

Deci, toți identificatorii de mai sus intră în coliziune.

Problemele implicate de coliziune pot fi rezolvate în mai multe moduri (vezi [3]).

În cartea de față adoptăm o soluție simplă și anume accea de a rezolva coliziunile cu ajutorul listelor simplu înlățuite. Astfel, toate înregistrările pentru care cheile au aceeași dispersie se inserează într-o listă simplu înlățuită.

Vom avea deci, mai multe liste simplu înlățuite, fiecare listă conținând înregistrările ale căror chei au aceeași dispersie.

Pentru a putea gestiona înregistrările din aceste liste este necesar să cunoaștem pointerul spre primul nod al fiecărei astfel de liste.

Numarul maxim al acestor liste este M , deoarece funcția hf are M valori: $0, 1, \dots, M-1$. Pointerii spre incepurile listelor respective se păstrează într-un tablou de M elemente. Numim $thash$ acest tablou. Atunci $thash[i]$ are ca valoare pointerul spre incepul listei ce conține înregistrările pentru care cheile au dispersia egală cu i .

Reluind exemplul de mai sus, $thash[53]$ are ca valoare pointerul spre incepul listei care conține înregistrările cu cheile: AB1, BA1 și B1A.

Initial, aceste liste nu există și în consecință elementele $thash[i]$ pentru $i=0, 1, 2, \dots, M-1$ au valoarea zero.

În capitolul 11 s-au utilizat variabilele globale *prim* și *ultim* pentru a defini capetele unei liste simplu înlățuite. În cazul de față un element al tabloului *thash* joacă același rol ca și variabila *prim*. Date fiind operațiile avute în vedere asupra listelor care conțin înregistrările aflate în coliziune, se constată că nu avem nevoie de variabila *ultim*.

Procedura de căutare într-o tabelă de dispersie se realizează în următorii pași:

1. Dacă $thash[h] = 0$, unde h este dispersia cheii curente, atunci înregistrarea respectivă nu există în tabelă și funcția de căutare returnează valoarea zero.
2. Dacă $thash[h]$ este diferit de zero, atunci acesta este pointerul spre incepul listei în care se află înregistrările pentru care cheile au dispersia egală cu h . Căutarea înregistrării curente se face în această listă începând cu nodul spre care pointează $thash[h]$.

Această căutare se realizează secvențial, nod după nod, pînă cînd se găsește o înregistrare cu aceeași cheie ca și cea curentă sau pînă cînd se ajunge la ultimul nod al listei, lucru care se întimplă cînd înregistrarea nu este prezentă în lista respectivă.

În primul caz funcția de căutare returnează pointerul spre înregistrarea găsită, iar în al doilea caz returnează valoarea zero.

Din cele de mai sus rezultă că procedura de căutare este cu atît mai eficientă, cu cît listele cu înregistrările aflate în coliziune sunt mai mici. Aceasta este funcție de constanta M și de funcțiile hf și f definite mai sus.

De obicei, în cazul tabelelor de simboluri utilizate în compilatoare, M se alege sub 1000. De asemenea, insumarea codurilor caracterelor identificatorilor să-a dovedit a fi o soluție simplă și eficientă pentru calculul dispersiei. În acest caz funcția hf de calcul a dispersiei se definește ca mai jos:

```
#define M 127
unsigned hf(char *s)
{
    unsigned h;
    for(h = 0; *s; ) h += *s++;
    return h%M;
}
```

În continuare, definim funcția de căutare în tabelă de dispersie, presupunind că aceste chei sunt siruri de caractere.

Numim *hcaut* această funcție. Ea returnează pointerul spre înregistrarea căutată sau zero dacă înregistrarea respectivă nu există în tabelă.

O înregistrare are tipul TNOD definit ca în cazul listelor simplu înlățuite:

```
typedef struct tnod {
    declarati;
    char *cheie;
    declarati;
    struct tnod *urm;
} TNOD;
```

Tabloul *thash* are M elemente și elementele lui sunt pointeri spre tipul TNOD. Vom presupune că acest tablou este global.

Initial elementele tabloului *thash* au valoarea zero (toate listele sunt vide). Acest tablou se definește astfel:

```
TNOD *thash[M];
```

Funcția *hcaut* are ca parametri pointerul spre sirul de caractere din compunerea cheii (se caută înregistrarea a cărei cheie coincide cu acest sir de caractere) și valoarea dispersiei acestui sir.

Cu aceste precizări, funcția de căutare se definește astfel:

```
TNOD *hcaut(char *s,unsigned h)
/* - cauta înregistrarea pentru care cheie pointeaza spre un sir de
   caractere identic cu cel spre care pointeaza s;
   - returneaza pointerul spre înregistrarea respectiva sau zero daca
   nu exista o astfel de înregistrare. */
{
    TNOD *p;

    /* cauta înregistrarea in lista spre care pointeaza thash[h] */
    for(p = thash[h]; p; p = p->urm)
        if(strcmp(s,p->cheie)==0)
            /* s-a gasit înregistrarea cautata */
            return p;

    /* nu exista o înregistrare pentru care cheie sa pointeze spre un sir
       identic cu cel spre care pointeaza s */
    return 0;
}
```

Operația de introducere a unei înregistrări într-o tabelă de simboluri se va realiza printr-o funcție pe care o numim *intab*. Ca și în cazul listelor, funcția *intab* va utiliza funcțiile *incnod* și *elibnod*, prima pentru a încărca datele înregistrării în zona de memorie alocată prin funcția *malloc*, iar a doua pentru a elibera o astfel de zonă.

Deoarece tabloul *thash* conține pointeri spre inceputul listelor și nu spre sfîrșitul lor, va fi util ca operația de inserare a nodului nou în listă să se facă înaintea primului ei nod, adică folosind o funcție analogă cu funcția *iniprim* definită în paragraful 11.1.3.1.

Astfel, dacă *p* pointează spre nodul care se inserează în listă, iar *h* este dispersia cheii lui, atunci știm că *thash[h]* pointează spre primul nod al listei în care urmează să se insereze înregistrarea spre care pointează *p*. Cum înregistrarea se inserează înaintea primului nod al listei, va trebui ca după inserare *p->urm* să pointeze spre acest prim nod, deci se face atribuirea:

```
p->urm = thash[h].
```

Totodată, înregistrarea care se inserează devine primul nod al aceleiasi liste, deci în continuare va trebui ca:

```
thash[h] = p.
```

În felul acesta, operația de inserare a înregistrării curente se realizează conform pașilor de mai jos:

1. Rezervă zona pentru înregistrarea curentă folosind funcția *malloc*.
2. Încarcă datele în zona rezervată la punctul 1.
3. Caută înregistrarea curentă în tabelă.
 - Dacă există, atunci se revine din funcție returnându-se pointerul spre înregistrarea găsită.
 - Altfel se inserează înregistrarea curentă în tabelă și se returnează pointerul spre ea.

Funcția returneză valoarea zero dacă nu mai există înregistrări sau s-a depistat o eroare la încărcarea datelor în înregistrarea curentă.

Definim mai jos funcția *intab*.

```
TNOD *intab()
/* - determină înregistrarea curentă și o inserează dacă nu a fost găsită în tabela;
   - returnează pointerul spre înregistrarea respectivă sau NULL în
   cazul cînd nu mai există înregistrări sau la creare. */
TNOD *p,*q;
unsigned h;

/* rezerva zona și încarcă datele înregistrării curente */
if(((p=(TNOD *)malloc(sizeof(TNOD)))!=0) && (incnod(p)==1)) {
    /* calculează dispersia cheii */
    h = hf(p->cheie);

    /* - dacă thash[h]==0, atunci există în tabela înregistrări a căror chei au dispersia h;
       - în acest caz se cauta înregistrarea în tabela */
    if(thash[h])
        if((q=hcaut(p->cheie,h))!=0)
            /* înregistrarea există în tabela */
            return echivalenta(q,p);
        else{
            /* - înregistrarea nu există în tabela;
               - se inserează înaintea nodului spre care pointează thash[h]. */
            p->urm = thash[h];
            thash[h] = p;
            return p;
        }
    else{
        /* nu există înregistrări care să intre în coliziune cu înregistrarea curentă */
        thash[h] = p;
        p->urm = 0;
        return p;
    }
}
if(p==0){
    printf("memorie insuficienta\n");
    exit(1);
}

/* creare la încărcarea datelor sau nu mai sunt înregistrări */
elibnod(p);
```

```

    return 0;
}

```

Funcția *echivalenta* se apelează în cazul în care în tabela există o înregistrare de același cheie cu cea a înregistrării curente. În general, aceasta funcție are ca efect modificarea sau actualizarea datelor înregistrării din tabela cu cele din înregistrarea curentă.

Ea a fost utilizată și în cazul inserării nodurilor în arbori binari (vezi capitolul 12).

În cazul în care înregistrarea curentă nu conține date pentru modificarea înregistrării din tabela care-i corespunde, funcția *echivalenta* se reduce la stergerea înregistrării curente, ca mai jos:

```

TNOD *echivalenta (TNOD *q, TNOD *p)
/* sterge înregistrarea spre care pointeaza p și returneaza valoarea lui q */
{
    elibnod(p);
    return q;
}

```

Observații:

1. Sistemul format din liste care conțin înregistrările aflate în coliziune și din tabeloul de pointeri spre primele înregistrări ale acestor liste, constituie o *tabelă de dispersie*. La baza inserării și accesului la o înregistrare din tabelă de dispersie se află funcția *hf* de calcul a dispersiei.
2. Tabelă de dispersie poate fi organizată înlocuind liste cu înregistrările aflate în coliziune prin arbori binari. În acest caz, elementele tabeloului *hash* au ca valori pointeri spre rădacinile acestor arbori. În felul acesta operațiile de căutare sunt, de obicei, mai performante (excepție fac cazurile cind arborii sunt degenerați, dar această posibilitate este extrem de redusă). În felul acesta, un arbore mare s-a descompus în mai mulți (cel mult *M*) care sunt relativ mici. Astfel, căutarea unei înregistrări nu se mai face într-un arbore mare ci într-unul mult mai mic care a fost selectat pe baza unui calcul simplu.

Exerciții:

- 13.1 Sa se scrie o funcție care calculează dispersia unui sir de caractere prin insumarea codurilor caracterelor sirului respectiv.

Aceasta este funcția *hf* definită în paragraful 13.2.

FUNCȚIA BXIII1

```

unsigned hf(char *s) /* calculeaza dispersia sirului spre care pointeaza s */
{

```

```

    unsigned h;
    for(h=0; *s; ) h += *s++;
    return h*M;
}

```

Observație:

M este o constantă simbolică și reprezintă numărul de elemente al tabloului de pointeri *hash*.

- 13.2 Să se scrie funcția *hcaut* care căută o înregistrare de cheie dată într-o tabelă de dispersie.

Funcția de față a fost definită în paragraful 13.2.

FUNCȚIA BXIII2

```

TNOD *hcaut (char *s,unsigned h)
/* - căuta înregistrarea de cheie identică cu sirul spre care pointează s
   și a cărei dispersie este h;
   - returnează pointerul spre înregistrarea respectivă sau zero
     dacă înregistrarea este absență. */
{
    TNOD *p;

    /* căuta înregistrarea în lista spre care pointează hash[h] */
    for (p=hash[h]; p;p=p->urm)
        if (strcmp(s,p->cuvant)==0)
            /* s-a gasit înregistrarea curentă */
            return p;

    /* nu există înregistrarea căutată */
    return 0;
}

```

- 13.3 Să se scrie funcția *intab* care inserează o înregistrare într-o tabelă de dispersie.

Funcția a fost definită în paragraful 13.2.

FUNCȚIA BXIII3

```

TNOD *intab()
/* - determină înregistrarea curentă și o inserează dacă nu există în tabela de dispersie;
   - returnează pointerul spre înregistrarea respectivă sau zero în cazul cind nu mai
     există înregistrări sau la eroare. */
{
    TNOD *p,*q;
    unsigned h;

    /* rezerva zona și încarcă datele înregistrării */
    if ((p=(TNOD *)malloc(sizeof(TNOD))) && (incnod(p)==1)) {
        /* calculează dispersia cheii înregistrării curente */
        h=hf(p->cuvant);

```

```

/* daca thash[h]!=0, se cauta inregistrarea in tabela */
if(thash[h])
    if(q=hcaut(p -> cuvant,h))
        /* s-a gasit inregistrarea cautata */
        return echivalenta(q,p);
    else{
        /* - inregistrarea curenta nu exista in tabela;
           - se insereaza */
        p -> urm=thash[h];
        thash[h]=p;
        return p;
    }
else{
    /* nu exista inregistrari care sa intre in coliziune cu inregistrarea curenta */
    p -> urm =0;
    thash[h]=p;
    return p;
}
if(p==0){
    printf("memorie insuficienta\n");
    exit(1);
}
elibnod(p);
return 0;
}

```

13.4 Să se scrie un program care citește cuvintele unui text și scrie numărul de apariții al fiecăruiu dintr-ele.

Această problemă a fost rezolvată în capituloarele precedente folosind:

- liste simplu înlănuite;
- liste dublu înlănuite;
- arborii binari.

În exercițiul de față vom folosi o tabelă de dispersie. Problema, ca și în cazul utilizării celorlalte metode, se rezolvă conform pașilor de mai jos:

1. Se citește cuvintul curent;
2. Se caută cuvintul curent în tabela de dispersie.

Dacă el există deja în tabelă, atunci se incrementează numărătorul său.

Altfel se inserează o înregistrare în tabela corespunzătoare cuvintului curent citit și se inițializează numărătorul lui cu valoarea 1.

Acești pași se execută pînă la întlnirea sfîrșitului de fișier care marchează sfîrșitul textului.

În programul de față se folosesc unele funcții care s-au utilizat și în programele precedente. Aceste funcții sunt:

citcuv, incnod, elibnod și echivalenta.

PROGRAMUL BXIII4

```

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <alloc.h>
#include <string.h>

typedef struct tnod {
    char *cuvant;
    int frecventa;
    struct tnod *urm;
} TNOD;

#define M 127

TNOD *thash[M];

#include "BX48.CPP" /* citeuv */
#include "BXI1.CPP" /* incnod */
#include "BXI2.CPP" /* clibnod */
#include "BXIII1.CPP" /* echivalenta */
#include "BXIII1.CPP" /* hf */
#include "BXIII2.CPP" /* hcaut */
#include "BXIII3.CPP" /* intab */

main() /* citește un text și afisează frecvența cuvintelor citite */
{
    TNOD *p;
    int i,j;

    /* initializează tabloul thash cu pointerul nul */
    for(i=0;i < M; i++) thash[i]=0;

    /* - creează tabela de dispersie;
       - fiecare cuvint citit din textul sursă îi corespunde o înregistrare de tipul TNOD. */
    while(intab())
        ;

    /* se listează cuvintele citite și frecvența lor */
    j=0;
    for(i=0;i < M;i++)
        for(p=thash[i];p;j++,p=p->urm){
            printf("cuvintul: %-5ls are\n"
                   "frecventa=%d\n", p->cuvant,p->frecventa);
            if((j+1)%23==0){
                printf("Actionati o tasta pentru a continua\n");
                getch();
            }
        }
}

```

13.5 Să se scrie un program care citește cuvintele dintr-un text și afișează următoarele date despre fiecare cuvint distinct citit:

- caracterele cuvintului;
- daca este rezervat sau nu;
- numărul liniei in care apare prima data cuvintul;
- numarul coloanei in care incepe cuvintul in linia afisata;
- numarul de aparitii al cuvintului in textul respectiv.

Cuvintele care se citește se vor pastra intr-o tabela de dispersie. Înregistrările acestei tabele sint de tipul TNOD definit mai jos:

```
typedef struct nod {
    char *cuvant;
    int frecventa;
    int nr_linie;
    int nr_col;
    Boolean tip;
} TNOD;
```

Variabila *tip* are valoarea 1 daca cuvintul este rezervat si 0 in caz contrar.

In program se defineste o tabela de cuvinte rezervate. Acestea sint alese din setul de cuvinte rezervate ale limbajului C.

Programul se realizează in conformitate cu pașii de mai jos:

1. Se citește un cuvint de la intrare.
Dacă s-a întîlnit sfîrșitul de fișier se trece la pasul 5.
Altfel se continua cu pasul 2.
2. Se construiște înregistrarea de tip TNOD pentru cuvintul citit corect.
3. Se cauta înregistrarea curentă in tabela de dispersie.
Dacă există, se incrementeaza frevența cuvintului din înregistrarea gasită in tabela de dispersie.
Altfel se inserează înregistrarea in tabela.
4. Se trece la pasul 1.
5. Se listeaza datele pastrate in tabela de dispersie.

Prin cuvint se înțelege o succesiune de litere mici și/sau mari.

PROGRAMUL BXIII5

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <alloc.h>
#include <string.h>

typedef enum {False,True} Boolean;

typedef struct tnod{
    char *cuvant;
    int frecventa;
```

```
    int nr_lin;
    int nr_col;
    Boolean tip;
    struct tnod *urm;
} TNOD;

#define M 127

TNOD *thash[M];
int linie,coloana;

int incnod(TNOD *p) /* citeste cuvintul curent si construiește
                     * inregistrarea corespunzatoare lui */
{
    int c,i,j;
    char t[255];
    char *q;
    int inf,sup;
    static char *tcr[]={

        "break","case","char","continue","default","do","double",
        "else","extern","float","for","goto","if","int","long",
        "register","return","short","sizeof","static","struct",
        "switch","typedef","union","unsigned","void","while"
    };

    inf=0;
    sup=sizeof(tcr)/sizeof(char *)-1;
    p->cuvant=0;
    while((c=getchar())!=EOF){
        coloana++;
        if(c >= 'A' && c <= 'Z' || c >= 'a' && c <= 'z') break;
        if(c=='\n'){
            linie++;
            coloana=0;
        }
    }
    if(c==EOF) return -1;
    p->nr_lin = linie;
    p->nr_col = coloana;
    for(i=0;i < 255;i++){
        if(!(c >= 'A' && c <= 'Z') && !(c >= 'a' && c <= 'z')){

            if(c=='\n'){
                linie++;
                coloana=0;
            }
            break;
        }
        t[i]=c;
        c=getchar();
        coloana++;
    }
    if(c==EOF) return -1;
    t[i]='\0';
    if((q=(char *)malloc(i))==0){
```

```

        printf("memorie insuficienta\n");
        exit(1);
    }
    strcpy(q,t);
    p -> cuvant=q;
    p -> frecventa=1;

/* cauta cuvintul citit in tabela de cuvinte rezervate */
for(i=(inf+sup)/2;inf <= sup;i=(inf+sup)/2){
    if((j=strcmp(t,tcr[i]))==0) break;
    if(j < 0) sup=i-1;
    else inf=i+1;
}
if(j)
/* nu este cuvint rezervat */
    p -> tip = False;
else /*cuvint rezervat */
    p -> tip = True;
return 1;
} /* sfirsit incnod */

#include "BXI2.CPP" /* clobnod */
#include "BXIII1.CPP" /* echivalenta */
#include "BXIII1.CPP" /* hf */
#include "BXIII2.CPP" /* heaut */
#include "BXIII3.CPP" /* intab */

main() /* - citeste cuvintele unui text;
           - afiseaza pentru fiecare cuvint distinct din text:
             - caracterele cuvintului;
             - daca este rezervat sau nu;
             - numarul liniei si coloanei in care apare cuvintul de prima data;
             - numarul de aparitii al cuvintului respectiv in textul citit. */
{
    int i,j;
    TNOD *p;

/* initializari */
linie = 1;
coloana = 0;
for(i = 0;i < M;i++) thash[i] = 0;

/* se citesc cuvintele de la intrare si se construieste tabela de dispersie */
while(intab())
    ;

/* afiseaza datele din tabela de dispersie */
j = 0;
for(i = 0;i < M;i++)
    for(p = thash[i];p;j++,p = p -> urm){
        printf("\t%s\n",p -> cuvant);
        if(p -> tip == True) printf(" rezervat");
        else printf("nerezervat");
    }
}

```

```

printf(" linia:%d\t coloana:%d\t aparitii:%d\n",
       p -> nr_lin,p -> nr_col, p -> frecventa);
if((j+1)%10==0){
    printf("Actionati o tasta pentru a continua\n");
    getch();
}
} /* sfirsit main */

```

14. SORTARE

În prezent există mai multe metode de sortare a datelor. Prin *sortare* înțelegem aranjarea datelor într-o anumită ordine. În acest scop considerăm că datele sunt o colecție de elemente de un anumit tip și fiecare element conține o date sau chiar mai multe, în raport cu care se realizează ordonarea. O astfel de date se numește *cheie*.

Sortarea se poate realiza:

1. Fie aranjind datele care se sortează în aşa fel încât cheile lor să corespundă ordinii dorite.
2. Fie ordonând un tablou de pointeri spre datele care trebuie sortate în aşa fel încât considerindu-i pe aceştia în ordinea crescătoare a indicilor tabloului, datele spre care pointeză să formeze o mulțime ordonată în conformitate cu ordinea dorită.

Un exemplu de sortare, de felul celui indicat la punctul 1 de mai sus, este exercițiul 4.40. În acest exemplu se definește o funcție care ordonează crescător elementele unui tablou de tip *double*.

Exercițiul 8.14. este un exemplu de sortare folosind un tablou de pointeri aşa cum se indică la punctul 2 de mai sus. În acest caz se sortează un șir de cuvinte ordonind pointerii care pointeză spre ele. Aceştia sunt păstrați într-un tablou de pointeri spre caractere. În timpul sortării se fac permutări ale acestor pointeri în aşa fel încât, în final, acești pointeri dacea sint considerați în ordinea crescătoare a indicilor, atunci cuvintele spre care pointeză se află în ordine alfabetică.

În ambele cazuri metoda de sortare a fost aceeași, metoda bulelor. De aceea, cele două cazuri de mai sus nu trebuie să se consideră ca fiind metode de sortare diferite.

Metoda bulelor are la bază compararea a două chei învecinate (de indice succesivi sau din noduri succesive). Dacă acestea nu sunt în ordinea cerută, atunci se permute elementele respective sau pointerii spre ele. O caracteristică a acestui algoritm este faptul că la fiecare parcurgere a șirului de date care se sortează, cel puțin un element ajunge să-și ocupe poziția să finală în conformitate cu ordonarea cerută. În cazul exercițiilor amintite mai sus nu s-a ținut seama de această proprietate. Propunem cititorului să scrie exercițiile respective ținând seama de faptul că la fiecare parcurgere cel puțin un element bine determinat este poziționat pe locul lui final. Aceasta înseamnă că la fiecare parcurgere se poate diminua cu 1 numărul elementelor de analizat în următoarea parcurgere. Aceasta conduce la o îmbunătățire a eficienței algoritmului de sortare prin metoda bulelor. Cu toate acestea, această metodă rămâne printre cele cu cel mai mare număr de permutări.

O altă metodă de sortare este legată de folosirea *arborilor binari*. Datele care se sortează se insereză în nodurile unui arbore binar și apoi acestea se listează în

inordine. O astfel de metodă a fost utilizată în exercițiul 12.14. Programul definit în acest exercițiu citește cuvintele dintr-un text și apoi le afișează în ordine alfabetică.

În general, sortarea cu ajutorul *arborilor binari* este mai eficientă decât metoda bulelor.

Există diverse algoritmi de sortare bazați pe arbori. Pentru detalii cititorul poate consulta [3].

În capitolul de față prezentăm două metode de sortare utilizate frecvent în diverse aplicații. Aceste metode sunt relativ simple și în același timp asigură o eficiență mai bună decât metoda bulelor.

Una dintre aceste metode este cunoscută sub denumirea de *sortare Shell* sau *sortare cu micșorarea incrementului*.

Cealaltă metodă se numește *quick sort* (sortare rapidă).

În continuare, vom avea în vedere numai sortări cu chei numerice intregi. Aceasta nu este o restricție esențială, deoarece sortările cu chei nenumerice se realizează similar, folosind tablouri de pointeri.

14.1. Sortare Shell

Metoda *Shell* a fost publicată de către Donald L. Shell, în lucrarea [19]. Metoda pornește de la analiza critică a metodei bulelor. În metoda bulelor, se compară elementele vecine și dacă nu sunt în ordinea cerută, atunci ele se permute. De aceea, elementele care nu sunt în ordine corectă se deplasează cu o singură poziție. Pentru a îmbunătăți acest procedeu, ar trebui să realizăm deplasări cu mai multe locuri ale elementelor, la o permutare a lor. Acest lucru se poate realiza dacă în loc de a compara elemente învecinate, comparăm elemente aflate la o anumită distanță între ele. În cazul în care ele nu sunt în ordinea cerută, ele se vor permuta. În felul acesta elementele se deplasează făcând salturi mai mari decât o poziție.

Distanța dintre elementele comparate se numește *increment*. Incrementul se micșorează după o parcurgere a șirului de elemente care se sortează și se reia parcurgerea de la începutul șirului. De aici rezultă denumirea *sortare cu micșorarea incrementului*.

Înainte de a defini exact metoda de sortare a lui Shell, vom considera un exemplu.

Fie de sortat, în ordine crescătoare, șirul de numere:

173 25 4 300 256 83 95 14 415 3

La început trebuie ales incrementul.

O metodă simplă, dar nu chiar cea mai eficientă, este de a fixa valoarea acestuia la jumătate din numărul total al elementelor de sortat. În cazul de față incrementul va fi:

inc = 5

Apoi, incrementul se injumătățește după fiecare parcurgere a șirului. Algoritmul se termină cind inc = 0.

Algoritmul începe cu a compara primul element al șirului (173) cu al 6-lea (incrementul fiind 5) (83). Cum $173 > 83$, se face permutarea elementelor respective:

83 25 4 300 256 173 95 14 415 3

Compararea următoare se realizează pentru elementul al 2-lea cu al 7-lea, deci se compară 25 cu 95. Ele fiind în ordinea cerută, nu se permute.

Apoi se compară elementele următoare, adică 4 cu 14. Ele fiind în ordinea cerută nu se permute. În continuare se compară elementele următoare, adică 300 cu 415, care și ele sunt în ordinea cerută și deci nu se permute. Elementele următoare 256 și 3 nu sunt în ordine crescătoare și în consecință ele se permute. Se obține:

83 25 4 300 3 173 95 14 415 256

Se injumătățește incrementul și se obține:

inc = 2

Se reia parcurgerea șirului ca mai jos.

Se compară primul element (83) cu al treilea (4). Deoarece $83 > 4$, se permute elementele respective:

4 25 83 300 3 173 95 14 415 256

Apoi se compară al doilea element cu al patrulea, adică 25 cu 300. Acestea sunt în ordine crescătoare și de aceea ele nu se permute. Apoi se compara elementul al treilea cu al cincilea, adică 83 cu 3. Cum $83 > 3$, elementele respective se permute și se obține:

4 25 3 300 83 173 95 14 415 256

În acest moment se permute și primul element cu al treilea, deoarece nici acestea nu se află în ordinea cerută. Se obține:

3 25 4 300 83 173 95 14 415 256

La fiecare permutare a două valori, algoritmul compara valoarea mai mică permuată cu cea precedentă (poziția precedentă se stabilește folosind incrementul) și realizează o nouă permutare dacă este nevoie.

Această comparație se continuă, propagându-se spre începutul șirului, pînă cind:

- nu se mai fac permutări;
sau
- nu mai există elemente în șir (se ajunge la un element care nu mai are precedent în șir).

În continuare se compară elementul al patrulea cu al șaselea, adică 300 cu 173. Acestea se permute:

3 25 4 173 83 300 95 14 415 256

Apoi se compară elementul mai mic permuat (173) cu precedentul lui, conform incrementului, adică cu 25. Cum $25 < 173$, aceste elemente nu se permute.

Se continuă parcurgerea șirului, adică se compara elementul al 5-lea cu al 7-lea (83 și 95). Acestea sunt în ordine crescătoare și deci nu se permute. Apoi se continuă cu elementul al 6-lea, care se compara cu al 8-lea: 300 cu 14. Acestea se permute și se obține:

3 25 4 173 83 14 95 300 415 256

Apoi se compara 173 cu 14 și se constată că și aceste elemente trebuie să se permute:

3 25 4 14 83 173 95 300 415 256

Propagarea comparării spre începutul șirului continuă cu inca un pas și anume, se compara 25 cu 14. Cum $25 > 14$ se permute și aceste elemente obținindu-se:

3 14 4 25 83 173 95 300 415 256

Se continuă cu parcurgerea șirului, comparindu-se elementul al 7-lea cu al 9-lea. Aceste elemente sunt 95 și 415, care sunt în ordine crescătoare, deci ele nu se permute. În sfîrșit, se compara elementul al 8-lea cu al 10-lea și se constată că ele trebuie permute. Se obține:

3 14 4 25 83 173 95 256 415 300

Compararea elementului al 8-lea cu al 6-lea nu conduce la alte permutări și în felul acesta se încheie cea de a doua parcurgere a șirului. Se injumătățește incrementul și se obține:

inc = 1.

În acest moment se parcurge șirul comparind elementele învecinate.

Se compara primul element cu al doilea: 3 cu 14. Ele sunt în ordine crescătoare, deci nu se permute. Se compara 14 cu 4 și cum aceste elemente nu sunt în ordine crescătoare, ele se permute:

3 4 14 25 83 173 95 256 415 300

Apoi se compara elementul al doilea cu precedentul (4 cu 3). Ele sunt în ordinea cerută și nu se mai permute.

O nouă permutare are loc la compararea elementului al 6-lea (173) cu al 7-lea (95). Se obține:

3 4 14 25 83 95 173 256 415 300

Nici această permutare nu se propagă spre începutul șirului deoarece $83 < 95$.

Ultima permutare se realizează la compararea ultimelor două elemente: 415 cu 300. După permutarea lor se încheie cea de treia parcurgere a șirului:

3 4 14 25 83 95 173 256 300 415

Se injumătaște incrementul și se obține $inc = 0$. Aceasta corespunde momentului în care șirul este sortat.

Aplinind metoda bulelor la același exemplu, se constată că sunt necesare mai multe parcurgeri și permutări decât în cazul utilizării metodei Shell.

Eficiența metodei Shell este mult mai mare în comparație cu metoda bulelor în cazul în care se sortează un număr mai mare de date (de ordinul sutelor sau miiilor).

O influență asupra eficienței metodei o are și modul în care se alege incrementul. În lucrarea [11] se arată că valorile incrementului la diferite parcurgeri ale șirului de sortat este bine să fie numere prime între ele.

Pentru alte detalii în legătură cu alegerea incrementului la fiecare pas, vezi [3].

În capitolul de față ne mărginim la alegerea incrementului inițial egal cu $n/2$, unde n este numărul elementelor șirului de sortat. De asemenea, după fiecare parcugere a șirului, acesta se va înjumătăți. Cazul cel mai nefavorabil al acestei metode se obține cind n este o putere a lui doi. Acest caz se poate evita dacă la fiecare parcugere a șirului incrementul se determină prin relația:

$$\text{increment} = \text{increment}/2 + 1, \text{ dacă } \text{increment} > 2.$$

Metoda de sortare Shell poate fi definită astfel:

1. $\text{inc} = n/2$, unde prin n am notat numărul elementelor care se sortează.
2. Se realizează o parcugere a șirului de elemente care se sortează.
3. $\text{inc} = \text{inc}/2$.
4. Dacă $\text{inc} > 0$, atunci se reia de la pasul 2.

Altfel șirul este sortat.

Parcugerea șirului de elemente implică pașii următori:

1. $i = \text{inc}$.
2. $j = i - \text{inc} + 1$.
3. Dacă $j > 0$ și elementele de ordin j și $j + \text{inc}$ nu satisfac criteriul de ordonare, atunci elementele respective se permute.

Altfel se continuă cu pasul 6.

4. $j = j - \text{inc}$.
5. Se reia de la pasul 3.
6. $i = i + 1$.
7. Dacă $i > n$, se termină parcugerea curentă a șirului.

Altfel se reia de la pasul 2.

Pașii 4 și 5 se realizează în cazul în care s-a efectuat permutarea elementelor de ordinul j și $j + \text{inc}$. În acest caz se permute, dacă este necesar, elementele precedente elementului deplasat în poziția j . Elementele precedente sunt:

$$j - \text{inc}, j - 2 * \text{inc}, \dots$$

Acasta se realizează repetind pasul 3 atât timp cit $j > 0$ și se fac permutări. Procedind ca în [2], metoda descriată mai sus se transcrie imediat în limbajul

C, ca mai jos.

Presupunem că elementele de sortat se află într-un tablou tab de tip int și ele sunt în număr de n .

```
void shellsort(int tab[], int n)
/* sortează în ordine crescătoare elementele lui tab prin metoda Shell */
{
    int inc;
    int i, j, t;

    for(inc = n/2; inc > 0; inc /= 2)

        /* se realizează o parcugere a șirului */
        for(i = inc; i < n; i++)
            /* - se compară două elemente aflate la distanța inc unul de altul și dacă este cazul
               se permute;
               - dacă s-a realizat o permutare, atunci elementul deplasat în față
               se compara cu precedentele lui și se permute dacă este cazul. */
            for(j = i - inc; j >= 0 && tab[j] > tab[j+inc]; j -= inc){
                t = tab[j];
                tab[j] = tab[j+inc];
                tab[j+inc] = t;
            }
    }
}
```

14.2. Sortare rapidă

Acastă metodă a fost publicată de C.A.R.Hoare în lucrarea [20].

Ideeua sortării rapide, ca și sortarea lui Shell, se bazează pe faptul că pentru a atinge o viteză mai bună este de dorit ca să permuteam elementele aflate la o distanță mai mare unul de altul decât 1.

Pentru a ne fixa ideile să presupunem că dorim să sortăm crescător elementele tabloului tab de tip int , numărul lor fiind n .

Alegem, în mod arbitrar, un element al tabloului tab și-l atribuim variabilei x . Apoi vom parcurge elementele tabloului și vom face permutări de elemente dacă este necesar, așa încât toate elementele mai mici decât x să fie în stînga tabloului (indici mai mici), iar cele mai mari în dreapta lui. În felul acesta, în etapa următoare, se poate considera că avem de sortat două tablouri distincte, unul care conține elemente mai mici decât x și unul care conține elemente mai mari decât x . Aceste tablouri se sortează și ele prin același procedeu. Se poate considera că, acest procedeu constă în divizarea unui tablou în două părți, așa încât elementele uneia să fie toate mai mici decât elementele celuilalt. De aceea, această metodă se mai numește și metoda prin *interschimb de poziții*.

Metoda sortării rapide poate fi realizată ca mai jos:

1. Se alege un element oarecare al tabloului, de exemplu elementul aflat la mijlocul tabloului și se atribuie lui x .
2. Se parcurge tabloul de la stînga la dreapta pînă cind se găsește un element

- tab[i] >= x.
3. Se parcurge tabloul de la dreapta la stînga pînă cînd se găsește un element tab[j] <= x.
 4. Se permută intre ele elementele tab[i] cu tab[j], dacă i nu-l depășește pe j.
 5. Se continuă pașii 2, 3 și 4 de mai sus pînă cînd se ajunge ca i > j.
Cînd i > j, tabloul este divizat în două.

Pașii de mai sus se continuă apoi, asupra fiecăreia dintre cele două părți în care s-a divizat tabloul inițial. În felul acesta se divide fiecare parte în două părți, realizindu-se o divizare în 4 părți a tabloului inițial și aşa mai departe. Divizarea continuă pînă cînd fiecare parte conține un singur element. În acest moment tabloul are elementele sortate în ordine crescătoare.

Evident, se poate defini un algoritm analog pentru a sorta elementele unui tablou în ordine descrescătoare.

Mai jos, exemplificăm metoda sortării rapide folosind tabloul la care s-a aplicat metoda lui Shell:

173 25 4 300 256 83 95 14 415 3

Sortarea se realizează conform pașilor de mai jos:

1. inf = 1 (indicele primului element).
2. sup = 10 (indicele ultimului element).
3. Se alege elementul de indice:
 $(inf+sup)/2$
deci
 $x = tab[(1+10)/2] = tab[5] = 256.$
4. i = inf.
5. j = sup.
6. Se parcurg elementele tabloului de la stînga la dreapta pînă se găsește un element mai mare sau egal cu 256.
Acesta este 300, deci i = 4 și tab[i] = 300.
7. Se parcurg elementele tabloului de la dreapta spre stînga pînă cînd se găsește un element mai mic sau egal cu 256.
Acesta este chiar ultimul element.
Deci j = 10 și tab[j] = 3.
8. Cum i < j, se permută tab[i] cu tab[j]:

173 25 4 3 256 83 95 14 415 300.

9. i = i+1, deci i = 5.
10. j = j-1, deci j = 9.
Cum i < j, se reia de la punctul 6:
6. Se obține i = 5, deoarece tab[5] = 256 și 256 >= 256 este primul element cu această proprietate.
7. Se obține j = 8, deoarece tab[8] = 14 și 14 < 256 este primul element întlnit de la dreapta spre stînga care să aibă această proprietate.

8. Cum i < j, se permută cele două elemente:

173 25 4 3 14 83 95 256 415 300.

9. i = i+1, deci i = 6.

10. j = j-1, deci j = 7.

Cum i < j se reia de la punctul 6.

6. tab[8] = 256 >= 256 și acesta este primul element cu această proprietate.
i = 8.
7. tab[7] = 95 < 256.
j = 7.
8. Cum i > j, nu se mai fac permutări.

În acest moment s-a terminat procesul de divizare cu i = 8 și j = 7.

Tabloul s-a divizat în două părți:

- partea intîi: tab[1], tab[2], ..., tab[j = 7];
- partea a două: tab[8], tab[9] și tab[10].

În continuare se aplică aceeași procedură la fiecare din aceste două părți pentru prima parte se ia:

inf = 1 și sup = 7.

La pasul 3 se obține:

3. $x = tab[(1+7)/2] = tab[4] = 3.$
4. i = 1.
5. j = 7.
6. tab[1] este primul element aşa încit tab[i] >= x.
Deci i = 1.
7. tab[4] este primul element aşa încit tab[j] <= x.
Deci j = 4.
8. Cum i < j, se permută tab[i] cu tab[j]:

3 25 4 173 14 83 95.

9. i = i+1.

Deci i = 2.

10. j = j-1.

Deci j = 3.

Cum i < j, procesul se reia de la punctul 6.

6. Deoarece tab[i] = 25 > x = 3, i nu se modifică.

7. tab[3] = 4, tab[2] = 25 și tab[1] = 3 deci tab[1] <= x = 3 și j = 1.

8. Cum i = 2 și j = 1, i > j și nu se mai fac permutări.

În acest moment s-a obținut o nouă divizare.

Cele două părți sunt:

- prima parte: 3;
- a două parte: 25 4 173 14 83 95.

deoarece prima parte s-a redus la un element, se continuă cu partea a două:

508

509

- $\inf = 2$.
- $\sup = 7$.
- $x = \text{tab}[(2+7)/2] = \text{tab}[4] = 173$.
- $i = 2$.
- $j = 7$.
- Se obține $i = 4$, deoarece $\text{tab}[4] \geq x$ este primul element cu această proprietate.
- $j = 7$, deoarece $\text{tab}[7] = 95 < x$.
- $i < j$ deci se permute $\text{tab}[i]$ cu $\text{tab}[j]$:

25 4 95 14 83 173.

- $j = i+1$.
Deci $i = 5$.
- $j = j-1$.
Deci $j = 6$.
Cum $i < j$, se reia de la pasul 6.
- Se obține $i = 7$ deoarece singurul element cu proprietatea $\text{tab}[i] \geq 173$ este $\text{tab}[7] = 173$.
- j nu se modifica deoarece $\text{tab}[6] = 83 < x = 173$.
- $i > j$, deci elementele respective nu se permute.

S-a obținut o nouă divizare a tabloului:
 – prima parte: 25 4 95 14 83;
 – partea a doua: 173.

Se continuă cu prima parte:

- $\inf = 2$.
- $\sup = 6$.
- $x = \text{tab}[(2+6)/2] = \text{tab}[4] = 95$.
- $i = 2$.
- $j = 6$.
- $i = 4$, deoarece $\text{tab}[4] = 95 \geq 95$.
- j rămîne 6, deoarece $\text{tab}[6] = 83 < 95$.
- $i < j$ și de aceea se permute $\text{tab}[i]$ cu $\text{tab}[j]$:

25 4 83 14 95.

- $i = i+1$; $i = 5$.
- $j = j-1$; $j = 5$.
Cum $i \leq j$ se reia de la punctul 6.
- $i = 6$ deoarece $\text{tab}[6] = 95 \geq 95$.
- j rămîne 5 deoarece $\text{tab}[5] = 14 < 95$.
- $i > j$, deci elementele respective nu se permute.

S-a obținut o nouă divizare a tabloului:

- prima parte: 25 4 83 14;
- partea a doua: 95.

Se continuă cu prima parte:

- $\inf = 2$.
- $\sup = 5$.
- $x = \text{tab}[(2+5)/2] = \text{tab}[3] = 4$.
- $i = 2$.
- $j = 5$.
- i rămîne pe loc deoarece $\text{tab}[2] = 25 > x = 4$.
- $j = 3$ deoarece $\text{tab}[3] = 4 \leq x = 4$.
- $i < j$, deci se face permutare:

4 25 83 14.

- $i = i+1$, deci $i = 3$.
- $j = j-1$, deci $j = 2$.

Deoarece $i > j$, s-a obținut o nouă divizare:
 – prima parte: 4;
 – partea a doua: 25 83 14.

Se continuă cu partea a doua:

- $\inf = 3$.
- $\sup = 5$.
- $x = \text{tab}[(3+5)/2] = \text{tab}[4] = 83$.
- $i = 3$.
- $j = 5$.
- $i = 4$ deoarece $\text{tab}[4] = 83 \geq 83$.
Este primul element cu această proprietate.
- j rămîne pe loc, deoarece $\text{tab}[5] = 14 < 83$.
- $i < j$, deci se face permutarea:

25 14 83.

- $i = i+1$, deci $i = 5$.
- $j = j-1$, deci $j = 4$.
Deoarece $i > j$, s-a obținut o nouă divizare:

- prima parte: 25 14;
- partea a doua: 83.

Se continuă cu prima parte:

- $\inf = 3$.
- $\sup = 4$.
- $x = \text{tab}[(3+4)/2] = \text{tab}[3] = 25$.
- $i = 3$.

5. $j = 4$.
 6. i rămîne pe loc deoarece $tab[3] = 25 \geq x = 25$.
 7. j rămîne pe loc deoarece $tab[4] = 14 \leq x = 25$.
 8. Cum $i < j$, se face permutare:

14 25

9. $i = i + 1$, deci $i = 4$.
 10. $j = j - 1$, deci $j = 3$.

Cum $i \geq j$, s-a obținut o nouă divizare:

- prima parte: 14;
 - partea a doua: 25.

În momentul de fată tabloul *tab* are elementele ordonate ca mai jos:

3 4 14 25 83 95 173 256 415 300

E) este aproape sortat. Partea neînțărată este partea a doua de la prima divizare;

256 415 300.

1. $\inf = 8.$
 2. $\sup = 10.$
 3. $x = \text{tab}[(8+10)/2] = \text{tab}[9] = 415.$
 4. $i = 8.$
 5. $j = 10.$
 6. $i = 9$, deoarece $\text{tab}[9] = 415 \geq x = 415$ și acesta este primul element cu această proprietate.
 7. j rămîne pe loc, deoarece $\text{tab}[10] = 300 < x = 415.$
 8. $i \leq i$, deci se face permutarea;

256 300 415

9. $i = i+1, i = 10$
10. $i = i-1, i = 3$

Generalization to other fields

- prima parte: 256 300;

Se continuă cu prima parte:

1. inf = 8.
 2. sup = 9.
 3. x = tab[(8+9)/2] = tab[8] = 256.
 4. i = 8.
 5. j = 9

6. i râmine pe loc, deoarece $\text{tab}[i] = 256 \geq 256$.
 7. $j = 8$, deoarece $\text{tab}[j] = 256 \leq 256$ și acesta este primul element cu aceasta proprietate.
 8. Cum $i = j$, nu se face permutare.
 9. $i = i+1$, deci $i = 9$.
 10. $j = j-1$, deci $j = 7$.

Din acest exemplu se observă că diviziunea este în mare măsură dependentă de elementul separator ales la fiecare pas. Dacă acesta este chiar elementul minim dintre elementele tabloului supus divizării, atunci prima parte se reduce chiar la acest element. În mod analog, dacă elementul separator este elementul maxim, atunci partea a doua se reduce chiar la acest element.

Acestea sunt cazuile cele mai ineficiente deoarece tabloul se divide în aşa fel incit una din părți conține un singur element.

În general, divizările pentru tablouri mari este bine să fie cît mai echilibrate, adică cele două părți să fie de lungimi apropriate. Nu există însă un procedeu simplu care să asigure acest fapt. Alegerea elementului de la mijlocul tabloului este o metodă simplă și destul de practică. Mai jos o să indicăm și alte metode pentru a alege elementul separator.

În continuare, descriem o funcție, în limbajul C, pentru metoda ilustrată mai sus. Numim *quicksort* funcția respectivă. Procedura *quicksort* se descrie simplu dacă funcția respectivă este recursivă. Natura ei recursivă decurge din faptul că ea constă din următoarele:

- se divide tabloul în două părți în aşa fel incit elementele unei părți să nu depășească elementele celeilalte părți;
 - pentru fiecare parte se aplică aceeași procedură de divizare pînă cind lungeimea partii devine 1.

Tabloul este sortat cind lungimea fiecarei parti a devenit egală cu 1.

Procedura de divizare constă, aşa cum s-a văzut mai sus, din selectarea elementului x din mijlocul tabloului, iar cele două părți se obțin permutând elementele mai mari sau egale cu x din stînga tabloului, cu cele mai mici sau egale cu x din dreapta tabloului.

Un element $x[i]$ este în stînga lui $x[j]$, dacă $i < j$. În mod analog, un element $x[k]$ este în dreapta lui $x[i]$, dacă $k \geq i$.

În anumite cazuri este necesar să se permute chiar și elementul selectat x , deoarece în caz contrar nu se poate realiza divizarea. Într-adevăr x nu-și schimbă poziția dacă în stînga lui sunt atîtea elemente mai mari decît el cîte sunt în dreapta lui mai mici decît el. Evident, această condiție nu este îndeplinită frecvent și de aici rezultă necesitatea permutării lui.

Functia *quicksort* are parametru

tab - Tabloul care se sortează

- Indicele inferior al părții care se divide.
- Indicele superior al părții care se divide.

Dacă tab are n elemente, atunci funcția *quicksort* se apelează astfel:

```
quicksort(tab,0,n-1);
```

Definim mai jos funcția *quicksort*.

```
void quicksort(int tab[], int inf, int sup)
/*- sortează în ordine crescătoare tabloul tab, folosind metoda sortării rapide;
- elementul separator este elementul din mijlocul tabloului. */
{
    int i, j, x, t;

    i = inf;
    j = sup;

    /* se alege elementul separator */
    x = tab[(i+j)/2];

    /* se face divizarea */
    do {
        /* avans peste elementele din stînga lui x și mai mici decît el */
        while(i < sup && tab[i] < x)
            i++;

        /* avans peste elementele din dreapta lui x și mari decît el */
        while(j > inf && tab[j] > x)
            j--;

        if(i <= j) {
            /* dacă i < j se permute elementele */
            t = tab[i];
            tab[i] = tab[j];
            tab[j] = t;
        }

        /* se trece la elementele următoare */
        i++;
        j--;
    }

    /* dacă i > j s-a realizat divizarea:
       - prima parte:
         inf - limita inferioară;
         j - limita superioară;
       - partea a două:
         i - limita inferioară;
         sup - limita superioară;
       dacă i <= j, procesul de divizare continua. */
}

while(i <= j)

/* dacă inf < j, se trece la divizarea primei părți deoarece aceasta conține
mai mult decît un element */
if(inf < j) quicksort(tab,inf,j);
```

```
/*dacă j < sup se trece la divizarea partii a două deoarece conține mai mult decît un element */
if(i < sup) quicksort(tab,i,sup);
```

Revenim la problema determinării elementului separator. Pentru detalii, cititorul poate consulta lucrările [3] și [11].

O metodă relativ simplă și care dă adesea rezultate bune constă în alegerea *medianei* următoarelor elemente:

- primul element al tabloului;
- elementul aflat la mijlocul tabloului;
- ultimul element al tabloului.

Amintim că dacă a, b, c sunt trei numere și între ele există relația

$$a < b < c$$

atunci *mediana* lor este b .

În felul acesta se elimină situațiile în care elementul separator este elementul minim sau maxim al tabloului.

Propunem cititorului să modifice funcția *quicksort* în aşa fel încit elementul separator să fie ales ca mediană a celor trei elemente amintite mai sus.

O altă metodă, descrisă amănuntit în [11] constă în alegerea ca element separator a *primului element* al tabloului și deplasarea acestuia pe locul lui final în procesul divizării tabloului.

În stînga lui se deplasează elementele mai mici decît el, iar în dreapta cele mai mari decît el. În felul acesta tabloul este divizat în două părți, iar elementul separator, fiind pe locul lui final, nu aparține nici uneia din cele două părți pe care le separă.

Mai jos definim funcția *quicksort* care utilizează această metodă la alegerea elementului separator.

În principiu, această metodă constă în următoarele:

Notăm cu inf și sup ($inf < sup$) indicii care definesc limita inferioară, respectiv superioară a părții din tabloul tab care urmează a fi divizată.

1. $x = tab[inf]$.
2. $i = inf$.
3. $j = sup$.
4. Dacă $i < j$ se trece la pasul 5.
Altfel la pasul 9.
5. Se determină un element $tab[i]$, incrementind pe i aşa încit $tab[i] > x$ și $i \leq sup$.
6. Se determină un element $tab[j]$, decrementind pe j , aşa încit $tab[j] \leq x$.
7. Dacă $i < j$, atunci se permute $tab[i]$ cu $tab[j]$.
Altfel se trece la pasul 8.
8. Se trece la pasul 4.

9. Se deplasează $\text{tab}[inf]$ pe locul lui final permutându-l cu $\text{tab}[j]$.

În acest moment tabloul este divizat astfel:

- prima parte:
 inf - limita inferioară;
 $j-1$ - limita superioară.
- $\text{tab}[j]$:
are poziția sa finală și nu se va mai permuta.
- partea a două:
 $j+1$ - limita inferioară;
 sup - limita superioară.

Procesul se reia pentru părțile care nu s-au redus încă la un singur element (limita inferioară a părții respective este mai mică decit cea superioară).

Exemplificăm metoda pe același exemplu:

173 25 4 300 256 83 95 14 415 3

$\text{inf} = 1$ și $\text{sup} = 10$

1. $x = \text{tab}[1] = 173$.
2. $i = \text{inf} = 1$.
3. $j = \text{sup} = 10$.
4. $i < j$, se trece la pasul 5.
5. $\text{tab}[i] > 173$ pentru $i = 4$; $\text{tab}[4] = 300$.
6. $\text{tab}[j] < 173$ pentru $j = 10$; $\text{tab}[10] = 3$.
7. Cum $i = 4 < j = 10$, se permute cele două elemente:

173 25 4 3 256 83 95 14 415 300.

8. Se reia de la pasul 4.
4. $i < j$, se trece la pasul 5.
5. $\text{tab}[5] = 256 > 173$, deci $i = 5$.
6. $\text{tab}[8] = 14 < 173$, deci $j = 8$.
7. $i < j$, deci se permute cele 2 elemente:

173 25 4 3 14 83 95 256 415 300.

8. Se reia de la pasul 4.
4. $i < j$, se trece la pasul 5.
5. $\text{tab}[8] > 173$, deci $i = 8$.
6. $\text{tab}[7] < 173$, deci $j = 7$.
7. Cum $i > j$, nu se face permutare.
8. Se reia de la pasul 4.
4. Cum $i > j$, se trece la pasul 9.

9. Se permute $\text{tab}[inf] = \text{tab}[1] = 173$, cu $\text{tab}[j] = \text{tab}[7] = 95$:

95 25 4 3 14 83 173 256 415 300.

Tabloul se descompune în:

- prima parte: A 95 25 4 3 14 83;
- 173 ocupă locul lui final;
- partea a două: B 256 415 300.

Aplicăm același algoritm la partea A:

$\text{inf} = 1$, $\text{sup} = 6$

1. $x = \text{tab}[inf] = 95$.
2. $i = 1$.
3. $j = 6$.
4. $i < j$, se trece la pasul 5.
5. Deoarece $\text{tab}[i] = 95$ este elementul maxim, i crește pînă la valoarea maximă, adică i devine egal cu 6.
6. $\text{tab}[j] = 83 < x = 95$, înseamnă că j rămîne nemodificat.
7. $i = j = 6$, deci nu se face permutare.
8. Se reia de la pasul 4.
4. $i = j$, se trece la pasul 9.
9. Se permute $\text{tab}[inf] = \text{tab}[1] = 95$ cu $\text{tab}[j] = 83$:

83 25 4 3 14 95.

Tabloul se descompune în:

- prima parte: AA 83 25 4 3 14;
- 95 ocupă locul lui final;
- partea a două: vida.

Aplicăm același algoritm la partea AA:

$\text{inf} = 1$, $\text{sup} = 5$

1. $x = \text{tab}[inf] = 83$.
2. $i = 1$.
3. $j = 5$.
4. $i < j$, se trece la pasul 5.
5. Deoarece $\text{tab}[i] = 83$ este elementul maxim, i crește pînă la valoarea maximă, adică i devine egal cu 5.
6. $\text{tab}[j] = 14 < x = 83$, înseamnă că j rămîne nemodificat.
7. $i = j = 5$, deci nu se face permutare.
8. Se reia de la pasul 4.
4. $i = j$, se trece la pasul 9.
9. Se permute $\text{tab}[inf] = \text{tab}[1] = 83$ cu $\text{tab}[j] = 14$:

14 25 4 3 83.

Tabloul se descompune în:

- prima parte: AAA 14 25 4 3;
- 83 ocupă locul lui final;
- partea a doua: vidă.

APLICAM ACELAȘI ALGORITM LA PARTEA AAA:

inf = 1, sup = 4

1. $x = \text{tab}[\text{inf}] = \text{tab}[1] = 14.$
2. $i = 1.$
3. $j = 4.$
4. $i < j$, se trece la pasul 5.
5. i crește la valoarea 2, deoarece $\text{tab}[2] = 25 > x$.
6. j nu se modifică deoarece $\text{tab}[j] = \text{tab}[4] = 3 < x$.
7. $i < j$, se permute $\text{tab}[2]$ cu $\text{tab}[4]$:

14 3 4 25.

8. Se reia de la pasul 4.
9. $i < j$, se trece la pasul 5.
10. i crește pînă la 4, deoarece $\text{tab}[4] = 25 > x$.
11. j devine egal cu 3, deoarece $\text{tab}[3] = 4 < x$.
12. $i > j$, deci nu se face permutare.
13. Se trece la pasul 4.
14. $i > j$, deci se trece la pasul 9.
15. Se permute $\text{tab}[\text{inf}] = \text{tab}[1] = 14$ cu $\text{tab}[j] = \text{tab}[3] = 4$:

4 3 14 25.

Tabloul se descompune în:

- prima parte: AAAA 4 3;
- 14 ocupă locul lui final;
- partea a doua: 25.

APLICAM ACELAȘI ALGORITM LA PARTEA AAAA:

inf = 1, sup = 2

1. $x = \text{tab}[\text{inf}] = 4.$
2. $i = 1.$
3. $j = 2.$
4. $i < j$, se trece la pasul 5.
5. Deoarece $\text{tab}[i] = 4$ este elementul maxim, i crește pînă la valoarea maximă, adică i devine egal cu 2.
6. $\text{tab}[j] = \text{tab}[2] = 3 < x = 4$. Înseamnă că j rămîne nemodificat.
7. $i = j = 2$, deci nu se face permutare.
8. Se reia de la pasul 4.

4. $i = j$, se trece la pasul 9.

9. Se permute $\text{tab}[\text{inf}]$ cu $\text{tab}[j]$:

3 4.

Tabloul se descompune în:

- prima parte: 3;
- 4 ocupă locul lui final;
- partea a doua: vidă.

ÎN ACEST MOMENT ELEMENTELE TABLOULUI SINT ORDONATE ASTFEL:

3 4 14 25 83 95 173 256 415 300.

ULTIMELE 3 ELEMENTE FORMEAZĂ PARTEA B DE LA PRIMA DIVIZIUNE A ACELEAȘI METODE:

inf = 8, sup = 10

1. $x = \text{tab}[\text{inf}] = \text{tab}[8] = 256.$
2. $i = 8.$
3. $j = 10.$
4. $i < j$, se trece la pasul 5.
5. i devine egal cu 9 deoarece $\text{tab}[9] = 415 > x$.
6. j devine egal cu 8 deoarece $\text{tab}[10] > x$ și $\text{tab}[9] > x$, $\text{tab}[8]$ fiind primul care nu are această proprietate.
7. $i > j$, deci nu se face permutare.
8. Se reia de la pasul 4.
9. $i > j$, se trece la pasul 9.
10. Cum $\text{inf} = j$, nu se face permutare.

Tabloul se descompune în:

- prima parte: vidă;
- 256 ocupă locul lui final;
- partea a doua: BB 415 300;

APLICAM ACELAȘI ALGORITM LA PARTEA BB ȘI ÎN FINAL SE OBȚINE:

3 4 14 25 83 95 173 256 300 415.

Mai jos, transcriem acest procedeu în limbajul C.

```
void quicksortinf(int tab[], int inf, int sup)
/* - sortează în ordine crescătoare tabloul tab folosind metoda sortării rapide;
 - elementul separator este tab[inf] care se deplasează pe locul lui final. */
{
    int i,j,t,x;

    x = tab[inf];
    i = inf;
    j = sup;
```

```

/* divizarea tabloului */
while(i < j){

/* se determină cel mai mic indice i astfel ca tab[i] > x și i <= sup;
   - se avansaza peste elementele mai mici sau egale cu x;
   - i = sup daca nu exista un astfel de element.*/
   while(tab[i] <= x && i < sup) i++;

/* - se determină cel mai mare indice j astfel ca tab[j]<=x;
   - se avansaza peste elementele mai mari decit x.*/
   while(tab[j] > x) j--;

/* daca i < j, se permuta tab[i] cu tab[j] */
if(i < j){ /* se face permutare */
   t = tab[i];
   tab[i] = tab[j];
   tab[j] = t;
}

/* - s-a terminat divizarea tabloului;
   - tab[inf] se deplaseaza pe locul lui final care este j;
   - totodata tab[j] poate fi deplasat in pozitia inf;
   - deci se permuta tab[inf] cu tab[j];
   - permutarea nu are loc daca j = inf. */
if(j > inf){
   tab[inf] = tab[j];
   tab[j] = x;
}

/* s-a realizat divizarea:
   - prima parte:
      tab[inf],tab[inf+1],...,tab[j-1];
   - tab[j] ocupa pozitia lui finala;
   - partea doua:
      tab[j+1],tab[j+2],...,tab[sup]. */
if(inf < j-1)

   /* prima parte are cel putin 2 elemente */
   quicksortinf(tab,inf,j-1);

   if(j+1 < sup)
      /* partea a doua are cel putin 2 elemente */
      quicksortinf(tab,j+1,sup);
}

```

Menționăm că în lucrarea [11], această metodă este transcrisă în limbajul C într-o funcție nerecursivă.

Exerciții:

14.1 Să se scrie un program care realizează următoarele:

- citește o succesiune de numere separate prin caractere albe și precedate

- de numărul lor;
- sortează în ordine crescătoare numerele citite și apoi le afișează; sortarea se realizează folosind următoarele metode:
 - metoda bulelor;
 - sortare Shell;
 - sortare rapidă cu alegerea elementului din mijloc ca element separator;
 - sortare rapidă cu alegerea primului element ca element separator.
- afișează numărul permutărilor efectuate în cadrul fiecărei metode de sortare utilizate.

PROGRAMUL BXIV1

```

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

#include "BVIII2.CPP" /* pcit_int */
#include "BVIII3.CPP" /* pcit_int lim */
#include "BVIII8.CPP" /* pdcit */
#include "BVIII9.CPP" /* pdvcit */

#define MAX 100

void dperm(double *p1,double *p2)
/* permută valorile de tip double spre care pointează p1 și p2 */
{
   double temp;

   temp = *p1;
   *p1 = *p2;
   *p2 = temp;
}

void dcopiază(double tab1[],double tab2[],int n)
/* copiază primele n elemente din tabloul tab2 în tabloul tab1 */
{
   int i;
   for(i=0;i < n;i++)
      tab1[i]=tab2[i];
}

void afistab(double tab[],int n)
/* - afișează primele n elemente din tabloul tab;
   - pe o linie se afișează 5 elemente. */
{
   char text[] = "Actionati o tasta pentru a continua\n";
   int i,j;
   double d;

```

```

printf("\n");
j=1;
for(i = 0; i < n; i++){
    d = tab[i];
    printf("%d ",d);
    if(i%5 ==4){
        printf("\n");
        if(j%22==0){
            printf(text);
            getch();
        }
    }
}
printf(text);
getch();
}

void afisantot(char *p) /* afiseaza un anot */
{
    printf("\n\n\t\t%s",p);
    printf("Actionati o tasta pentru a continua\n");
    getch();
}

void afisnrperm(char *p,long n)
/* afiseaza textul spre care pointeaza p si valoarea lui n */
{
    printf("\n Metoda %t numar permutari=%ld\n",p,n);
}

long bubblesort(double tab[],int n)
/* - sorteaza elementele tabloului tab in ordine crescatoare folosind metoda buzelor;
   - returneaza numarul permutarilor efectuate. */
{
    int ind,i;
    long s;

    ind = 1;
    s=0;
    while(ind){
        ind = 0;
        for(i = 0; i < n-1; i++)
            if(tab[i] > tab[i+1]){
                dperm(&tab[i],&tab[i+1]);
                s++; /* numara permutari */
                ind = 1;
            } /* sfarsit if */
    } /* sfarsit while */
    return s;
} /* sfarsit bubblesort */

long shellsort(double tab[],int n)
/* - sorteaza elementele tabloului tab in ordine crescatoare folosind metoda lui Shell;
   - returneaza numarul permutarilor efectuate. */

```

```

(
    int inc;
    int i,j;
    long s;

    s = 0;
    for(inc = n/2; inc > 0; inc /= 2)
        for(i = inc; i < n; i++)
            for(j = i-inc; j >= 0 && tab[j] > tab[j+inc]; j -= inc)
                dperm(&tab[j],&tab[j+inc]);
                s++;
    }
    return s;
} /* sfarsit shellsort */

void quicksort(double tab[],int inf,int sup)
/*- sorteaza elementele tabloului tab in ordine crescatoare folosind metoda sortarii rapide;
   - elementul separator este elementul din mijlocul tabloului;
   - numarul permutarilor realizate se atribuie variabilei globale s. */
{
    extern long s;
    int i,j;
    double x;

    i = inf;
    j = sup;
    x = tab[(i+j)/2];
    do{
        while(i < sup && tab[i] < x) i++;
        while(j > inf && tab[j] > x) j--;
        if(i <= j){
            if(i < j){
                dperm(&tab[i],&tab[j]);
                s++;
            }
            i++; j--;
        }
    }while(i <= j);
    if(inf < j) quicksort(tab,inf,j);
    if(i < sup) quicksort(tab,i,sup);
} /* sfarsit quicksort */

void quicksortinf(double tab[],int inf,int sup)
/*- sorteaza elementele tabloului tab in ordine crescatoare folosind metoda sortarii rapide;
   - elementul separator este primul element al tabloului si acesta se deplascaza pe locul lui final;
   - numarul permutarilor realizate se atribuie variabilei globale s. */
{
    extern long s;
    int i,j;
    double x;

    x = tab[inf];
    i = inf; j = sup;
    while(i < j){

```

```

        while(tab[i] <= x && i < sup) i++;
        while(tab[j] > x) j--;
        if(i < j){
            dperm(&tab[i],&tab[j]);
            s++;
        }
    }
    if(j > inf){
        tab[inf] = tab[j];
        tab[j] = x;
        s++;
    }
    if(inf < j-1) quicksortinf(tab,inf,j-1);
    if(j+1 < sup) quicksortinf(tab,j+1,sup);
} /* sfarsit quicksortinf*/

long s;

main()
/* - citeste o succesiune de numere separate prin caractere albe si precedate de numarul lor;
 - sorteaza in ordine crescatoare numerele citite si apoi le afiseaza;
 - sortarea se face folosind urmatoarele metode:
    metoda bulelor;
    sortare shell;
    sortare rapida cu alegerea elementului din mijloc ca separator;
    sortare rapida cu alegerea primului element ca element separator;
 - afiseaza numarul permutarilor efectuate in cadrul fiecarei metode utilizate. */
{
    double tini[MAX],tsort[MAX];
    int n;
    long a,b,c,d;

    /* citeste numerele de sortat */
    if((n = pdvcit(MAX,tini)) < 2){
        printf("nu s-au tastat nici 2-numere\n");
        exit(1);
    }

    /* sortare prin metoda bulelor */
    dcopiazza(tsort,tini,n); /* tsort = tini */
    a = bubblesort(tsort,n);
    afisantet("metoda bulelor");
    afistab(tsort,n);

    /* sortare prin metoda Shell */
    dcopiazza(tsort,tini,n);
    b = shellsort(tsort,n);
    afisantet("metoda Shell");
    afistab(tsort,n);

    /* sortare rapida - elementul separator este elementul din mijlocul tabloului */
    dcopiazza(tsort,tini,n);
    s = 0;
    quicksort(tsort,0,n-1);
}

```

```

    c = s;
    afisantet("sortare rapida: elementul din mijloc");
    afistab(tsort,n);

    /* sortare rapida - elementul separator este primul element */
    dcopiazza(tsort,tini,n);
    s = 0;
    quicksortinf(tsort,0,n-1);
    d = s;
    afisantet("sortare rapida: primul element");
    afistab(tsort,n);

    /* se afiseaza numarul permutarilor */
    afisnrperm("Bulelor",a);
    afisnrperm("Shell",b);
    afisnrperm("Sortare rapida: mijloc",c);
    afisnrperm("Sortare rapida: inferior",d);
}

```

15. DIN NOU DESPRE PREPROCESARE ÎN C

În capitolul 1 s-a arătat că preprocessarea este o fază care precede compilarea propriu-zisă. Preprocesorul limbajului C este relativ simplu. El, în principiu, execută *substituții* de texte. Prin intermediul lui se poate realiza:

- incluzări de fișiere sursă;
- definiții și apeluri de macrouri;
- compilare condiționată.

Preprocesorul recunoaște construcțiile care încep prin caracterul diez (#). Acestea se mai numesc și *directive*.

Așa de exemplu, incluzările de fișiere se realizează cu ajutorul construcției `#include`. Aceasta construcție a fost descrisă în capitolul 1.

În capitolul de față vom reveni asupra construcției `#define`.

În general, construcția `#define` se folosește la definirea de macrouri.

Apoi, ne vom ocupa de compilarea condiționată.

15.1. Definiții și apeluri de macrouri

Definiția constantelor simbolice este un caz particular de definiție de macro. Amintim că o constantă simbolică se definește printr-o construcție de formă:

`#define nume succesiune de caractere`

Efectul acestei construcții constă în substituția în textul programului a numelui aflat după `#define` cu *succesiune de caractere* din construcția `#define` respectivă.

Substituția se realizează pentru orice apariție a lui *nume* în textul sursă care urmează după definirea lui prin construcția `#define`, exceptând cazurile în care *nume* apare într-un șir de caractere sau într-un comentariu.

Efectul construcției `#define` se anulează la întărirea construcției:

`#undef nume`

După o astfel de construcție, *nume* poate fi redefinit printr-o altă construcție `#define`.

La scrierea construcției `#define`, *nume* este precedat și urmat de cel puțin un spațiu. Succesiunea de caractere care se substituie numelui începe cu primul caracter care nu este alb. Ea se poate continua pe rindul următor terminând-o prin caracterul backslash (\).

Un *macro* este o facilitate generală de prelucrare a textelor prin substituție. Un macro, ca și o funcție, are o definiție și unul sau mai multe apeluri. Definiția

macroului trebuie să preceată orice apel al său. Ea definește textul care urmează să se substituie la fiecare apel al macroului. Acest text poate fi variabil, variind de la apel la apel. De aceea, un astfel de text, de obicei, va conține parametri. Aceștia se concretizează la fiecare apel. Parametrii utilizati în definiția macroului se numesc *formali*. Valorile lor dintr-un apel al macroului sunt parametrii *efectivi* sau *concreți*.

Forma generală a unei definiții de macro este următoarea:

`#define nume(p1,p2,...,pn) text`

unde:

- | | |
|---------------------|---|
| <i>nume</i> | - Este numele macroului. |
| <i>p1,p2,...,pn</i> | - Sunt parametri formalii ai macroului. |
| <i>text</i> | - Este textul de substituție.
- El conține parametri formalii p1,p2,...,pn.
- Dacă este nevoie, el se poate continua pe rindul următor folosind caracterul <i>backslash</i> . |

Fiecare p1,p2,...,pn este un nume.

Dacă macroul nu are parametri, atunci este absentă toată construcția (p1,p2,...,pn) și evident, textul de substituție devine constant (nu mai conține parametri). În acest caz definiția de macro devine o definiție de constantă simbolică.

În scrierea unei definiții de macro cu parametri trebuie ca între numele macroului și paranteza deschisă să nu existe nici un caracter alb. În caz contrar, macroul se consideră fără parametri și paranteza inchisă, ce conține lista parametrilor, se consideră ca facind parte din textul de substituție.

Apelul unui macro constă din numele lui urmat de lista valorilor parametrilor inclusă între paranteze rotunde.

Parametrii de la apel (efectivi) se separă prin virgulă dacă sunt mai mulți. Ei se substituie parametrilor formalii în textul de substituție al macroului apelat și apoi textul rezultat se substituie apelului.

Corespondența dintre parametri formalii și valorile lor de la apel se realizează prin poziție.

Substituția apelului unui macro prin textul de substituție după ce, în prealabil, eventualii parametri formalii au fost înlocuiți cu valorile lor din apel, se numește *expandare*.

Expandarea poate fi anihilată cu ajutorul construcției `#undef`:

`#undef nume`

unde:

- | | |
|-------------|--------------------------|
| <i>nume</i> | - Este numele macroului. |
|-------------|--------------------------|

O definiție de macro este valabilă din punctul în care este scrisă și pînă la sfîrșitul textului sursă în care apare sau pînă la întărirea construcției `#undef` care

o anulează.

Apelul unui macro se expandează dacă acesta se află în textul sursă în zona în care este valabilă definiția lui. Evident, un apel de macro nu se expandează dacă se află într-un comentariu sau într-un șir de caractere.

Exemple:

- Macroul de mai jos definește maximul dintre valorile a două expresii. În acest scop se folosește o expresie condițională.
Macroul se definește în felul următor:

```
#define MAX(x,y) ((x) > (y) ? (x) : (y))
```

Dăm mai jos exemple de apeluri ale acestui macro:

a. int i,j,k;

...
k = MAX(i+j,i-j);

Apelul MAX(i+j,i-j) se expandează la preprocesare, iar la compilare se întâlnește instrucțiunea de atribuire de mai jos:

```
k = ((i+j) > (i-j) ? (i+j) : (i-j));
```

b. float a,b,c;

...
c = MAX(a,b);

Rezultatul expandării este:

```
c = ((a) > (b) ? (a) : (b));
```

- Mai jos, se definește un macro pentru calculul valorii absolute:

```
#define ABS(x) ((x) < 0 ? -(x) : (x))
```

Exemple de apeluri și expandări ale acestui macro:

a. int i,j,k;

...
k = ABS(i-j);

Se expandează astfel:

```
k = ((i-j) < 0 ? -(i-j) : (i-j));
```

b. double a,b,c,eps;

...
c = ABS(a-b) < eps;

Se expandează astfel:

```
c = ((a-b) < 0 ? -(a-b) : (a-b)) < eps;
```

- Macrouile de mai jos se utilizează la transformarea literelor mici în litere mari și invers:

```
#define UPPER(c) ((c)-'a'+'A')
```

```
#define LOWER(c) ((c)-'A'+'a')
```

Exemple de apel:

```
putchar(UPPER(c));
```

Scrie o literă mare dacă valoarea lui c este codul ASCII al unei litere mici;

```
putchar(LOWER(c));
```

Scrie o literă mică dacă valoarea lui c este codul ASCII al unei litere mari.

- Macroul de mai jos permite permutarea a două valori de tip *int*:

```
#define IPERM(X,Y) (\n    int t;\n    t = X;\n    X = Y;\n    Y = t;\n)
```

Apelul:

```
IPERM (a,b)
```

se expandează astfel:

```
{\n    int t;\n\n    t = a;\n    a = b;\n    b = t;\n}
```

Potem realiza un macro mai general de permutare care să fie valabil pentru date numerice de orice tip predeterminat:

```
#define PERM(TIP,X,Y) (\n    TIP t;\n    t = X;\n    X = Y;\n    Y = t;\n)
```

Apelul

```
PERM(int,a,b)
```

se expandează la fel ca și IPERM.

Pentru a permuta date flotante vom folosi apelul:

```
PERM(float,a,b)
```

Se observă o analogie între funcții și macrou. Ambele au o definiție și unul sau mai multe apeluri. De asemenea, atât în definiția de funcție cât și în cea de macro se pot utiliza parametri formală. În ambele cazuri, valorile parametrilor

formali se definesc la apeluri. Există însă o diferență esențială între cele două moduri de definire a valorilor parametrilor.

În cazul macrourilor, valoarea parametrului formal este succesiunea de caractere corespunzătoare lui din apel. Această succesiune de caractere se substituie în locul parametrului formal peste tot unde acesta apare în textul de substituție al macroului. De exemplu, la apelul PERM(int,a,b) de mai sus, parametrul formal TIP se substituie prin succesiunea de caractere *int* în textul de substituție și în felul acesta se obține declarația:

```
int t;
```

În cazul funcțiilor, parametrile formale li se atribuie valorile parametrilor de la apel care sunt date de tipuri predefinite sau adrese de zone de memorie.

O altă diferență esențială constă în aceea că, în timp ce apelul unei funcții înseamnă un salt la o zonă de memorie în care se păstrează instrucțiunile în format executabil corespunzător funcției respective, în cazul macrourilor apelul unui macro se înlocuiește chiar cu instrucțiunile rezultate în urma compilării, expandării macroului respectiv.

De exemplu, fie funcția:

```
int abs(int x) /* returneaza valoarea absolută a lui x */
{
    return x < 0 ? -x : x;
}
```

și apelul ei:

```
int i,j;
...
j = abs(i);
...
```

La apelul funcției se realizează un salt la zona în care se află rezultatul compilării funcției *abs* definită mai sus.

După executarea instrucțiunilor corespunzătoare funcției *abs*, se revine în punctul de apel și se atribuie lui *j* valoarea returnată de funcție (valoarea absolută a lui *i*).

Înlocuind definiția funcției *abs* prin definiția de macro de mai jos:

```
#define abs(x) ((x) < 0 ? -(x) : (x))
```

același apel se expandează astfel:

```
j = ((i) < 0 ? -(i) : (i));
```

În acest caz, apelul macroului se înlocuiește prin instrucțiunile care calculează valoarea absolută a lui *i*.

Cu alte cuvinte, în cazul macrourilor instrucțiunile definite de ele se generează și apoi se compilează direct în locurile apelurilor, adică în poziția în care este nevoie de ele. În cazul funcțiilor, instrucțiunile definite de corpul unei

funcții se păstrează într-o zonă de memorie a cărei poziție nu are nimic comun cu apelurile funcției. La fiecare apel al funcției se realizează un salt la zona respectivă, iar după terminarea execuției funcției se revine la punctul de după apel.

Având în vedere cele de mai sus, se pune problema existenței avantajelor la utilizarea macrourilor în locul funcțiilor.

Utilizarea macrourilor este avantajoasă pentru procese de calcul foarte simple, cum sunt exemplele de mai sus: maximul dintre două valori numerice, valoarea absolută, permutarea valorilor a două variabile etc.

Într-adevăr, prin utilizarea macrourilor, în locul funcțiilor, se elimină apelurile funcțiilor. Amintim că la apelul unei funcții se face nu numai salt la zona de memorie corespunzătoare funcției apelate, ci se alocă pe stivă parametri formali și variabilele automate ale funcției respective. De asemenea, la revenirea din funcție se face curățirea stivei. Toate aceste activități pot fi mai complexe decât însuși procesul de calcul realizat prin funcție. De aceea, în acest caz, se preferă utilizarea macrourilor în locul funcțiilor.

Prin utilizarea macrourilor nu se mai face apel și nici alocări de parametri pe stivă și respectiv dezalocarea lor. Apelul macroului pur și simplu se înlocuiește prin instrucțiunile care realizează procesul de calcul exprimat prin expandarea macroului. În acest caz se obișnuiește să se spună că macroul definește o funcție generată *în linie* (in-line).

Pentru procese de calcul mai complexe nu se justifică utilizarea macrourilor, deoarece substituirea fiecărui apel prin textul de substituție al macroului apelat, va conduce la un consum mare de memorie.

În limbajul C++ se pot defini funcții *in-line*. Acestea, ca și macrourile, se generează pe locul apelurilor lor. De aceea, o funcție *in-line* trebuie să corespundă unui calcul simplu care se poate realiza cu un număr mic de instrucțiuni (2-3 instrucțiuni). În caz contrar, se poate ajunge la un consum mare de memorie.

Diferența dintre funcțiile *in-line* și macrouri constă în faptul că, la apelul funcțiilor *in-line* se fac controale asupra tipului parametrilor utilizati la apel, în timp ce în cazul macrourilor astfel de controale nu mai au loc.

15.2. Compilare condiționată

Compilarea condiționată permite să se aleagă, dintr-un text general, părțile care să se compileze împreună. Acest lucru este util pentru programe care au părți comune sau care depind de calculator sau de versiunea compilatorului. În felul acesta, în funcție de anumiți parametri, se pot alege părțile din textul sursă care urmează să fie compilate împreună.

Compilarea condiționată se realizează folosind construcțiile (directive) preprocesorului:

#if, #ifdef și #ifndef.

Construcția #if are două formate:

```
#if expr  
    text  
#endif  
și  
#if expr  
    text1  
#else  
    text2  
#endif
```

unde:

- expr* - Este o expresie constantă (valoarea ei poate fi evaluată de preprocesor la intilnirea ei).
- text, text1 și text2* - Sunt texte sursă care sunt supuse compilării în funcție de valoarea expresiei *expr*.

În primul format, dacă *expr* are valoarea adevărat (este diferită de zero), atunci *text* se supune preprocesării și în consecință și compilării, apoi se continuă cu textul sursă aflat după *#endif*. În caz contrar (*expr* are valoarea zero), se continuă cu textul aflat după *#endif*.

În formatul al doilea, dacă *expr* are valoarea adevărat, atunci se supune preprocesării *text1* și deci și compilării apoi se continuă cu textul care urmează după *#endif*. În caz contrar (*expr* are valoarea zero) se supune preprocesării *text2* și deci și compilării, apoi se continuă cu textul aflat după *#endif*.

Directivele #if/def și #ifndef/def se utilizează în mod analog.

În cazul directivei #ifdef se utilizează următoarele formate:

```
#ifdef nume  
    text  
#endif  
și  
#ifdef nume  
    text1  
#else  
    text2  
#endif
```

În primul format, *text* se supune preprocesării și deci și compilării, dacă *nume* este definit în momentul intilnirii directivei #ifdef de către preprocesor.

În formatul al doilea *text1* se preprocesează și apoi compilează numai dacă *nume* este definit în momentul intilnirii directivei #ifdef de către preprocesor. În caz contrar (*nume* nu este în prealabil definit) se preprocesează și apoi compilează *text2*.

Directivea #ifndef are o acțiune opusă directivei #ifdef. Deci, *text* din formatul:

```
#ifndef nume  
    text  
#endif
```

se preprocesează și apoi compilează dacă *nume* nu este definit în momentul intilnirii directivei #ifndef de către preprocesor.

În mod analog, formatul:

```
#ifndef nume  
    text1  
#else  
    text2  
#endif
```

realizează preprocesarea și apoi compilarea lui *text1* dacă *nume* nu este definit în momentul intilnirii directivei #ifndef de către preprocesor. În caz contrar (*nume* este în prealabil definit) se preprocesează și apoi compilează *text2*.

Exemplu:

Un program se compune din două fișiere *fis1.cpp* și *fis2.cpp*. Ambele fișiere conțin apeluri la funcții care au prototipurile în fișierul *stdio.h*.

Întrucit cele două fișiere au fost editate de programatori diferiți, ele au fost la început compilate separat pentru a se elibera erorile sintactice. De aceea, fiecare fișier conține la începutul lui directiva:

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

În final, cele două fișiere se compilează împreună inserind directiva:

```
#include "fis1.cpp"
```

în fișierul *fis2.cpp* sau directiva:

```
#include "fis2.cpp"
```

în fișierul *fis1.cpp*.

În acest caz fișierul *stdio.h* se va include (preprocesat) de două ori. Pentru a evita acest lucru se pot folosi directivele compilării condiționate, ca mai jos.

Se inserează la începutul fiecărui fișier secvența:

```
#ifndef __STDIO_H  
#include <stdio.h>  
#define __STDIO_H  
#endif
```

Această secvență de directive realizează următoarele:

- dacă numele __STDIO_H nu este definit, atunci:
- se include fișierul *stdio.h*;

- se definește numele `_STDIO_H`.
- în caz contrar (numele `_STDIO_H` este definit) se trece la preprocesarea textului aflat după `#endif`.

Se observă că această secvență permite includerea fișierului `stdio.h` numai în cazul în care `_STDIO_H` este nedefinit. Totodată, după includerea lui `stdio.h` se definește variabila `_STDIO_H`, ceea ce permite suprimarea includerilor ulterioare ale fișierului `stdio.h`.

Direcțiva `#define` de mai sus nu atribuie o valoare de substituție pentru numele `_STDIO_H` deoarece acest nume nu este folosit la substituție ci numai în direcțiva `#ifndef`.

Numele `_STDIO_H` a fost ales ca să fie sugestiv și pe cît posibil să nu coincidă cu unul care să aibă alte utilizări.

Exerciții:

15.1 Să se definească un macro care apoi să fie apelat la evaluarea expresiilor de mai jos:

$$i = (|a| + |b|) / (1 + |a - b|)$$

unde:

i, a, b - Sint de tip *int*.

$$z = (|x+y| - |x-y|) / (1 + |x+y||x-y|)$$

unde:

x, y, z - Sint de tip *double*.

Definiția macroului și apelurile lui pentru evaluarea acestor expresii intră în compunerea programului de mai jos.

Valorile variabilelor a, b, x și y se citesc de la intrarea standard.

PROGRAMUL BXV1

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

#define ABS(X) ((X)<0 ? -(X) : (X))

#include "BVI1I2.CPP" /* pcit_int */
#include "BVI1I3.CPP" /* pcit_int_lim */

int pcit_double(char *p,double *d);

main() /* citeste valorile variabilelor a, b, x si y, calculeaza expresiile de
mai jos si afiseaza valorile variabilelor i si z:
    i=(|a|+|b|)/(1+|a-b|);
    z=(|x+y|-|x-y|)/(1+|x+y||x-y|). */
{
    int a,b,i;
    double x,y,z;
```

```
    double s,d;
    char er[] = "s-a tastat EOF\n";
    /* citeste valoarea lui a */
    if(pcit_int("a",-32768,32767,&a)==0){
        printf(er);
        exit(1);
    }

    /* citeste valoarea lui b */
    if(pcit_int("b",-32768,32767,&b)==0){
        printf(er);
        exit(1);
    }

    /* citeste valoarea lui x */
    if(pcit_double("x",&x)==0){
        printf(er);
        exit(1);
    }

    /* citeste valoarea lui y */
    if(pcit_double("y",&y)==0){
        printf(er);
        exit(1);
    }

    /* calculul valorii lui i */
    i = (ABS(a)+ABS(b)) / (1+ABS(a-b));

    /* calculul valorii lui z */
    s = ABS(x+y);
    d = ABS(x-y);
    z = (s-d) / (1+s*d);

    /* afisare rezultate */
    printf("a = %d\ntb = %d\n",a,b);
    printf("(ABS(a)+ABS(b))/(1+ABS(a-b))=%d\n",i);
    printf("x = %f\ty = %f\n",x,y);
    printf("(ABS(x+y)-ABS(x-y))/(1+ABS(x+y)*ABS(x-y))=%g\n",z);

} /* sfirsit main */

int pcit_double(char *p,double *d)
/* - afiseaza textul spre care pointeaza p;
   - citeste un numar si-l pastreaza in zona spre care pointeaza d;
   - returneaza zero la intalnirea sfirșitului de fisier si unu in caz contrar. */
{
    char t[255];
    double f;

    for(;;){
        printf("%s",p);
        if(gets(t)==0) return 0;
        if(sscanf(t,"%lf",&f)==1) break;
    }
}
```

```

        printf("nu s-a tastat un numar\n");
    }
    *d=f;
    return 1;
}

```

15.2 Să se scrie un program în care să se definească un macro pentru calculul unui polinom de gradul doi și care să fie apelat pentru evaluarea expresiilor:

$$\text{pol1} = 3x^2 + 7x - 8;$$

$$\text{pol2} = x^2 - 3.5x + 1.2.$$

Programul citește valoarea lui x și afișează valorile variabilelor pol1 și pol2 .

PROGRAMUL BXV2

```

#include <stdio.h>
#include <stdlib.h>

#define POL(A,B,C) ((A)*x*x+(B)*x+(C))

main() /* citeste pe x, calculeaza si afiseaza valorile expresiilor:
         3x^2+7x-8
         si
         x^2-3.5x+1.2. */
{
    double x;
    char t[255];

    for(;;){
        printf("x=");
        if(gets(t)==0){
            printf("s-a tastat EOF\n");
            exit(1);
        }
        if(sscanf(t,"%lf",&x)==1) break;
        printf("nu s-a tastat un numar\n");
    }
    printf("x = %f\t 3x^2+7x-8 = %g\n",x,POL(3,7,-8));
    printf("x = %f\t x^2-3.5x+1.2 = %g\n",x,POL(1,-3.5,1.2));
}

```

16. INTRĂRI/IEȘIRI

Limbajul C nu dispune de instrucțiuni de intrare/ieșire. Aceste operații se realizează prin intermediul unor *funcții* din biblioteca standard a limbajului. Aceste funcții pot fi aplicate în mod eficient la o gamă largă de aplicații. De asemenea, ele asigură o portabilitate bună a programelor, fiind implementate într-o formă compatibilă pe diferite sisteme de operare. Aceasta însă nu înseamnă că ele nu au și facilități suplimentare pe anumite sisteme, cum ar fi de exemplu în cazul mediului de programare Turbo C.

În acest capitol ne vom referi la funcțiile din biblioteca standard I/O care au o utilizare frecventă în diferite aplicații. Deși ele au un caracter general, în capitolul de față ne vom referi la facilitățile oferite de funcțiile existente în biblioteca compilatorului Turbo C.

Majoritatea operațiilor de intrare/ieșire se realizează în ipoteza că datele sunt organizate în *fișiere*.

În general, prin *fișier* înțelegem o colecție ordonată de elemente, numite *înregistrări*, care sunt păstrate pe diferite suporturi externe. Cele mai utilizate suporturi pentru fișiere sunt cele magnetice. Acestea, de obicei, sunt *discuri și benzi magnetice*. Ele se numesc suporturi *reutilizabile*, deoarece zona utilizată pentru a păstra înregistrările unui fișier poate fi ulterior reutilizată pentru a păstra înregistrările altui fișier.

Datele introduse de la tastatura unui terminal se consideră că formează un *fișier de intrare*. Înregistrarea în acest caz, de obicei, este formată din datele tastate la terminal pe un rind deci caracterul de rind nou (*newline*) este *terminator de înregistrare*. În mod analog, datele care se afișează pe terminal formează un *fișier de ieșire*. Înregistrarea, și în acest caz, poate fi formată din caracterele unui rind.

Datele care se scriu la imprimantă formează și ele un fișier de ieșire. Un rind scris la imprimantă este o înregistrare.

Un fișier are o înregistrare care marchează *sfîrșitul de fișier*. În cazul fișierelor de intrare de la tastatură sfîrșitul de fișier se generează prin sevența:

<Ctrl>-Z

De obicei, prin *intrare standard* se înțelege tastatura terminalului de la care s-a lansat programul. În mod analog, *iesirea standard* este ecranul același terminal.

Prelucrarea fișierelor implică un număr de *operații* specifice acestora. În primul rind, orice fișier, înainte de a fi prelucrat, trebuie *deschis*. De asemenea, la terminarea prelucrării unui fișier, acesta trebuie *inchis*.

Alte operații frecvente în prelucrarea fișierelor sunt:

- crearea unui fișier;
- citirea (consultarea) înregistrărilor unui fișier;

- actualizarea unui fișier;
- adăugarea de înregistrari într-un fișier;
- poziționarea într-un fișier;
- ștergerea unui fișier.

Toate aceste operații pot fi realizate prin funcții standard existente în biblioteca limbajului C.

Tratarea fișierelor se poate face la două niveluri. Primul nivel face apel direct la sistemul de operare. Acesta este *nivelul inferior* de prelucrare a fișierelor.

Celalalt nivel se realizează prin intermediul unor proceduri specializate de prelucrare a fișierelor care utilizează structuri speciale de tip FILE (fișier). Aceasta este *nivelul superior* de prelucrare a fișierelor.

Pînă în prezent am utilizat funcții pentru prelucrarea la nivel superior a fișierelor standard de intrare/ieșire la terminalul de la care s-a lansat programul.

Așa de exemplu, pentru a citi o înregistrare de la intrarea standard am utilizat funcția *gets*. Pentru a afișa înregistrarea la terminalul standard am folosit funcția *puts*. Alte funcții utilizate, care permit și conversii din format intern în cel extern și invers, sunt funcțiile *scanf* și *printf*. Prima permite citirea datelor de la intrarea standard sub controlul formatorilor. Cea de a doua funcție permite afișarea la ieșirea standard sub controlul formatorilor.

ACESTE FUNCȚII AU PROTOTIPIRILE IN FIȘIERUL *stdio.h*.

Pentru citirea de caractere de la intrarea standard am folosit macroul *getchar* definit în fișierul *stdio.h*. În mod analog, pentru a afișa caractere la ieșirea standard am folosit macroul *putchar* definit tot în fișierul *stdio.h*. În continuare se indică funcții standard analoge din biblioteca C care pot fi utilizate pentru prelucrarea altor fișiere decit cele de la intrarea sau ieșirea standard.

16.1. Nivelul inferior de prelucrare a fișierelor

16.1.1. Deschiderea unui fișier

Așa cum s-a spus mai sus, orice fișier înainte de a fi prelucrat trebuie *deschis*. Deschiderea unui fișier existent se realizează cu ajutorul funcției *open*. La revenirea din funcția *open* se returnează așa numitul *descriptor de fișier*. Aceasta este număr întreg nenegativ. El identifică în continuare fișierul respectiv în toate operațiile realizate asupra lui.

În forma cea mai simplă, funcția *open* se apelează printr-o expresie de atribuire de forma:

df = open(...);

unde:

df Este o variabilă de tip *int*.

Funcția *open* are prototipul:

int open(const char *cale, int acces);

unde:

cale

- Este un pointer spre un sir de caractere care definește calea spre fișierul care se deschide.

acces

- Este o variabilă de tip întreg care poate avea una din valorile următoare:

- **O_RDONLY** - fișierul se deschide numai în citire (read-only);
- **O_WRONLY** - fișierul se deschide numai în scriere (write_only);
- **O_RDWR** - fișierul se deschide în citire/scriere;
- **O_APPEND** - fișierul se deschide pentru adăugarea de înregistrări la sfîrșitul lui;
- **O_BINARY** - fișierul se prelucrează binar;
- **O_TEXT** - fișierul este de tip text (se prelucrează pe caractere).

ACESTE VALORI SE POT COMBINA CU AJUTORUL OPERATORULUI SAU LOGIC PE BIȚI (""). DE EXEMPLU:

O_RDWR|O_BINARY

INSEAMNĂ CĂ FIȘIERUL ESTE DESCHIS ÎN CITIRE/SCRIERE BINARĂ.

ÎN MOD IMPLICIT SE CONSIDERĂ CĂ UN FIȘIER ESTE DE TIP TEXT, DECI O_TEXT SE poate omite.

UTILIZAREA FUNCȚIEI *open* IMPLICĂ INCLUDEREA FIȘIERELOR *io.h* și *fcntl.h*:

#include <io.h>

și

#include <fcntl.h>

CALEA SPRE FIȘIER TREBUIE SĂ RESPECTE CONVENTIILE SISTEMULUI DE OPERARE MS-DOS. ÎN CEA MAI SIMPLĂ FORMĂ EA ESTE UN SIR DE CARACTERE CARE DEFINEȘTE NUMELE FIȘIERULUI, URMAT DE PUNCT ȘI EXTENSA FIȘIERULUI. ACEASTA PRESUPUNE CĂ FIȘIERUL RESPECTIV SE AFLĂ ÎN DIRECTORUL CURENT.

ÎN CAZUL ÎN CARE FIȘIERUL NU ESTE ÎN DIRECTORUL CURENT, NUMELE ESTE PRECEDAT DE O CONSTRUCȚIE DE FORMA:

litera:\nume_1\ume_2\...\nume_k

unde:

litera

- Definește discul (în general A, B pentru disc flexibil, C, D, ... pentru disc fix).

ume_i

- Este nume de subdirector, i = 1,2,...,k.

Deoarece *calea* se include între ghilimele, caracterul '\' se dublcază.

Deschiderea unui fișier nu reușește dacă unul din parametri este eronat. În acest caz funcția *open* returnează valoarea -1. Un caz frecvent de eroare este acela în care se încercă deschiderea unui fișier inexistent.

Exemplu:

```
1. char nfis[] = "fis1.dat";
   int df;
   df = open(nfis,O_RDONLY);
```

Fișierul *fis1.dat* din directorul curent se deschide în citire. Funcția returnează valoarea -1 dacă nu există un astfel de fișier în directorul curent.

```
2. int d;
   d = open("A:\\JOC\\BIO.C",O_RDWR);
```

Fișierul *BIO.C* din directorul *JOC* de pe discul A se deschide în citire scriere. În acest caz sirul de caractere aflat pe locul primului parametru al apelului funcției *open* se păstrează într-o zonă de memorie specială rezervată sirurilor de caractere. Adresa de început a acestei zone de memorie se atribuie primului parametru al funcției *open* în momentul apelului.

```
3. int i;
   i=open("c:\\borlandc\\include\\text\\text.h",O_APPEND);
   Fișierul text.h se deschide pentru adăugare de înregistrare.
```

```
4. int c;
   c=open("fis.c",O_WRONLY);
```

În directorul curent trebuie să existe fișierul *fis.c*. Prin deschiderea lui în scriere, acesta se crează din nou, vechiul conținut al fișierului pierzindu-se.

Pentru a crea un fișier nou se utilizează funcția *creat* în locul funcției *open*. Aceasta are prototipul:

```
int creat(const char *calea, int mod);
```

unde:

calea - Este un pointer spre un sir de caractere care definește calea spre fișierul care se deschide în creare.

mod - Este un întreg care poate fi definit folosind constantele simbolice de mai jos:

- **S_IREAD** - proprietarul poate citi fișierul;
- **S_IWRITE** - proprietarul poate scrie în fișier;
- **S_IEXE** - proprietarul poate executa programul conținut în fișierul respectiv.

Acești indicatori pot și combinați folosind caracterul '|'. De exemplu, pentru citire scriere se va folosi:

S_IREAD|S_IWRITE

Funcția *creat*, ca și funcția *open*, returnează descriptorul de fișier sau -1 în caz de eroare.

Utilizarea funcției *creat* implică includerile de fișiere:

```
#include <io.h>
```

și

```
#include <stat.h>
```

În cazul în care funcția *creat* se folosește pentru a deschide un fișier existent, atunci acesta se va șterge, urmând ca în locul lui să se creeze fișierul nou.

Menționăm că fișierele de intrare/ieșire standard se deschid automat la lansarea programului în execuție. De aceea, aceste fișiere nu se deschid prin program de către programator.

Descriptorul de fișier reprezentat de intrarea standard are valoarea zero, iar cel reprezentat de ieșirea standard are valoarea unu.

Sistemul oferă utilizatorului încă o ieșire standard care, de obicei, este destinată pentru afișarea *erorilor*.

Fișierul corespunzător acestei ieșiri standard are descriptorul de fișier egal cu doi. Nici acest fișier nu se deschide prin program de către utilizator.

16.1.2. Citirea dintr-un fișier

Operația de citire a unei înregistrări dintr-un fișier deschis cu ajutorul funcției *open* se realizează folosind funcția *read*. Aceasta returnează numărul de octeți citiți din fișier sau -1 la eroare.

Prototipul funcției este:

```
int read(int df, void *buf, unsigned lung);
```

unde:

df - Este descriptorul de fișier returnat de funcția *open* la deschiderea fișierul respectiv.

buf - Este pointerul spre zona de memorie în care se păstrează înregistrarea citită din fișier.

lung - Este lungimea în octeți a înregistrării citite.

La fiecare apel al funcției *read* se citește înregistrarea curentă. Astfel, la primul apel se citește prima înregistrare din fișier, la al doilea apel a doua înregistrare și aşa mai departe.

Ordinea înregistrărilor este cea definită la crearea fișierului și eventual la adăugarea ulterioară de înregistrări noi.

La un apel al funcției *read* se citește cel mult *lung* octeți. La înălțirea sfîrșitului de fișier nu se citește nimic și de aceea funcția *read* returnează zero în acest caz.

Dacă lung=1, atunci la apelul lui *read* se citește un singur octet din fișier. Acest lucru nu este avantajos mai ales atunci cind se citesc înregistrările de pe disc. În acest caz o valoare utilizată frecvent pentru *lung* este 512.

Funcția *read* poate fi folosită pentru a căuta caracterul de la intrarea standard. În acest caz descriptorul de fișier are valoarea zero la apelul funcției *read*.

Utilizarea funcției *read* implică includerea fișierului *io.h*.

16.1.3. Scrierea într-un fișier

Pentru a scrie o înregistrare într-un fișier folosim funcția *write*. Fișierul trebuie să fie în prealabil deschis cu ajutorul funcției *open* sau *creat*. Ea este asemănătoare cu funcția *read* și are același prototip. Diferența constă în aceea că realizează transferul datelor în sens invers, adică din zona de memorie spre care pointează cel de al doilea parametru al ei, în fișier.

Ea returnează numărul octetilor scriși în fișier, adică o valoare care coincide cu aceea a celui de al treilea parametru al ei. În cazul în care funcția returnează o altă valoare decit valoarea celui de al treilea parametru din apel, înseamnă că scrierea a fost eronată.

Funcția *write* poate fi folosită pentru a scrie la cele două ieșiri standard, folosind descriptori de fișier de valoare 1 sau 2. Utilizarea funcției *write* implică includerea fișierului *io.h*.

16.1.4. Poziționarea într-un fișier

Operațiile de citire/scriere se execută *secvențial*. Aceasta înseamnă că la fiecare apel al funcției *read* sau *write* se citește înregistrarea curentă, respectiv se scrie înregistrarea în poziția curentă de pe suportul fișierului.

Înregistrările se scriu una după alta pe suportul fișierului. La citire ele se citesc în același ordine în care au fost scrise la crearea fișierului. Acest mod de scriere și acces la înregistrările fișierului se numește *acces secvențial*. Accesul secvențial este util atunci cind dorim să prelucrăm toate înregistrările fișierului, una după alta.

În practică apar însă situații în care noi dorim să scriem și să citim înregistrările într-o ordine diferită de cea secvențială, eventual chiar aleatoare. În acest caz se spune că *accesul* la fișier este *aleator*. Pentru a realiza un acces aleator este nevoie să ne putem poziționa oriunde în fișierul respectiv. O astfel de poziționare este posibilă pe suporturile de tip disc magnetic și se realizează folosind funcția *Iseek*.

Ea are prototipul:

long Iseek (int df, long deplasament, int origine);

unde:

df

- Este descriptorul de fișier.

- | | |
|--------------------|---|
| <i>deplasament</i> | - Definește numărul de octeți peste care se va deplasa capul de citire/scriere al discului. |
| <i>origine</i> | - Are una din valorile: <ul style="list-style-type: none">• 0 - deplasamentul se consideră de la începutul fișierului;• 1 - deplasamentul se consideră din poziția curentă a capului de citire/scriere;• 2 - deplasamentul se consideră de la sfîrșitul fișierului. |

Prin apelul funcției *Iseek* nu se realizează nici un fel de transfer de informație, ci numai poziționarea în fișier. Operația următoare realizată prin apelul funcției *read* sau *write* se va realiza din această nouă poziție a capului de citire/scriere.

Funcția returnează poziția capului de citire/scriere în număr de octeți, față de începutul fișierului. În caz de eroare, se returnează -1L.

Utilizarea funcției *Iseek* implică includerea fișierului *io.h*.

Apelul:

Iseek(df,0L,2)

permite poziționarea capului de citire/scriere la sfîrșitul fișierului definit de valoarea descriptorului de fișier *df*.

În mod analog, apelul:

Iseek(df,0L,0)

permite poziționarea capului de citire/scriere la începutul fișierului definit de *df*.

16.1.5. Închiderea unui fișier

După terminarea prelucrării unui fișier acesta trebuie închis. Acest lucru se realizează automat dacă programul se termină prin apelul funcției *exit*.

Programatorul poate închide un fișier folosind funcția *close*. Se recomandă ca un fișier să se închidă de îndată ce s-a terminat prelucrarea lui. Aceasta din cauză că numărul fișierelor ce pot fi deschise simultan este limitat. Această limită poate fi definită de utilizator. De obicei, ea are valoarea 20.

Mentionăm că fișierele corespunzătoare intrărilor și ieșirilor standard nu se închid de către programator.

Funcția *close* are prototipul:

int close (int df);

unde:

df

- Este descriptorul fișierului care se închide.

La o închidere normală a fișierului, funcția returnează valoarea zero. În caz de eroare se returnează -1.

Utilizarea funcției *close* implică includerea fișierului *io.h*.

Exerciții:

16.1 Să se scrie un program care copiază intrarea standard la ieșirea standard folosind o zonă tampon de 70 de caractere.

PROGRAMUL BXVI1

```
#include <io.h>

#define LZT 70

main() /* copiaza intrarea standard la ieșirea standard */
{
    char zt[LZT];
    int lung;

    while((lung=read(0,zt,LZT))>0) write(1,zt,lung);
}
```

16.2 Să se scrie un program care citește un sir de numere de la intrarea standard și crează două fișiere *fis1.dat* și *fis2.dat*, primul conține numerele de ordin impar citite de la intrarea standard - primul, al treilea, al cincilea etc., iar al doilea conține numerele de ordin par citite de la aceeași intrare - al doilea, al patrulea etc. Apoi se listează, la ieșirea standard, cele două fișiere în ordinea *fis1.dat*, *fis2.dat* cite un număr pe un rind în formatul:

număr de ordine număr

PROGRAMUL BXVI2

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <iolib.h>
#include <conio.h>

main() /* - citește un sir de numere și creaază fișierele fis1.dat cu numerele de ordin impar
         și fis2.dat cu numerele de ordin par;
         - apoi se listează fișierele. */
{
    union unr {
        float nr;
        char tnr[sizeof(float)];
    };
    union unr nrcit;
    int df1,df2;
    int i,j;

    /* se deschid fișierele fis1.dat și fis2.dat în creare cu acces în citire/scrisoare */
    if((df1=creat("fis1.dat",S_IWRITE|S_IREAD))==-1){
        printf("nu se poate deschide fișierul fis1.dat\n");
        exit(1);
    }
}
```

```
    if((df2=creat("fis2.dat",S_IWRITE|S_IREAD))==-1){
        printf("nu se poate deschide fișierul fis2.dat\n");
        exit(1);
    }

    /* se citește sirul de numere și se pastrează în cele două fișiere */
    j=1;
    printf("tastati numerele separate prin caractere albe\n");
    while ((scanf("%f",&nrcit.nr))==1){
        if(j&1) /* este impar */
            if((write(df1,nrcit.tnr,sizeof(float)))!=
               sizeof(float)){
                printf("eroare la crearea fișierului fis1.dat\n");
                exit(1);
            }
        else /* este par */
            if((write(df2,nrcit.tnr,sizeof(float)))!=
               sizeof(float)){
                printf("eroare la crearea fișierului fis2.dat\n");
                exit(1);
            }
        j++;
    }

    /* s-a terminat citirea numerelor și scrierea lor în fișiere */
    if(close(df1)<0) {
        printf("eroare la inchiderea fișierului fis1.dat\n");
        exit(1);
    }
    if(close(df2)<0) {
        printf("eroare la inchiderea fișierului fis2.dat\n");
        exit(1);
    }

    /* se redeschide fișierul fis1.dat și se listează la ieșirea standard */
    if((df1=open("fis1.dat",O_RDONLY))==-1){
        printf("nu se poate deschide fișierul fis1.dat în citire\n");
        exit(1);
    }
    j=1;
    while((i=read(df1,nrcit.tnr,sizeof(float)))>0){
        printf("%6d\t%g\n",j,nrcit.nr);
        if(j&46==0){
            printf("Actionati o tasta pentru a continua\n");
            getch();
        }
        j += 2;
    }
    if(i<0){
        printf("eroare la citirea din fis1.dat\n");
    }
}
```

```

        exit(1);
    }
    close(df1);
    printf("Actionati o tasta pentru a continua \n\n\n");
    getch();

/* se redeschide fisierul fis2.dat si se listeaza la iesirea standard */
if((df2=open("fis2.dat",O_RDONLY))==-1){
    printf("nu se poate deschide fisierul\\
          fis2.dat in citire\n");
    exit(1);
}
j=0;
while((i=read(df1,nrcit.tnr,sizeof(float)))>0){
    printf("%6d\t%g\n",j,nrcit.nr);
    if(j%46==0){
        printf("Actionati o tasta pentru a continua\n");
        getch();
    }
    j += 2;
}
if(i<0){
    printf("eroare la citirea din fis2.dat\n");
    exit(1);
}
close(df2);
}

```

16.3 Să se rescrie programul din exercițiul precedent folosind un singur fișier *fis.dat*.

În acest caz se păstrează, în fișierul *fis.dat*, toate datele citite de la intrarea standard. Apoi fișierul *fis.dat* se parcurge de două ori, din două în două înregistrări, întii începând cu prima înregistrare, apoi începând cu a două.

PROGRAMUL BXVI3

```

#include <stdio.h>
#include <stdlib.h>
#include <sys\stat.h>
#include <fcntl.h>
#include <io.h>
#include <conio.h>

main() /* - citește un sir de numere și crează fisierul fis.dat;
         - apoi listează fisierul fis.dat din două înregistrări;
         - întii se începe cu prima înregistrare, apoi cu a două. */
{
    union unr {
        float nr;
        char tnr[sizeof(float)];
    };
    union unr nrcit;
    int df;

```

```

    int i,j,k;

/* se deschide fisierul fis.dat în creare cu acces în citire/scrivere */
if((df=creat("fis.dat",S_IWRITE|S_IREAD))==-1){
    printf("nu se poate deschide fisierul fis.dat\\
          in creare\n");
    exit(1);
}

/* se citește sirul de numere și se păstrează în fis.dat */
printf("tastati numerele separate prin caractere albe\n");
while ((scanf("%f",&nrcit.nr))!=1)
    if((write(df,nrcit.tnr,sizeof(float))) != sizeof(float)){
        printf("eroare la crearea fisierului fis.dat\n");
        exit(1);
    }

/* s-a terminat citirea numerelor și scrierea lor în fisier */
if(close(df)<0) {
    printf("eroare la inchiderea fisierului fis.dat\n");
    exit(1);
}

/* se redeschide fisierul fis.dat și se listează la iesirea standard */
if((df=open("fis.dat",O_RDONLY))==-1){
    printf("nu se poate deschide fisierul fis.dat\\
          in citire\n");
    exit(1);
}
j=1;
for(k=0;k<2;k++){
    while((i=read(df,nrcit.tnr,sizeof(float)))>0){
        printf("%6d\t%g\n",j,nrcit.nr);
        if(j%46==0){
            printf("Actionati o tasta pentru a continua\n");
            getch();
        }
        j += 2;
    }
    /* avans peste o înregistrare */
    if(lseek(df,(long)sizeof(float),1)==-1L)
        break; /* s-a ajuns la sfârșitul fisierului fis.dat */
    j += 2;
} /* sfârșit while */
if(i<0){
    printf("eroare la citirea din fis.dat\n");
    exit(1);
}

/* se trece la citirea a două: se face pozitionare pe înregistrarea a două a
   fisierului fis.dat */
if(k<1){
    if(lseek(df,(long)sizeof(float),0)==-1L){
        printf("nu se poate face pozitionare pe\\
              inregistrarea a două\n");
    }
}

```

```

        exit(1);
    }
    printf("Actionati o tasta pentru a continua\n\n");
    getch();
    j=2;
}
/* sfirsit for */
close(df);
}

```

16.2. Nivelul superior de prelucrare a fișierelor

La acest nivel operațiile de prelucrare a fișierelor se execută utilizându-se funcții specializate de gestiune a fișierelor. De asemenea, fiecărui fișier i se atașează o structură de tip FILE. Acest tip este definit în fișierul *stdio.h*. De asemenea, toate funcțiile din această clasă au prototipurile în fișierul *stdio.h*.

16.2.1. Deschiderea unui fișier

Pentru a deschide un fișier la acest nivel de prelucrare a fișierelor se utilizează funcția *fopen*.

Ea returnează un pointer spre tipul FILE (tipul fișier) sau pointerul nul în caz de eroare.

Prototipul funcției este următorul:

FILE *fopen (const char *calea, const char *mod);

unde:

calea

- Are aceeași semnificație ca și în cazul funcției *open*;
- Este un pointer spre un sir de caractere care definește modul de prelucrare al fișierului după deschidere. Acest sir de caractere se definește astfel:
 - "r" - deschidere în citire (read);
 - "w" - deschidere în scriere (write);
 - "a" - deschidere pentru adăugare (append);
 - "r+" - deschidere pentru modificare (citire/scriere);
 - "rb" - citire binară;
 - "wb" - scriere binară;
 - "r+b" - citire/scriere binară.

Cu ajutorul funcției *fopen* se poate deschide un fișier inexistent în modul *w* sau *a*. În acest caz, fișierul respectiv se consideră în creare.

Dacă se deschide un fișier existent în modul "w", atunci se va crea din nou fișierul respectiv și vechiul conținut al său se va pierde.

Deschiderea unui fișier în modul "a" permite adăugarea de înregistrări după

ultima înregistrare existentă în fișier.

Cu ajutorul funcțiilor din această clasa se pot trata intrările și ieșirile standard.

Fișierele corespunzătoare nu se deschid de către utilizator, deoarece ele sunt deschise automat la lansarea programului.

Pentru a utiliza aceste fișiere se vor folosi următorii pointeri spre tipul FILE:

stdin

- Pentru a citi de la intrarea standard.

stdout

- Pentru a afișa pe ecranul de la ieșirea standard.

stderr

- Pentru afișarea erorilor la ieșirea standard.

stdprn

- Ieșirea paralelă pe imprimantă.

stdaux

- Comunicație serială.

16.2.2. Prelucrarea pe caractere a unui fișier

Fișierele pot fi scrise și citite caracter cu caracter folosind două funcții simple:

putc

- Pentru scriere.

getc

- Pentru citire.

Funcția *putc* are prototipul:

int putc (int c, FILE *pf);

unde:

c

- Este codul ASCII al caracterului care se scrie în fișier.

pf

- Este un pointer spre tipul FILE a carui valoare a fost returnată de funcția *fopen* la deschiderea fișierului în care se face scrierea.

- În particular, *pf* poate fi unul din pointerii:

- **stdout** (ieșire standard);
- **stderr** (ieșire standard pentru eroare);
- **stdprn** (ieșire paralelă la imprimantă);
- **stdaux** (ieșire serială).

Funcția *putc* returnează valoarea lui *c* sau -1 la eroare.

Macroul *putchar* se definește cu ajutorul funcției *putc* astfel:

#define putchar(c) putc(c,stdout)

Această definiție se află în fișierul *stdio.h*.

Funcția *getc* are prototipul:

int getc(FILE *pf);

unde:

pf

- Este pointerul spre tipul FILE.

Valoarea lui *pf* este definită la apelul funcției *fopen*.

În particular, *pf* poate fi unul din pointerii:

- *stdin* (intrare standard);
- *stdaux* (intrare serială).

Funcția *getc* returnează codul ASCII al caracterului citit din fișier.

Macroul *getchar* se definește în fișierul *stdio.h* astfel:

```
#define getchar() getc(stdin)
```

Exerciții:

- 16.4 Să se scrie un program care copiază intrarea standard la ieșirea standard folosind funcțiile *getc* și *putc*.

PROGRAMUL BXVI4

```
#include <stdio.h>

main() /* copiază intrarea standard la ieșirea standard */
{
    int c;

    while((c=getc(stdin))!=EOF) putc(c,stdout);
}
```

Observație:

Programul poate fi scris și cu ajutorul macrourilor *getchar* și *putchar*. Nu avem decât să utilizăm:

getchar() în loc de *getc(stdin)*

și

putchar(c) în loc de *putc(c,stdout)*

În urma expandării acestor macrouri se obțin chiar apelurile lui *getc* și *putc* din programul de mai sus.

16.2.3. Închiderea unui fișier

După terminarea prelucrării unui fișier, acesta urmează să fie închis. Închiderea unui fișier se realizează cu ajutorul funcției *fclose* de prototip:

```
int fclose(FILE *pf);
```

unde:

pf - Este pointerul spre tipul FILE. Valoarea lui a fost definită prin funcția *fopen* la deschiderea fișierului.

Funcția returnează valorile:

- | | |
|-----------|-------------------------|
| <i>0</i> | - La închidere normală. |
| <i>-1</i> | - La eroare. |

Exerciții:

- 16.5 Să se scrie un program care listează la imprimanta paralelă un fișier a cărui nume este argumentul din linia de comandă. În cazul în care în linia de comandă nu există un astfel de argument, se listează caracterele citite de la intrarea standard.

PROGRAMUL BXVI5

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

main(int argc,char *argv[])
/* listează la imprimanta paralelă un fișier a cărui nume este argumentul din linia de comandă;
   - dacă argumentul lipsește, se listează caracterele de la intrarea standard. */
{
    FILE *pf;
    int c;

    if(argc ==1 )
        /* nu există argumente */
        pf=stdin;
    else
        if(argc==2)
            /* se deschide fișierul a cărui nume se află în linia de comandă */
            if((pf=fopen(argv[1],"r"))==0){
                printf("nu se poate deschide fișierul\n%s\n",argv[1]);
                exit(1);
            }
        else
            printf("număr argumente eronat în linia de comandă\n");
            exit(1);
    while((c=getc(pf))!=EOF) putc(c,stdout);
    fclose(pf);
}
```

16.2.4. Intrări/ieșiri de siruri de caractere

Biblioteca standard a limbajului C conține funcțiile *fgets* și *fputs* care permit citirea, respectiv scrierea, înregistrărilor care sunt siruri de caractere.

Funcția *fgets* are prototipul:

```
char *fgets (char *s, int n, FILE *pf);
```

unde:

s - Este pointerul spre zona în care se păstrează caracterele citite din fișier.

n - Este dimensiunea în octeți a zonei în care se citesc

pf

caracterele din fișier.

- La un apel a lui *fgets* se citesc cel mult n-1 caractere.
- Este pointerul spre tipul FILE a căruia valoare s-a definit la deschiderea fișierului.

Citirea caracterelor se întrerupe la întâlnirea caracterului '\n' sau după citirea a cel mult n-1 caracter. În zona spre care pointează *s* se păstrează caracterul '\n' dacă acesta a fost citit din fișier, iar apoi se memorează caracterul nul ('\0').

În mod normal, funcția returnează valoarea pointerului *s*. La întâlnirea sfîrșitului de fișier funcția returnează valoarea zero.

Funcția *fputs* scrie un sir de caractere (succesiune de caractere terminată cu '\0') într-un fișier.

Ea are prototipul:

```
int fputs (const char *s, FILE *pf);
```

unde:

s - Este pointerul spre începutul zonei de memorie care conține sirul de caractere care se scrie în fișier.

pf - Este pointerul spre tipul FILE a căruia valoare a fost definită la deschiderea fișierului prin apelul lui *fopen*.

Funcția *fputs* returnează codul ASCII al ultimului caracter scris în fișier sau -1 la eroare.

Aceste două funcții sunt similare cu funcțiile *gets* și *puts* utilizate în multe din exercițiile din această carte.

16.2.5. Intrări/ieșiri cu format

Biblioteca standard a limbajului C conține funcții care permit realizarea operațiilor de intrare/ieșire cu format. În acest scop se pot utiliza funcțiile *fscanf* și *fprintf*.

Acestea sunt similare cu funcțiile *sscanf*, respectiv *sprintf*. Diferența dintre ele constă în faptul că *fscanf* și *fprintf* au ca prim parametru un pointer spre tipul FILE, iar *sscanf* și *sprintf* au ca prim parametru un pointer spre o zonă în care se păstrează caractere. De aici rezultă și utilizările celor două funcții. Astfel, funcția *fscanf* citește date dintr-un fișier și le convertește păstrând rezultatele acestor conversii în conformitate cu parametri existenți la apelul funcției, în timp ce *sscanf* realizează același lucru dar utilizând date din memorie. În mod analog, funcția *fprintf*, convertește date din format intern în format extern și apoi le scrie într-un fișier, spre deosebire de funcția *sprintf* care realizează aceleași conversii, dar rezultatele se păstrează în memorie.

Prototipul funcției *fscanf* este:

```
int fscanf (FILE *pf, const char *format,...);
```

unde:

pf

- Este un pointer spre tipul FILE a căruia valoare a fost definită prin apelul funcției *fopen*.
- Aceasta definește fișierul din care se face citirea.

Cealalti parametri sunt identici cu cei utilizati în funcția *scanf*.

Funcția *fscanf*, ca și funcțiile *scanf* și *sscanf*, returnează numărul cimpurilor citite corect din fișier. La întâlnirea sfîrșitului de fișier funcția returnează valoarea EOF, definită în fișierul *stdio.h*.

Funcția *fprintf* are prototipul:

```
int fprintf (FILE *pf, const char *format,...);
```

unde:

pf

- Este un pointer spre tipul FILE a căruia valoare a fost definită prin apelul funcției *fopen*.
- Aceasta definește fișierul în care se face scrierea.

Cealalti parametri sunt identici cu cei utilizati în funcția *printf*.

Funcția *fprintf*, ca și funcțiile *printf* și *sprintf*, returnează numărul caracterelor scrise în fișier sau -1 în caz de eroare.

16.2.6. Vidarea zonei tampon a unui fișier

Există situații în care programatorul dorește să videze zona tampon a unui fișier. În acest scop se poate utiliza funcția *fflush*. Ea are prototipul:

```
int fflush (FILE *pf);
```

unde:

pf

- Este pointerul spre tipul FILE care definește fișierul pentru care se videază zona tampon.

Dacă fișierul este deschis în scriere, atunci conținutul zonei tampon se scrie în fișierul respectiv.

Dacă fișierul este deschis în citire, caracterele necitite din zona tampon se pierd, deoarece după apelul funcției zona tampon devine goală.

Un exemplu de utilizare a acestei funcții este acela al recuperării după o eroare în citire.

Exemplu:

```
int i,n;
...
do {
    printf("tastati valoare lui n=");
    if((i=scanf("%d",&n))<1) break;
    printf("nu s-a tastat un intreg\n");
    if(i==EOF) /* s-a tastat sfîrșitul de fisier */
```

```

    exit(1);
fflush(stdin); /* se elibera caracterele necitite din zona
tampon atasata intrarii standard */
}while(1);

```

Funcția returnează valoarea -1 în caz de eroare și zero dacă vidarea se realizează fără erori.

16.2.7. Poziționarea într-un fișier

Biblioteca standard a limbajului C conține funcția *fseek* care permite deplasarea capului de citire/scrivere al discului în vederea prelucrării înregistrărilor fișierului într-o ordine diferita de cea secvențială.

Ea are prototipul:

```
int fseek(FILE *pf, long deplasament, int origine);
```

unde:

pf - Este pointerul spre tipul FILE care definește fișierul în care se face poziționarea capului de citire/scrivere.

Valoarea lui se definește la deschiderea fișierului prin apelul funcției *fopen*.

Cealaltă parametru se definesc la fel ca în cazul funcției *fseek*.

Funcția *fseek* returnează valoarea zero la o poziționare corectă și o valoare diferită de zero în caz de eroare.

O altă funcție utilă în cazul accesului aleator la fișier, este funcția *fstell*. Această funcție permite să se cunoască poziția curentă a capului de citire/scrivere. Ea are prototipul:

```
long fstell(FILE *pf);
```

unde:

pf - Este pointerul spre tipul FILE care definește fișierul în cauză.

Funcția returnează valoarea care definește poziția curentă a capului de citire/scrivere.

Valoarea returnată de *fstell* este deplasamentul în octeți a poziției capului de citire/scrivere față de începutul fișierului.

16.2.8. Prelucrarea fișierelor binare

Fișierele organizate ca date binare (octeții nu sunt considerați ca fiind coduri de caractere) pot fi prelucrate folosind funcțiile *fread* și *fwrite*.

În acest caz se consideră că înregistrarea este o colecție de *articole*. Articolul este o date de un tip oarecare (predefinit sau definit de utilizator). La o citire, se transferă într-o zonă specială, numita *zonă tampon* (buffer), un număr de articole care se presupune că au o lungime fixă. În mod analog, la scriere se transferă din

zona tampon un număr de articole de lungime fixă.

Prototipul funcției *fread* este următorul:

```
unsigned fread(void *ptr, unsigned dim, unsigned nrart, FILE *pf);
```

unde:

ptr - Este pointerul spre zona tampon care conține articolele citite (înregistrarea citită).

dim

nrart - Definește dimensiunea unui articol în octeți.

pf

- Definește numărul de articole din compunerea înregistrării citite.

- Este pointerul spre tipul FILE care definește fișierul din care se face citirea.

Funcția returnează numărul de articole citite sau -1 în caz de eroare.

Funcția *fwrite* are prototipul similar cu cel al funcției *fread*:

```
unsigned fwrite(void *ptr, unsigned dim, unsigned nrart, FILE *pf);
```

unde:

ptr - Este pointerul spre zona tampon care conține articolele care se scriu în fișier.

dim

nrart - Definește lungimea unui articol.

pf

- Definește numărul de articole din compunerea înregistrării care se scrie în fișier.

- Este pointerul spre tipul FILE care definește fișierul în care se scrie înregistrarea.

Funcția returnează numărul articolelor scrise în fișier sau -1 în caz de eroare.

Menționăm că pentru a prelucra fișiere binare, la deschiderea lor se folosește modul *b*:

"wb" - Pentru creare.

"rb" - Pentru citire.

"r+b" - Pentru citire/scrivere.

"w+b" - Idem.

"ab" - Pentru adăugare de înregistrări.

16.3. Stergerea unui fișier

Un fișier poate fi sters apelând funcția *unlink*. Aceasta are prototipul:

```
int unlink(const char *calea);
```

unde:

calea - Este un pointer spre un sir de caractere identic cu cel utilizat

la crearea fișierului în funcția *creat* sau *fopen*.

Funcția *unlink* returnează valoarea zero la o ștergere reușită și -1 în caz de eroare.

16.4. Redirectarea fișierelor de intrare/ieșire standard

Am văzut că funcțiile: *scanf*, *printf*, *gets* și *puts*, precum și macrourile *getchar* și *putchar* citesc și scriu la terminalul standard. Ele pot fi utilizate și cu alte periferice dacă, în prealabil, se face o redirectare a fișierelor standard de intrare/ieșire. Acest lucru se realizează folosind, în linia de comandă a apelului execuției programului, caracterele '<' și '>'.

Caracterul '<' se folosește pentru redirectarea fișierului de intrare (*stdin*), iar caracterul '>' pentru redirectarea fișierului de ieșire (*stdout*).

Aceste caractere se folosesc în felul următor:

<specificator_de_fisier_de_intrare

și

>specificator_de_fisier_de_iesire

Specificatorul de fișier depinde de sistemul de operare utilizat.

În cazul sistemului de operare MS-DOS specificatorul de fișier poate fi:

prn - Pentru ieșirea pe imprimanta paralelă.

com1 - Pentru transferul de date serial.

calea - Spre un fișier de pe disc.

NUL - Perifericul nul.

Menționăm că fișierul de ieșire standard pentru erori (*stderr*) nu poate fi redirectat.

Exemple:

1. Programul BXVI4 copiază intrarea standard la ieșirea standard. Fie BXVI4.EXE imaginea executabilă a programului respectiv.

Un apel de forma:

>BXVI4.EXE >prn

afisează la imprimanta paralelă caracterele citite de la intrarea standard.

2. Apel:

>BXVI4.EXE <fis1.dat>fis2.dat

crează fișierul *fis2.dat* prin copierea fișierului *fis1.dat*. Ambele fișiere sunt în directorul curent (în care se află și fișierul BXVI4.EXE).

3. Apel:

>BXVI4.EXE <fis1.dat

afisează la ieșirea standard conținutul fișierului *fis1.dat*.

Menționam că pentru a opri afișarea în vederea analizării datelor afișate, se acționează tasta *Break*. Pentru a continua afișarea se va acționa o tasta oarecare, de exemplu bara pentru spațiu.

Exerciții:

- 16.6 Să se scrie un program care citește de la intrarea standard datele a căror format sunt definite mai jos și le scrie în fișierul misc.dat din directorul curent.

Fișierul este organizat binar, fiecare articol conținând următoarele date:

tip	denumire	nm	cod	preț	cantitate
I	REZISTENTA	010KO	123456789	10	10000

PROGRAMUL BXVI6

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

#define MAX 50

typedef struct {
    char tip[2];
    char den[MAX+1];
    int val;
    char unit[3];
    long cod;
    float pret;
    float cant;
} ARTICOL;
/* 6 articole în zona tampon */

union {
    ARTICOL a[6];
    char zt[6*sizeof(ARTICOL)];
} buf;

int citart(ARTICOL *str)
/* - citește datele de la intrarea standard și le păstrează în zona spre care pointează str;
   - returnează EOF la întâlnirea sfârșitului de fișier și 1 în caz contrar.*/
{
    int c,nr;
    float x,y;
    for(;;)
        if((nr=fread(str->tip,str->den,&str->val,str->unit,
                      &str->cod,&x,&y))!=7){
            if(nr==EOF) return EOF;
            printf("date eronate se reia citirea\n");
            fflush(stdin);
        }
        else
```

```

        break;
str->pret=x;
str->cant=y;
return 1;
} /* sfirsit citire */

main() /* crearea fisierului misc.dat cu datele citite de la intrarea standard */
{
FILE *pf;
int i,n;

/* se deschide fisierul in scriere binara */
if((pf=fopen("misc.dat","wb"))==0)
printf("nu se poate deschide in creare misc.dat\n");
exit(1);
}
for(;;){
/* se umple zona tampon a fisierului */
for(i=0;i<6;i++)
if((n=fcitart(&buf.a[i]))==EOF) break; /* s-a intilnuit EOF */

/* se scrie zona tampon daca nu este vida */
if(i) /* zona tampon nu este vida */
if(i!=fwrite(buf.zt,sizeof(ARTICOL),i,pf)){
/* eroare la scrierea in fisier */
printf("eroare la scrierea in fisier\n");
exit(1);
}
if(n==EOF) break;
} /* sfirsit for */
fclose(pf);
} /* sfirsit main */

```

Observație:

În datele de intrare, *tip* este o literă:

- I* - Intrari.
- E* - Ieșiri.
- T* - Transfer etc.

Tipul se citește folosind specificatorul `%ls`. În felul acesta se omit eventualele caractere albe care preced litera ce definește tipul.

16.7 Să se scrie un program care afișează articolele păstrate în fișierul `misc.dat` creat în exercițiul precedent, pentru *tip=I*.

PROGRAMUL BXVI7

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 50
```

```

typedef struct {
char tip[2];
char den[MAX+1];
int val;
char unit[3];
long cod;
float pret;
float cant;
}ARTICOL;

union {
ARTICOL a[6];
char zt[6*sizeof(ARTICOL)];
}buf;

main() /* citeste si afisaza articolele din fisierul misc.dat pentru tip=I */
{
FILE *pf;
int i,n;

if((pf=fopen("misc.dat","rb"))==0)
printf("nu se poate deschide fisierul misc.dat\n");
exit(1);
}
printf("%30s\n\n", "INTRARI");
while((n=fread(buf.zt,sizeof(ARTICOL),6,pf))>0)

/* listeaza articolele dintr-o inregistrare cu tipul=I */
for(i=0;i<n;i++)
if(buf.a[i].tip[0]=='I')
printf("%-50s %03d%s %09ld %.2f %.2f\n",
buf.a[i].den,buf.a[i].val,
buf.a[i].unit,buf.a[i].cod,
buf.a[i].pret,buf.a[i].cant);
fclose(pf);
}
```

17. FUNCȚII STANDARD

Mediul de programare TURBO C pune la dispoziția utilizatorului un set de funcții standard care pot fi utilizate cu succes într-o serie de aplicații din diferite domenii. Aceste funcții pot fi apelate atât din programe scrise în C cât și din cele scrise în C++. Ele pot fi grupate în mai multe clase, ținând seama de utilizările lor.

Apelurile funcțiilor standard implică includerii de fișiere de tip *h* care contin prototipurile lor. Ca exemple de astfel de fișiere putem aminti: *stdio.h*, *stdlib.h*, *conio.h*, *io.h*, *fenvt.h*, *alloc.h* etc.

De obicei, fișierelor de tip *h* relative la funcțiile standard se află în subdirectorul INCLUDE al directorului TC pentru TURBO C sau BORLANDC pentru TURBO C++.

Listarea subdirectorului INCLUDE printr-o comandă de forma:

```
dir \tc\include
```

permite afișarea numelor fișierelor de tip *h*.

În continuare se poate lista conținutul fișierului *nume.h* folosind o comandă de forma:

```
type \tc\include\nume.h
```

În cazul în care dorim să afișăm la imprimanta paralelă conținutul fișierului *nume.h*, putem utiliza comanda de mai jos:

```
type \tc\include\nume.h >prn
```

Printr-o astfel de listare se pot pune în evidență prototipurile funcțiilor standard aflate în fișierul respectiv.

Pentru a obține informații suplimentare despre o funcție standard se poate proceda ca mai jos.

1. Setare pe directorul *tc* sau *borlandc*:
 >cd \tc sau cd \borlandc
2. Se activează mediul de dezvoltare integrat TURBO C sau C++ tastând:
 tc sau *bc*
3. Se activează meniul de editare:
 <Alt> - E
4. Se tastează, în fereastra de editare, numele funcției standard despre care se doresc informații.
5. Acționând tasta cu săgeată spre stânga, se vine cu cursorul pe una din literele din compunerea numelui funcției respective.
6. Se tastează
 <Ctrl> - F1

Se obține o fereastră de *Help* cu explicații despre funcția respectiva. Aceste explicații conțin:

- prototipul funcției;
- fișierul sau fișierele de tip *h* care conțin prototipul funcției respective;
- descrierea pe scurt a efectului apelului funcției respective;
- nume de parametri și eventualele funcții înrudite pentru care se pot consulta informații de *help* care să vină în completarea celor de față;
- exemple de utilizare a funcției respective.

Cu ajutorul tastelor cu săgeți și a tastei TAB se pot selecta nume de parametri și de funcții înrudite în vederea obținerii de informații de *help* suplimentare. În acest scop, după selectarea unui astfel de nume se acționează tasta *Enter*. Se deschide o nouă fereastră de *help*.

În cazul în care există mai multe ferestre de *Help*, se poate trece de la o fereastră la alta folosind tasta *PgDn*.

Pentru a reveni la o fereastră anterioară vom acționa tasta *PgUp*.

Pentru a reveni dintr-o fereastră de *Help* în fereastra de editare se acționează tasta *ESC*.

Amintim că pentru a reveni în sistemul de operare se acționează:

<Alt> - X

În cele ce urmează vom trece în revistă clasele de funcții standard utilizate mai frecvent.

O prima clasă de funcții sunt cele destinate prelucrării fișierelor. Funcțiile din această clasă care se utilizează mai frecvent au fost descrise în capitolul precedent. De asemenea, funcțiile utilizate la operațiile de intrare/iesire cu terminalul standard au fost folosite în toate capitolele cărții de față. De aceea, în acest capitol nu ne vom mai referi la funcțiile din această clasă.

O altă clasă de funcții standard se referă la alocarea dinamică a memoriei. Acestea au prototipul în fișierul *alloc.h*.

Funcțiile mai importante din această clasă au fost descrise și utilizate deja în mai multe capitole din carte. În acest capitol ne vom referi la încă o funcție din această clasă care uneori are o utilizare importantă (funcția *coreleft*).

Într-un capitol precedent au fost prezentate funcțiile mai importante utilizate la prelucrarea sirurilor: *strlen*, *strcpy*, *strcat* și *strcmp* împreună cu unele versiuni ale lor.

Ele au prototipul în fișierul *string.h*.

ACESTE FUNCȚII AU FOST UTILIZATE ÎN DIFERITE EXERCITII DIN CAPITOЛЕLE PRECEDENTE ȘI DE ACEEA NU NE VOM MAI REFERI LA ELE ÎN CAPITOЛUL DE FAȚĂ.

O clasă importantă de funcții o constituie funcțiile de gestiune a ecranului. Aceste funcții au prototipul în fișierele *conio.h* și *graphics.h*. Ele prezintă o importanță mare în diferite aplicații și de aceea funcțiile respective vor fi descrise și aplicate în capitolele următoare.

În capitolul de față ne vom opri asupra următoarelor clase:

- funcții pentru realizarea de conversii;
- funcții de calcul;
- funcții pentru gestiunea memoriei;
- funcții de control ale proceselor;
- funcții de gestiune a datei și orei.

De asemenea, ne vom opri și asupra unor macrouri definite în fișierul *ctype.h* și anume:
macrouri de clasificare
și
macrouri de transformare a simbolurilor.

17.1. Macrouri de clasificare

În această clasă distingem un număr de macrouri care au aplicații în prelucrarea caracterelor. Aceste macrouri sunt definite în fișierul *ctype.h*.

Un prim macro din această clasă este macroul *isascii* de prototip:

```
int isascii(int c);
```

El returnează o valoare diferită de zero dacă valoarea lui *c* aparține intervalului [0,127] și zero în caz contrar.

Celelalte macrouri sunt de forma:

```
int is ... (int c);
```

Valoarea lui *c* trebuie să aparțină intervalului [0,127]. Cele trei puncte reprezintă o succesiune de litere ca mai jos.

ACESTE MACROURI REPREZINTĂ O VALOARE DIFERITĂ DE ZERO DACĂ *c* ESTE:

pentru	<i>isalpha</i>	o literă
	<i>isalnum</i>	o literă sau o cifră
	<i>isdigit</i>	o cifră
	<i>isgraph</i>	un caracter grafic
	<i>islower</i>	o literă mică
	<i>isprint</i>	un caracter imprimabil
	<i>isspace</i>	un caracter spațiu, tabulator, return de car, interline, tabulator vertical sau salt de pagina
	<i>isupper</i>	o literă mare
	<i>isxdigit</i>	o cifră hexazecimală

17.2. Macrouri de transformare a caracterelor

În această clasă distingem macrouri definite tot în fișierul *ctype.h*. Prototipurile lor sunt:

- int toascii(int c);** - returnează ultimii 7 biți ai lui *c*.
- int tolower (int c);** - transformă pe *c* din literă mare în literă mică.
- int toupper(int c);** - transformă pe *c* din literă mică în literă mare.

Exemplu:

Modificăm programul din exercițiul 16.4 așa încât literele mari de la intrarea standard să fie afișate ca litere mici:

```
...  
while((c = getchar()) != EOF)  
    if(isupper(c)) putchar(tolower(c));  
    else putchar(c);  
...
```

17.3. Funcții de conversie

Biblioteca standard a limbajului C conține mai multe funcții de conversie a datelor din format extern în cel intern și invers.

Până în prezent am utilizat două astfel de funcții:
sscanf

și
sprintf

Aceste funcții au prototipurile în fișierul *stdio.h*. Ele permit realizarea de conversii sub controlul formatorilor.

În afară de aceste două funcții, biblioteca standard mai conține și alte funcții de conversie dintre care amintim pe cele utilizate mai frecvent.

Funcția *atoi* are prototipul:

```
int atoi(const char *p);
```

unde:

p - Pointează spre un sir de caractere în compunerea căruia intră cifre zecimale și care eventual sunt precedate de un semn.

Funcția convertește întregul zecimal definit de sirul spre care pointează *p* în binar de tip *int* și returnează rezultatul acestei conversii.

Funcția *itoa* realizează conversia inversă.

Ea are prototipul:

```
char *itoa(int val, char *sir, int baza);
```

unde:

val - Este valoarea binară de tip *int* care se convertește ca un

(continuare)

nume	echivalentul în limbajul matematic
exp	e la puterea x
log	ln
log10	log în baza 10
sqrt	radacina patrată
pow10	10 la puterea x
ceil	rotunjire prin exces
floor	rotunjire prin lipsă
fabs	valoare absolută
sinh	sh
cosh	ch
tanh	th

sir

- Este pointerul spre zona în care se păstrează rezultatul conversiei sub formă de sir de caractere.

baza

- Este baza rezultatului.

Funcția returnează valoarea pointerului *sir*.

Biblioteca conține funcții similare pentru tipul *long*. Astfel, funcția *atol* are prototipul:

long atol(const char *p);

și realizează conversia din zecimal a șirului spre care pointează *p*, în binar de tip *long*.

Conversia inversă se realizează cu ajutorul funcției *itoa*:

char *itoa(long val,char *sir,int baza);

Parametrii *val*, *sir* și *baza* au aceleași utilizări ca și în cazul funcției *itoa*. Funcția returnează valoarea pointerului *sir*.

Toate aceste funcții au prototipul în *stdlib.h*.

Tot din această clasă face parte și funcția *atof* de prototip:

double atof(const char *p);

Această funcție convertește un număr aflat în zona spre care pointează *p*, în format flotant dublă precizie și returnează rezultatul acestei conversii. Funcția *atof* realizează aceeași conversie ca și cea realizată cu ajutorul funcției *scanf* relativ la specificatorul de format `%lf`. Aceasta înseamnă că în compunerea numărului spre care pointează *p* se poate afla caracterul punct, precum și un exponent.

Funcția *atof* are prototipul în fișierele *math.h* și *stdlib.h*.

17.4. Funcții de calcul

Majoritatea funcțiilor standard utilizate la calcule numerice au prototipul în fișierul *math.h*. Multe dintre acestea au prototipul:

double nume (double x);

unde:

nume

- Are semnificația de mai jos:

nume	echivalentul în limbajul matematic
acos	arccos
asin	arcsin
atan	arctg
cos	cos
sin	sin

Alte funcții de calcul cu prototipurile în *math.h* și care se utilizează frecvent sunt:

double atan2(double y,double x);

returnează arctg(*y/x*);

double pow(double x,double y);

returnează *x* la puterea *y*;

double cabs(struct complex z);

returnează modulul numărului complex *z*;

double poly(double x, int n, double c[]);

returnează valoarea polinomului în *x* de grad *n*, coeficienții fiind păstrați în tabloul *c* astfel:

c[0]

- Termen liber.

c[1]

- Coeficientul lui *x*.

...

- Coeficientul lui *x* la puterea *i*.

c[n]

- Coeficientul lui *x* la puterea *n*.

Următoarele funcții de calculul au prototipurile în fișierul *stdlib.h*:

int abs(int x);

returnează valoarea absolută a lui *x* de tip *int*;

long labs(long x);

returnează valoarea absolută a lui *x* de tip *long*;

int rand(void);

returnează un număr natural pseudo-aleator mai mic decit 32768;

```
int random(int nr);
```

returnează un număr natural pseudo-aleator mai mic decit *nr*;

```
void srand(int n);
```

setează sămînta numerelor pseudo-aleatoare la valoare lui *n*.

O parte din funcțiile de calcul amintite mai sus au fost utilizate în diferite exerciții din capituloarele acestei cărți.

17.5. Funcții pentru gestiunea memoriei

Funcțiile din această clasa au prototipurile în fișierul *alloc.h*.

În paragraful 8.8 au fost amintite funcțiile *malloc*, *calloc*, *free*, și versiunile acestora pentru cazul în care pointerii sunt de tip *far* : *farmalloc*, *farcalloc* și *farfree*.

În acest paragraf amintim funcția *coreleft* care are prototipul:

```
unsigned coreleft(void);
```

Această funcție se utilizează pentru modelele de memorie: *tiny*, *small* și *medium* (pentru aceste modele pointerii sunt de tip *near*). Ea returnează dimensiunea memoriei libere în momentul apelului ei.

Pentru modelele de memorie *compact*, *large* și *huge* (pointerii sunt de tip *far*) funcția *coreleft* are prototipul:

```
unsigned long coreleft(void);
```

Menționăm că în cazul mediilor integrate de dezvoltare Turbo C și C++, modelul se definește cu ajutorul submeniuului *Compile* al meniului *Options*.

17.6. Funcții de control ale proceselor

Indicăm mai jos prototipurile unor funcții mai importante din această clasă.

Funcția *exit* are prototipul:

```
void exit(int stare);
```

Ea termină execuția programului.

Parametrul *stare* are valoarea zero la o terminare normală a execuției programului și diferență de zero pentru o terminare anormală. Funcția *exit* videază zonele tampon ale fișierelor deschise în scriere, în momentul apelului ei. Apoi se închid toate fișierele deschise.

Funcția *system* are prototipul:

```
int system(const char *comanda);
```

unde:

comanda

- Este un pointer spre un sir de caractere care reprezintă o comandă MS-DOS. Ea are ca efect execuția comenzii definite de sirul de caractere spre care pointează parametrul *comanda*.

Funcția returnează codul de stare al execuției comenzii:

- 0 - La terminare normală.
- 1 - La eroare.

Funcția *spawnl* este asemănătoare cu funcția *system*. Ea permite lansarea în execuție a imaginii executabile a unui program păstrat într-un fișier.

Prototipul funcției este:

```
int spawnl(int mod,char *cale,char *arg0,char *arg1,...,NULL);
```

unde:

mod

- Are ca valoare constanta simbolică *P_WAIT*.

- Această valoare înseamnă că programul din care se apelează funcția *spawnl* își intrerupe execuția pînă la terminarea execuției programului lansat prin apelul funcției *spawnl*.

cale

- Este pointerul spre sirul de caractere care definește fișierul cu imaginea executabilă a programului care se lansează.

arg0,arg1,...

- Sunt pointeri spre sîruri de caractere care reprezintă argumentele programului care se lansează.

După o execuție cu succes a programului apelat, funcția *spawnl* returnează valoarea codului de stare de la apelul funcției *exit* prin care s-a terminat execuția programului respectiv.

În caz de eroare (nu se reușește lansarea în execuție a unui program), funcția *spawnl* returnează valoarea -1.

Acste funcții au prototipul în fișierele *stdlib.h* și *process.h*.

17.7. Funcții pentru gestiunea datei și a orei

Indicăm mai jos prototipurile a patru funcții pentru citirea/setarea datei și a orei. Ele implică includerea fișierului *dos.h*.

Funcțiile pentru citirea datei și orei au prototipurile:

```
void getdate(struct date *d);
```

și

```
void gettime(struct time *t);
```

Funcțiile pentru setarea datei și orei au prototipurile:

```
void setdate(struct date *d);
```

```
void settime(struct time *t);
```

Structurile *date* și *time* sint definite in *dos.h* astfel:

```
struct date {  
    int da_year;  
    int da_day;  
    int da_mon;  
};  
struct time {  
    unsigned char ti_min;  
    unsigned char ti_hour;  
    unsigned char ti_hund;  
    unsigned char ti_sec;  
};
```

17.8. Diferite funcții de uz general

Funcția *clrscr* de prototip:

```
void clrscr(void);
```

Șterge fereastra activă sau tot ecranul dacă n-a fost activat în prealabil o fereastră (definirea și activarea unei ferestre se va descrie într-un capitol următor; în mod implicit *clrscr* șterge tot ecranul).

Prototipul funcției se află în fișierul *conio.h*.

Funcția *kbhit* de prototip:

```
int kbhit(void);
```

returnnează o valoare diferită de zero dacă există un caracter disponibil de la tastatură; în caz contrar ea returnnează valoarea zero.

Funcția *kbhit* se poate apela ciclic pentru a aștepta tastarea unui caracter la terminalul standard. Astfel, ciclul:

```
while(!kbhit())  
    ;
```

se execută pînă în momentul în care se acționează o tastă la terminalul standard.

Pe perioada execuției ciclului se afișează ecranul utilizator și de aceea acest ciclu se poate utiliza la fel ca și apelul funcției *getch* pentru a afișa ecranul respectiv.

Funcțiile care urmează și încep cu prefixul *str* implică includerea fișierului *string.h*.

Funcția *strlwr* are prototipul:

```
char *strlwr(char *sir);
```

Ea convertește literele mari din zona spre care pointează *sir* în litere mici.

Funcția returnnează valoarea *sir*.

Funcția *strupr* de prototip:

```
char *strupr(char *sir);
```

Este inversa funcției *strlwr*.

Funcția *strset* de prototip:

```
char *strset(char *sir,int c);
```

Initializează zona spre care pointează *sir* cu valoarea lui *c* (ultimii 8 biți, cei mai puțini semnificativi). Substituirea valorii lui *c* se realizează pînă la întîlnirea caracterului NUL, care rămîne nemodificat. Funcția returnnează valoarea lui *sir*.

O variantă a acestei funcții este funcția *strnset*. Ea are prototipul:

```
char *strnset(char *sir,int c,unsigned n);
```

Funcția are aceeași acțiune ca și funcția *strset*, cu deosebirea că se initializează cu valoare lui *c* numai cel mult primele *n* caractere ale zonei spre care pointează *sir*.

Funcția *strchr* are prototipul:

```
char *strchr(const char *sir,char c);
```

Se caută prima apariție a caracterului *c* în zona spre care pointează *sir*.

Funcția returnnează pointerul spre poziția determinată sau zero în cazul în care nu există o intrare a caracterului *c* în zona respectivă.

Funcția *strrechr* are prototipul:

```
char *strrechr(const char *sir,char c);
```

Ea are aceeași acțiune ca și funcția *strchr*, cu deosebirea că determină numărul apariției a caracterului *c*, cî ultima.

Funcțiile care urmează au prototipul în fișierul *dos.h*.

Funcția *delay* are prototipul:

```
void delay(unsigned i);
```

Ea suspendă execuția programului pentru o perioadă de *i* milisecunde.

Funcția *sleep* are prototipul:

```
void sleep(unsigned i);
```

Ea suspendă execuția programului pentru o perioadă de *i* secunde.

Funcția *sound* are prototipul:

```
void sound(unsigned h);
```

Ea pornește difuzorul calculatorului cu un ton egal cu *h* Hz.

Funcția *nosound* are prototipul:

```
void nosound(void);
```

Ea oprește sunetul de la difuzorul calculatorului.

17.9. Tratarea erorilor

Distingem două mari categorii de erori:

erori care apar la compilare;

și

erori care apar la execuție.

Erorile de compilare sunt de trei niveluri:

erori fatale

- Semnifică o eroare în compilator.
- Dacă aceste erori nu provin de la apelul incorrect al unei definiții de macro, atunci ele vor fi semnalate firmei Borland.
- Compilatorul afișează, în fereastra de mesaje, pentru fiecare eroare, un text explicit al acesteia.
- Sunt prevăzute aproximativ 140 de cazuri de eroare.

erori

- Sunt mesaje care indică:

- Situații care în anumite cazuri pot constitui erori.
- Informații cu privire la portabilitatea și calitatea programului. Sunt prevăzute circa 30 de astfel de cazuri.

Erorile care apar la execuție pot fi tratate prin:

- Valorile returnate de DOS prin intermediul variabilelor globale *errno*, *doserrno*, *sys_errlist*, *sys_nerr*. Aceste variabile sunt declarate în fișierele <errno.h> și <dos.h>. Funcția *perror* afișează mesajul de eroare corespondent.
- Gestionaerea erorilor prin funcția *harderr* din <dos.h>.
- Detectarea erorilor prin funcția *ferror* în prelucrarea fișierelor.
- Gestionaerea erorilor de calcul prin apelul funcției *matherr*.

Exerciții:

17.1 Programul de mai jos afișează data și ora furnizate de sistemul de operare.

Formatul de afișare este:

zz//aaaa ora hh:mm:ss

PROGRAMUL BXVII1

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

main() /* afișează data și ora */
{
    struct date d;
    struct time t;
    getdate(&d);
```

```
    gettime(&t);
    printf("\n\t%02d/%02d/%04d", d.da_day, d.da_mon, d.da_year);
    printf("\t%02d:%02d:%02d\n", t.ti_hour, t.ti_min, t.ti_sec);
}
```

17.2 Să se scrie un program care afișează un sir de numere naturale pseudo-alatoare mai mici decit $10^{*(sec+1)}$, unde prin *sec* s-a notat secunda din ora curentă.

PROGRAMUL BXVII2

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

main() /* - afișează 3 numere pseudo-alatoare mai mici decit
          10*(sec+1);
          - sec este secunda curentă. */
{
    struct time ora_crt;
    unsigned sec;
    int i;

    gettime(&ora_crt);
    sec=ora_crt.ti_sec;
    sec++;
    for(i=0;i<3;i++) printf("%d\n", random(10*sec));
}
```

17.3 Să se scrie un program care afișează un sir de numere naturale pseudo-alatoare mai mici decit *n*, citit de la intrarea standard. Primul număr se afișează automat, iar celelalte în urma acțiunării unei taste.

Sămîntă sirului de numere pseudo-alatoare *s*, se calculează astfel:

$s=(ti_min*60 + ti_hour*3600 + ti_sec) \% 65535;$

Programul se întrerupe la tastarea cifrei zero.

PROGRAMUL BXVII3

```
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <conio.h>

main () /* afișează numere pseudo-alatoare din intervalul [0,n] */
{
    struct time ora_crt;
    int i,n,s;

    for(;;){
        printf("n=");
        if((i=scanf("%d",&n))==1&&n>0) break;
        printf("nu s-a tastat un intreg pozitiv\n");
    }
```

```

if(i==EOF){
    printf("s-a tastat EOF \n");
    exit(1);
}

fflush(stdin); /* videaza zona tampon a fisierului de intrare standard */
gettime(&ora_crt);
s=(3600L*ora_crt.ti_hour+60*ora_crt.ti_min+
    ora_crt.ti_sec)%65535;
srand(s);
for(;;){
    printf("%d\n",random(n));
    printf("Pentru a continua actionati o tasta\neferita de zero\n");
    if(getch()=='0') exit(0);
}

```

Observație:

Numerele pseudo-aleatoare obținute în acest fel se pot utiliza în diferite aplicații pentru a simula situații reale. Dacă $n=6$, atunci numerele pseudo-aleatoare rezultate, mărite cu 1, pot simula aruncarea unui zar.

- 17.4 Să se scrie un program care citește numere de la intrarea standard și le copiază într-un fișier binar pe disc. Calea spre fișier este definită de argumentul programului. În caz de eroare, programul se termină apelând funcția *exit* cu următoarele coduri de stare:

- 1 - Nu se poate deschide fișierul definit de argumentul programului.
- 2 - Eroare la scrierea numerelor în fișier.
- 3 - Eroare la închiderea fișierului.
- 4 - Fișier vid.

La terminarea programului fără erori se apelează funcția *exit* cu codul de stare zero.

PROGRAMUL BXVII4

```

#include <stdio.h>
#include <stdlib.h>

main(int argc,char *argv[])
/* citeste numere de la intrarea standard si le pastreaza in fisierul a carui
   calc este definita de argumentul argv[1]*/
{
    FILE *pf;
    int i,n;
    double f[10];
    void *zt;
    double x;

```

```

if(argc!=2){
    printf("nr arg=%d\n",argc);
    printf("%s:numar argumente eronat\n",argv[0]);
    exit(1);
}
if((pf=fopen(argv[1],"wb"))==0){
    printf("%s:nu se poate deschide fisierul:%s\n",
        argv[0],argv[1]);
    exit(1);
}
n=0;
zt=(void *)f;
printf("Tastati numerele separate prin caractere albe\n");
for(;;){
    for(i=0;i<10;i++){
        if(fscanf("%lf",&x)!=1) break;
        f[i]=x;
    }
    if(i==0)
        break;
    if(fwrite(zt,sizeof(double),i,pf)!=i){
        printf("%s:eroare la scrierea in fisierul:%s\n",
            argv[0],argv[1]);
        exit(2);
    }
    n++;
    if(i<10) break;
}
if(n==0){
    printf("%s:nu s-a citit nici un numar\n",argv[0]);
    exit(4);
}
if(fclose(pf)==0) exit(0);
printf("%s:eroare la inchiderea fisierului:%s\n",
    argv[0],argv[1]);
exit(3);
}

```

- 17.5 Să se scrie un program care realizează următoarele:

- a. Citește numere de la intrarea standard și le păstrează în fișierul *dnr.dat*.
- b. Folosind utilitarul *pkzip.exe*, determină forma condensată *dnr.zip* a fișierului *dnr.dat*.
- c. Șterge fișierul *dnr.dat*.

Punctul *a* se realizează apelind programul din exercițiul precedent. În acest scop vom folosi funcția *spawnl*.

Punctul *b* se realizează apelind utilitarul *pkzip* folosind funcția *system*.

La punctul *c* se apelează funcția *unlink*.

Utilitarul *pkzip.exe* se află în directorul *arc* de pe discul c. Se va utiliza linia de comandă:

c:\arc\pkzip.exe -a dnr.zip dnr.dat

PROGRAMUL BXVII5

```

#include <stdio.h>
#include <process.h>

main() /* citeste numere de la intrarea standard si le pastreaza in fisierul dnr.zip */
{
    char *text[] = {
        "",
        "cale spre fisier eronata\n",
        "eroare la scriere in fisier\n",
        "nu se poate include fisierul\n",
        "fisier vid\n"
    };
    int cod;
    if((cod= spawnl(P_WAIT, "C:\\borlandc\\dest\\\\BXVII4.EXE",
                     "", "dnr.dat", 0))!=0){
        printf("\n\n***%s\n", text[cod]);
        exit(1);
    }
    unlink("dnr.zip");
    if(system("c:\\arc\\pkzip.exe -a dnr.zip dnr.dat")!=0){
        printf("eroare la apelul utilitarului pkzip\n");
        exit(1);
    }
    if (unlink("dnr.dat")!=0){
        printf("eroare la stergerea fisierului dnr.dat\n");
        exit(1);
    }
}

```

Observatie:

Utilitarul *pkzip* afisează o serie de informații la terminalul standard care uneori poate afecta aspectul ecranului cu datele aplicației. Pentru a elibera acest inconvenient, se poate face o redirectare pentru a afișa datele respective pe un alt periferic. În acest caz funcția *system* se va apela folosind sirul de caractere:

"c:\arc\pkzip.exe -a dnr.zip dnr.dat >numc"

În acest caz, informațiile afișate de `pkzip` se vor scrie în fișierul `nume` din directorul curent. Aceste informații pot fi suprimate dacă redirectarea se face spre perifericul NUL, adică înlocuind `nume` din sirul de caractere de mai sus cu NUL.

17.6 Să se scrie un program care citește de la intrarea standard un text și-l redă prin sunet utilizând alfabetul Morse.

Se redau numai literele din compunerea cuvintelor. Restul caracterelor se neglijăază.

Pentru redarea punctului se utilizează un semnal cu o frecvență de 1000 de Hz.

pe o durata de 0,075 secunde.

Pentru redarea liniei se utilizează același semnal, dar pe o durată de trei ori mai mare.

De asemenea, între două caractere din compunerea unei litere se face pauză de 0,075 secunde. Între două litere din compunerea aceluiasi cuvint se face o pauză de 0,225 secunde, iar între două cuvinte o pauză de 0,375 secunde.

O codificare similară a unui text este propusă în exercițiul P.131 din Gazeta de Informatică nr. 7 - 8/1992 (pag 48). În acest caz textul se codifică printr-o secvență de cifre 0 și 1.

PROGRAMUL BXVII/6

```

        if(*q=='.') delay(UNIT); /* punct */
        else delay(3*UNIT); /* linie */
        nosound();
        if(++q) /* nu s-a terminat litera curenta */
            delay(UNIT);
    }
    if(*p) /* s-a terminat o litera */
        delay(INTERCAR);
}
/* s-a terminat un cuvint */
delay(INTERCUV);
}

```

18. GESTIUNEA ECRANULUI ÎN MOD TEXT

Biblioteca standard a limbajelor C și C++ conține funcții pentru gestiunea ecranului. Acestea poate fi gestionat în două moduri:

- mod text;

sau

- mod grafic.

În capitolul de față prezentăm funcțiile standard mai importante utilizate la gestiunea ecranului în mod text.

În capitolul următor se tratează gestiunea ecranului în mod grafic.

Toate funcțiile standard de gestiune a ecranului în mod text au prototipurile în fișierul *conio.h*.

Modul *text* presupune că ecranul este format dintr-un număr de linii și un număr de coloane. În mod curent se utilizează 25 de linii a 80 sau 40 de coloane fiecare. Aceasta înseamnă că ecranul are o capacitate de $25 \times 80 = 2000$ sau $25 \times 40 = 1000$ caractere.

Pozitia pe ecran a unui caracter se definește printr-un sistem de două coordonate întregi:

(*x,y*)

unde:

- | | |
|----------|---|
| <i>x</i> | - Este numărul coloanei în care este situat caracterul. |
| <i>y</i> | - Este numărul liniei în care este situat caracterul. |

Colțul din stanga sus al ecranului are coordonatele (1,1). Colțul din dreapta jos al ecranului are coordonatele (80,25) sau (40,25).

În mod implicit, funcțiile de gestiune a ecranului în mod text au acces la tot ecranul. Accesul poate fi limitat la o parte din ecran utilizând aşa numitele *ferestre*. *Fereastra* este un dreptunghi care este o parte a ecranului și care poate fi gestionată independent de restul ecranului.

Un caracter de pe ecran, pe lîngă coordonate, mai are și următoarele atribut:

- culoarea caracterului afișat;
- culoarea fondului;
- elipirea caracterului.

Aceste atrbute sunt dependente de adaptorul grafic utilizat. Cele mai utilizate adaptoare sunt:

- placa MDA, care este un adaptor monocrom;
- placa Hercules, care este un adaptor monocrom;
- placa CGA, care este un adaptor color;

- placa EGA, care este un adaptor color;
 - placa VGA, care este un adaptor color de mare performanță.
- Pentru adaptoarele color de mai sus, se pot utiliza 8 culori de fond și 16 pentru afișarea caracterelor.

Atributul unui caracter se definește cu ajutorul formulei:

$$(1) \text{atribut} = 16 * \text{culoare_fond} + \text{culoare_caracter} + \text{clipire}.$$

unde:

culoare_fond
(background)

culoare_caracter
(foreground)

clipire

- Este o cifră din intervalul [0,7] și are semnificația din tabela de mai jos.
- Este un întreg din intervalul [0,15] și are semnificația din tabela de mai jos.
- Are valoarea 128 (clipirea caracterului) sau 0 (fără clipire).

În tabelul de mai jos se indică corespondența dintre valorile numerice și culorile definite de ele cu ajutorul relației (1).

Culoare	Constantă simbolică	Valoare
negru	BLACK	0
albastru	BLUE	1
verde	GREEN	2
turcoaz	CYAN	3
roșu	RED	4
purpuriu	MAGENTA	5
maro	BROWN	6
gri deschis	LIGHTGRAY	7
gri inchis	DARKGRAY	8
albastru deschis	LIGHTBLUE	9
verde deschis	LIGHTGREEN	10
turcoaz deschis	LIGHTCYAN	11
roșu deschis	LIGHTRED	12
purpuriu deschis	LIGHTMAGENTA	13
galben	YELLOW	14
alb	WHITE	15
clipire	BLINK	128

În paragrafele următoare se indică funcțiile standard mai importante pentru gestiunea ecranului în mod text.

18.1. Setarea ecranului în mod text

Se realizează cu ajutorul funcției *textmode*. Aceasta are prototipul:

void textmode(int modtext);

unde:

modtext

- Poate fi exprimat numeric sau simbolic în felul următor:

Modul text activat	Constantă simbolică	Valoare
Caractere albe pe fond negru; 40 de coloane	BW40	0
Color 40 de coloane	C40	1
Caractere albe pe fond negru; 80 de coloane	BW80	2
Color 80 de coloane	C80	3
Monocrom 80 de coloane	MONO	7
Color cu 43 linii pentru placa EGA și 50 de linii pentru placa VGA	C4350	64
Modul precedent	LASTMODE	-1

Modul MONO se poate seta pe un adaptor monocolor.
Celelalte moduri se pot seta pe adaptoare color.

18.2. Definirea unei ferestre

După setarea ecranului în mod text, acesta are caracteristicile indicate în paragraful precedent.

Adesea dorim să partajăm ecranul în zone care să poată fi gestionate independent. Aceasta se realizează cu ajutorul ferestrelor.

O fereastră este o zonă dreptunghiulară de pe ecran. Ea se poate defini cu ajutorul funcției *window*. Prototipul ei este:

void window(int stanga,int sus, int dreapta,int jos);

unde:

- (*stanga, sus*) - Reprezintă coordonatele colțului din stînga sus al ferestrei.
(*dreapta, jos*) - Reprezintă coordonatele colțului dreapta jos al ferestrei.

La un moment dat o singură fereastră este *activă* și anume aceea definită de ultimul apel al funcției *window*.

Funcțiile de gestiune a ecranului în mod text acționează totdeauna asupra ferestrei active.

După setarea modului text cu ajutorul funcției *textmode*, este activ tot ecranul.

Mentionăm că funcția *window* nu are nici un efect dacă parametrii de la apel sunt cronoți.

18.3. Ștergerea unei ferestre

Fereastra activă se șterge cu ajutorul funcției *clrscr*. Ea are prototipul:

void clrscr(void);

După apelul funcției *clrscr*, fereastra activă (sau tot ecranul, dacă nu s-a definit în prealabil o fereastră prin apelul funcției *window*) devine vidă. Fondul ei

are culoarea definită prin culoarea de fond (*background*) curentă.

Funcția *clrscr* poziționează *cursorul* pe caracterul din stînga sus al ferestrei active, adică în poziția de coordonate (1,1) a ferestrei active.

18.4. Gestiuinea cursorului

Utilizatorul poate plasa cursorul pe un caracter al ferestrei folosind funcția *gotoxy*. Ea are prototipul:

```
void gotoxy(int coloana,int linie);
```

unde:

(*coloana,linie*) - Reprezintă coordonatele caracterului pe care se plasează cursorul; aceste coordonate sunt relative la fereastra activă.

Dacă coordonatele de la apel sunt în afara ferestrei active, atunci apelul funcției este ignorat.

Pozitia cursorului din fereastra activă se poate determina cu ajutorul a două funcții, care au prototipurile:

```
int wherex(void);
```

returnează numărul coloanei în care se află cursorul;

```
int wherey(void);
```

returnează numărul liniei în care se află cursorul.

Există cazuri cind se dorește *ascunderea* cursorului. Acest lucru se poate realiza printr-o secvență specială în care se utilizează funcția *geninterrupt*. O secvență de acest fel este următoarea:

```
void ascundecursor() /* face invizibil cursorul */
{
    _AH = 1;
    _CH = 0x20;
    geninterrupt(0x10);
}
```

Cursorul poate fi reafisat apelind funcția de mai jos:

```
void afiscursor() /* face vizibil cursorul */
{
    _AH = 1;
    _CH = 6;
    _CL = 7;
    geninterrupt(0x10);
}
```

Amintim că *_AH*, *_CH* și *_CL* sunt nume utilizate pentru regisztrii calculatorului.

18.5. Determinarea parametrilor ecranului

Utilizatorul are posibilitatea să obțină parametri curenti ai ecranului prin apelarea funcției *gettextinfo*. Ea are prototipul:

```
void gettextinfo(struct text_info *p);
```

unde structura *text_info* este definită în fișierul *conio.h* astfel:

```
struct text_info {
    unsigned char winleft;
    unsigned char wintop;
    unsigned char winright;
    unsigned char winbottom;
    unsigned char attribute;
    unsigned char normattr;
    unsigned char currmode;
    unsigned char screenheight;
    unsigned char screenwidth;
    unsigned char curx;
    unsigned char cury;
};
```

După apelul funcției *gettextinfo* structura de tip *text_info*, spre care pointează *p*, este completată cu următoarele informații:

- amplasarea colțurilor ferestrei;
- culoarea fondului, a caracterelor și clipirea acestora;
- modul curent;
- dimensiunea ecranului;
- poziția cursorului.

18.6. Modurile video alb/negru

Modurile video alb/negru sunt două:

modul intens

și

modul normal.

Modul intens se obține apelind funcția *highvideo* de prototip:

```
void highvideo(void);
```

Modul normal se obține cu ajutorul funcției *lowvideo* de prototip:

```
void lowvideo(void);
```

Intensitatea inițială este de obicei cea normală. Se poate reveni la intensitatea normală dacă se apeleză funcția *normvideo* de prototip:

```
void normvideo(void);
```

18.7. Setarea culorilor

Culoarea fondului se setează cu ajutorul funcției *textbackground* de prototip:

```
void textbackground(int culoare);
```

unde:

culoare

- Este un întreg în intervalul [0,7] și are semnificația definită în tabelul de la începutul acestui capitol.

Culoarea caracterelor se setează cu ajutorul funcției *textcolor* de prototip:

```
void textcolor(int culoare);
```

unde:

culoare

- Este un întreg din intervalul [0,15] și are semnificația definită în tabelul de la începutul acestui capitol.

Se pot seta ambele culori, precum și elipirea caracterului folosind funcția *textattr* de prototip:

```
void textattr(int atribut);
```

unde:

atribut

- Se definește cu ajutorul relației (1).

Exerciții:

18.1 Să se scrie o funcție care afișează parametrii ecranului.

FUNCȚIA BXVIII1

```
void pecr() /* afișează parametrii ecranului */
{
    struct text_info parecr;

    clrscr();
    gettextinfo(&parecr);
    printf("stinga:%u sus:%u dreapta:%u jos:%u\n",
           parecr.winleft, parecr.wintop, parecr.winright,
           parecr.winbottom);
    printf("atribut:%u mod curent:%u\n",
           parecr.attribute, parecr.currmode);
    printf("înălțimea ecranului:%u latimea ecranului:%u\n",
           parecr.screenheight, parecr.screenwidth);
    printf("coloana cursorului:%u linia cursorului:%u\n",
           parecr.curx, parecr.cury);
}
```

18.2 Să se scrie un program care setează pe rind modurile text, definite cu ajutorul constantelor simbolice:

BW40, C40, BW80, C80 și C4350

și afișează parametrii ecranului pentru fiecare din modurile respective.

PROGRAMUL BXVIII2

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

#include "bxviii1.cpp"

main() /* setează modurile definite cu ajutorul constantelor simbolice BW40, C40, BW80, C80 și C4350 și afișează parametrii ecranului în fiecare caz */
{
    int i;
    int tab[]={BW40,C40,BW80,C80,C4350};
    char *text[]{"BW40","C40","BW80","C80","C4350"};

    for(i=0;i<5;i++){
        textmode(tab[i]);
        pecr();
        printf("\n\t\t\t%s\n",text[i]);
        printf("Actionati o tasta pentru a continua\n");
        getch();
    }
}
```

Observație:

Programul de față presupune prezența la calculator a unui adaptor color de tip EGA sau VEGA.

18.8. Gestionația textelor

Pentru afișarea caracterelor colorate în conformitate cu atributele definite prin relația:

atribut = 16*cupoare_fond + cupoare_caracter + clipire

se pot folosi funcțiile:

- | | |
|----------------|---|
| <i>putch</i> | - Afișează un caracter. |
| <i>cpus</i> | - Afișează un sir de caractere (este analogă cu funcția <i>puts</i>). |
| <i>cprintf</i> | - Afișează date sub controlul formatorilor de conversie (este analogă cu funcția <i>printf</i>). |

Toate aceste funcții au prototipul în fișierul *conio.h*.

Atributul de culoare și clipire se setează cu ajutorul funcțiilor indicate în paragraful 18.7.

Biblioteca standard a limbajului C conține și alte funcții utile în gestionația textelor. Dintre acestea amintim pe cele mai importante.

Operațiile de stergere și inserare de linie se pot realiza prin funcțiile de mai jos:

```
void insline(void);
```

inserează o linie cu spații în fereastră; liniile de sub poziția curentă a cursorului se deplasează în jos cu o poziție;

void clreol(void);

șterge sfîrșitul liniei începind cu poziția cursorului;

void delline(void);

șterge toată linia pe care este poziționat cursorul.

Un text poate fi copiat dintr-o zonă dreptunghiulară a ecranului în alta, folosind funcția *movetext*. Ea are prototipul:

```
int movetext(int stanga, int sus, int dreapta, int jos,
             int stanga_dest, int sus_dest);
```

unde:

stanga,sus - Definesc inceputul textului care se copiază.

dreapta,jos - Definesc sfîrșitul textului care se copiază.

stanga_dest, sus_dest - Definesc poziția primului caracter al textului după copiere. Celelalte poziții rezultă automat din structura textului.

Funcția returnează valoarea:

1 - Dacă textul s-a copiat corect.

0 - La eroare.

Textele dintr-o zonă dreptunghiulară pot fi salvate într-o zonă de memorie sau citite dintr-o astfel de zonă folosind funcțiile *gettext* și *puttext*. Ele au prototipurile de mai jos:

```
int gettext(int stanga, int sus, int dreapta, int jos, void *destinatie);
```

unde:

stanga,sus - Definesc o zonă dreptunghiulară din ecran care conține textul de salvat.

dreapta,jos - Este pointerul spre zona de memorie în care se salvează textul.

Funcția returnează:

1 - La copiere cu succes.

0 - La eroare.

```
int puttext(int stanga, int sus, int dreapta, int jos, void *sursa);
```

unde:

stanga,sus - Definesc o zonă dreptunghiulară din ecran în care se va afișa textul citit din memorie.

sursa - Este pointerul spre zona de memorie din care se transferă textul pe ecran.

Funcția returnează:

1 - La copiere cu succes;

0 - La eroare.

Menționăm că fiecare caracter de pe ecran se păstrează pe doi octeți:

- pe un octet caracterul;

- pe octetul următor atributul caracterului.

Exerciții:

18.3 Să se scrie un program care afișează texte în modurile video intens și video normal.

PROGRAMUL BXVIII3

```
#include <conio.h>

main() /* afiseaza texte in modurile video intens si normal */
{
    textmode(BW80);
    window(10,4,60,4);
    clrscr();
    lowvideo();
    cputs("lowvideo");
    highvideo();
    cputs(" highvideo");
    normvideo();
    cputs(" normvideo");
    textmode(LASTMODE);
    cprintf("\n\rActionati o tasta pentru a continua" );
    getch();
}
```

18.4 Să se scrie un program care afișează toate combinațiile de culori posibile pentru fond și caractere. Se consideră că se dispune de un adaptor color EGA/VGA.

PROGRAMUL BXVIII4

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

main() /* - afiseaza toate combinatiile de culori posibile pentru fond si caractere;
           - se dispune de un adaptor EGA/VGA. */
{
    static char *tculoare[] = {
        " 0  BLACK      negru",
        " 1  BLUE       albastru",
        " 2  GREEN      verde",
        " 3  CYAN      turcoaz",
        " 4  RED        rosu",
        " 5  MAGENTA   purpuriu",
        " 6  BROWN      maro",
        " 7  LIGHTGRAY gri deschis",
        " 8  DARKGRAY   gri inchis"
    };
}
```

```

    " 9  LIGHTBLUE
    " 10 LIGHTGREEN
    " 11 LIGHTCYAN
    " 12 LIGHTRED
    " 13 LIGHTMAGENTA
    " 14 YELLOW
    " 15 WHITE
};

int i,j,k;
struct text_info atr;

gettextinfo(&atr);
for(i=0; i <8; i++){
/* i alege culoarea fondului */
    window(3,2,60,20);
    k=2;
    textbackground(i);
    clrscr();
    for(j=0; j < 16; j++, k++) {
        textcolor(j);
        gotoxy(2,k);
        /* j alege culoarea caracterului */
        if(i == j) continue;
        cputs(tculoare[j]);
    }
    gotoxy(1,18);
    printf("actionati o tasta pentru a continua\n");
    getch();
}
window(atr.winleft,atr.wintop,atr.winright,atr.winbottom);
textattr(atr.attribute);
clrscr();
}

```

18.5 Să se scrie un program pentru rezolvarea problemei turnurilor din Hanoi.

Această problemă a fost rezolvată în exercițiul 9.6. Mai jos se rezolvă aceeași problemă prin imagini pentru $n < 7$, n fiind numărul de discuri ($n > 0$). Se presupune că se dispune de un adaptor color EGA/VGA.

Programul folosește o matrice de ordinul 3×6 pentru a păstra discurile pe cele trei tije. Numim *stiva* tabloul care păstrează elementele acestei matrice. Ea este de tip *int*. Amintim că discul de cel mai mic diametru se numerotează cu 1, discul cu diametru imediat mai mare decit acesta se numerotează cu 2 și aşa mai departe. Dacă sunt n discuri, atunci discul cu cel mai mare diametru se numerotează cu n .

Discurile aflate pe tija A se păstrează ca elemente ale tabloului *stiva* pentru care primul indice este zero:

stiva[0][0], *stiva[0][1]*, *stiva[0][2]*, ...

Discurile aflate pe tija B se păstrează ca elemente ale aceluiași tablou pentru

albastru deschis",
verde deschis",
turcoaz deschis",
rosu deschis",
purpuriu deschis",
galben",
alb"

care primul indice are valoarea unu:

stiva[1][0], *stiva[1][1]*, *stiva[1][2]*, ...

În mod analog, pentru discurile aflate pe tija C se utilizează elementele pentru care primul indice are valoarea doi:

stiva[2][0], *stiva[2][1]*, *stiva[2][2]*, ...

Inițial discurile sint pe tija A, deci

stiva[0][i] = n-i pentru $i = 0, 1, \dots, n-1$.

Un alt tablou utilizat în program determină numărul discurilor aflate pe fiecare tijă. Numim *istiva* acest tablou. El are 3 elemente: *istiva[0]* are ca valoare numărul discurilor aflate pe tija A, *istiva[1]* are ca valoare numărul discurilor de pe tija B, iar *istiva[2]* are ca valoare numărul discurilor de pe tija C.

Inițial *istiva[0] = n*, *istiva[1] = 0* și *istiva[2] = 0* deoarece toate cele n discuri se află pe tija A.

Cele trei tije se reprezintă prin trei dreptunghiuri albastre. Primul are coordonatele:

- colțul din stanga sus (7,3);
- colțul din dreapta jos (8,17).

Următorul se obține făcind o translație cu 28 de coloane spre dreapta; deci acesta are coordonatele (7+28,3) și (8+28,17).

Ultimul dreptunghi se obține printr-o nouă translație tot de 28 de coloane.

Sub fiecare dreptunghi se afișează cu galben litera tijei pe care o reprezintă (A, B sau C).

Discurile se reprezintă prin dreptunghiuri colorate și acestea au culorile:

discul	n	LIGHTGRAY
	n-1	BROWN
	n-2	MAGENTA
	n-3	RED
	n-4	CYAN
	n-5	GREEN

Fondul ecranului este negru.

Mutările discurilor se realizează cu ajutorul funcției recursive *hanoi*. Aceasta este analogă cu cea definită în exercițiul 9.6. În cazul de față, pentru $n < 7$, în loc să se afișeze mutarea sub formă unui text, se realizează deplasarea prin imagini a discului care se mută de pe o tijă pe alta. Acum lucru se realizează apelând funcția *transfer*. Aceasta are prototipul:

void transfer(int tija1, int tija2);

Ea transferă discul din virful tijei *tija1*, în virful tijei *tija2*.

PROGRAMUL BXVIII5

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

int stiva[3][6];
int istiva[3];
int m;

typedef struct {
    int x;
    int y;
    int u;
    int v;
    int poz;
} F;

struct text_info parecr;

void initanoii (int n) /* initializeaza discurile pe tija A */
{
    int x,y,u,v,i,lit,culoare;

    /* salveaza parametrii ecranului */
    gettextinfo (&parecr);

    /* fond negru pentru tot ecranul */
    textbackground (BLACK);

    /* culoarea caracterelor se seteaza la galben */
    textcolor (YELLOW);

    /* sterge ecranul */
    clrscr ();

    /* se amplaseaza cele trei tije de culoare albastra fiecare */
    x = 7;
    y = 3;
    u = 8;
    v = 17;
    lite= 'A';
    for (i = 0; i<3; i++){
        window(x,y,u,v);
        textbackground(BLUE);
        clrscr();

        /* fereastra pentru litera */
        window (x,v,u,v+2);

        /* fond negru si caracter galben = 14 */
        textattr (14);
        clrscr ();
    }
}
```

```
/* serie litera */
putch (lit);
/* se modifica parametrii pentru tija urmatoare */
lit++;
x = x+28;
u = u+28;
}

/* se pun discurile pe tija A */
/* culorile discurilor au valorile 7,6,...,2 */
/* discul n are lungimea 2*n+1 caractere si inaltimea de 1 caracter */
culoare = 7;
x = 7-n;
y = 15;
u = 8+n;
for (i = n; i>0; i--){
    window (x,y,u,v-1);
    textbackground (culoare);
    clrscr ();
    culoare--;
    x++;
    u--;
    v = y;
    y = y-2;
}

/* se initializeaza tablourile stiva si istiva */
for (i = 0; i<n; i++) stiva [0][i] = n-i;
istiva[0] = i;
istiva[1] = 0;
istiva[2] = 0;
getch(); /* afiseaza starea initiala */
} /* sfirsit initanoii */

void fereastra (F *frstr,int nrstv,int vs,int ndisc)
/* defineste fereastra activa pe discul ndisc aflat in virful vs pe tija nrstv */
{
    int poz;

    /* se determina pozitia tijei */
    poz = (nrstv*4 + 1)*7; /* nrstv = 0 tija A;
                                1 tija B;
                                2 tija C;
                                poz - pozitia tijei. */

    /* se determina coordonatele discului ndisc */
    frstr -> x = poz - ndisc;
    frstr -> u = poz+1+ndisc;
    frstr -> y = 17 -2*vs; /* vs - numar discuri pe tija nrstv */
    frstr -> v = frstr -> y +1;

    /* se activeaza fereastra pe discul ndisc */
    window (frstr -> x, frstr -> y, frstr -> u, frstr -> v);
}
```

```

frstr -> poz = poz; /* pastreaza pozitia tijei */
} /* sfarsit fereastra */

void transfer ( int tija1,int tija2)
/* deplaseaza discul din virful tijei tija1 in virful tijei tija2 */
{
    int i,j,k,l,nr;
    F cf;

    /* determina numarul tijei tija1 */
    /* tija A are numarul 0, tija B are numarul 1, tija C are numarul 2 */
    if (tija1 == 'A') i = 0;
    else
        if (tija1 == 'B') i = 1;
        else i = 2;

    /* i - numarul tijei tija1;
     * j - numarul discurilor pe tija tija1;
     * nr - numarul discului din virful tijei tija1. */
    j = istiva[i];
    nr = stiva[i][j-1];

    /* - determina fereastra cu discul din virful tijei a i-a;
     * - accasta devine activa. */
    fereastra (&cf,i,j,nr);

    /* sterge fereastra din virful stivei i */
    textbackground (BLACK);
    clrscr ();
    /* refac tija a i-a */
    window ( cf.poz, cf.y, cf.poz+1, cf.v );
    textbackground (BLUE);
    clrscr ();

    /* - sc scoate discul din stiva a i-a;
     * - acesta corespunde discului care a fost in virful tijei tija1 */
    istiva[i]--;
}

/* determina numarul tijei tija2 */
if (tija2 == 'A') i=0;
else
    if (tija2 == 'B') i = 1;
    else i = 2;

/* punca discul nr pe tija a i-a */
stiva[i] [istiva[i]] = nr;
istiva[i]++;
}

/* determina fereastra pe tija a i-a pentru discul nr */
fereastra (&cf,i,istiva[i],nr);

```

```

/* fereastra activa devine fereastra de pe tija tija2 in virful careia se va pune discul nr */
/* stabileste culoarea fereastrii */
textbackground(7-m+nr);
clrscr ();
} /* sfarsit transfer */

void hanoi(int n,int a,int b,int c)
/* functia recursiva pentru rezolvarea problemei turnurilor din Hanoi */
{
    if (n == 1)
        if (m>6){
            printf("se muta discul 1 de pe tija: %c pe \
                    tija: %c\n",a,b);
            getch();
            return;
        }
        else{
            transfer(a,b);
            getch();
            return;
        }
    hanoi(n-1,a,c,b);
    if(m > 6){
        printf("discul: %d se muta de pe tija :%c pe \
                tija: %c\n",n,a,b);
        getch();
    }
    else{
        transfer (a,b);
        getch();
    }
    hanoi(n-1,c,b,a);
} /* sfarsit hanoi */

main () /* rezolva problema turnurilor din Hanoi */
{
    int i,c;
    char t[255];

    for(;;){
        printf("numarul discurilor=");
        if(gets(t)==0){
            printf("s-a tastat EOF\n");
            exit(1);
        }
        if(sscanf(t,"%d",&m)==1&&m>0) break;
        printf("nu s-a tastat un intreg pozitiv\n");
    }
    if(m < 7){
        printf("rezolvare insotita de imagini\n");
        inithanoi(m);
    }
    else printf("rezolvarea nu este insotita de imagini\n");
    hanoi (m,'A','B','C');
}

```

```

if(m < 7){
    window(1,1,parecr.screenwidth,parecr.screenheight);
    textattr(parecr.attribute);
    clrscr();
}
/* sfirsit main */

```

18.6 Să se scrie o funcție care afișează o fereastră limitată de un chenar și pe fondul căreia se afișează un întreg. Cursorul devine invizibil la afișarea ferestrei.

Funcția are prototipul:

```
void fereastra(int st, int sus, int dr, int jos, int fond,
               int culoare, int chenar, int n);
```

unde:

- | | |
|----------|---|
| (st,sus) | - Coordonatele colțului din stînga sus. |
| (dr,jos) | - Coordonatele colțului din dreapta jos. |
| fond | - Întreg din intervalul [0,7] care definește culoarea de fond a ferestrei. |
| culoare | - Întreg din intervalul [0,15] care definește culoarea pentru afișarea caracterelor. |
| chenar | <ul style="list-style-type: none"> - Definește tipul chenarului: <ul style="list-style-type: none"> • 0 - fără bordură; • 1 - linie simplă; • 2 - linie dublă; • n - numărul întreg care se afișează în fereastră începînd cu punctul de coordonate relative (3,3). |

Zona de ecran în care se afișează fereastra se păstrează în memoria *heap* înainte de a se afișa fereastra. Adresa acestei zone de memorie se pune pe o stivă definită cu ajutorul unui tablou de pointeri spre tipul *void*. Acest tablou este global și are 100 de elemente. El se definește astfel:

```
void far *stiva[100];
```

Locul liber în tablou se definește cu ajutorul variabilei globale *istiva*:

```
int istiva;
```

FUNCȚIA BXVIII6

```

void orizontal(int,int);
void vertical(int,int,int,int);

void fereastra(int st,int sus,int dr,int jos,
               int fond,int culoare,int chenar,int n)
/* afișează o fereastra limitată de un chenar și pe fondul căreia se afișează numărul ferestrei */
{

```

```

extern ELEM far *stiva[];
extern int istiva;

/* memorizează partea din ecran pe care se va afișa fereastra */
if(istiva==MAX){
    printf("\nprea multe ferestre\n");
    exit(1);
}
if((stiva[istiva]=(ELEM *)farmalloc(sizeof(ELEM)))==0){
    printf("memorie insuficientă\n");
    exit(1);
}
if((stiva[istiva]->zonfer=
    farmalloc(2*(dr-st+1)*(jos-sus+1)))==0){
    printf("\nmemorie insuficientă\n");
    exit(1);
}
stiva[istiva]->x=st;
stiva[istiva]->y=sus;
stiva[istiva]->u=dr;
stiva[istiva]->v=jos;
if((gettext(st,sus,dr,jos,stiva[istiva]->
           zonfer))==0){
    printf("\neroare la memorarea ecranului\n");
    exit(1);
}
istiva++;

/* activează fereastra și o afișează pe ecran */
window(st,sus,dr,jos);
textattr(16*fond+culoare);
clrscr();

/* trasare chenar */
if(chenar){
    textcolor(WHITE);
    highvideo();

/* colțul stînga sus */
switch(chenar){
    case SIMPLU:
        putch(218);
        break;
    case DUBLU:
        putch(201);
        break;
}

/* chenar orizontal sus */
orizontal(dr-st-2,chenar);

/* colțul dreapta sus */
switch(chenar){
    case SIMPLU:

```

```

        putch(191);
        break;
    case DUBLU:
        putch(187);
        break;
    }

/* chenar vertical stanga */
vertical(jos-sus, 1, 2, chenar);

/* colțul stanga jos */
gotoxy(1, jos-sus+1);
switch(chenar){
    case SIMPLU:
        putch(192);
        break;
    case DUBLU:
        putch(200);
        break;
}

/* chenar orizontal jos */
orizontal(dr-st-2, chenar);
/* chenar vertical dreapta */
vertical(jos-sus-1, dr-st, 2, chenar);

/* colțul dreapta jos */
gotoxy(dr-st, jos-sus+1);
switch(chenar){
    case SIMPLU:
        putch(217);
        break;
    case DUBLU:
        putch(188);
        break;
}
normvideo();
textattr(16*fond+culoare);
} /* sfîrșit afisare chenar */

/* scrie pe n în fereastra */
gotoxy(3, 3);
cprintf("%d", n);

/* ascunde cursor */
_AH=1;
_CH=0x20;
geninterrupt(0x10);

} /* sfîrșit fereastra */

void orizontal(int a, int chenar)
/* trasează un chenar orizontal */
{

```

```

while(a--)
    switch(chenar){
        case SIMPLU:
            putch(196);
            break;
        case DUBLU:
            putch(205);
            break;
    }
}

void vertical(int a, int col, int lin, int chenar)
{
    while(a--) {
        gotoxy(col, lin++);
        switch(chenar){
            case SIMPLU:
                putch(179);
                break;
            case DUBLU:
                putch(186);
                break;
        }
    }
}

```

- 18.7 Să se scrie un program care afișează ferestre pe ecran în mod aleator. Ferestrele sunt de dimensiune fixă, dar au poziții aleatoare pe ecran. De asemenea, ele pot avea chenar format dintr-o linie simplă sau dublă sau să nu aibă chenar. Culoarele de fond și de afișare a caracterelor sunt aleatoare. Ferestrele se numără și numărul ferestrelor se afișează în fereastră. Prima fereastră afișată se numără cu 1.

După afișarea unei ferestre se va acționa o tastă oarecare, corespunzătoare codului ASCII, pentru a afișa fereastra următoare. Se revine la fereastra precedentă dacă se acționează tasta ESC.

Execuția programului se termină în cazul în care se acționează tasta ESC în momentul în care nu este afișată nici o fereastră.

Se presupune că se dispune de un adaptor color EGA/VGA.

PROGRAMUL BXVIII

```

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <alloc.h>
#include <dos.h>

#define MAX 100
#define ESC 0x1b
#define SIMPLU 1
#define DUBLU 2

```

```

typedef struct {
    int x,y,u,v;
    void far *zonfer;
} ELEM;
ELEM far *stiva[MAX];

int istiva;

#include "bviii2.cpp" /* pcit_int */
#include "bviii3.cpp" /* pcit_int_lim */
#include "bxviii6.cpp" /* fereastra */

main() /* afiseaza ferestre pe ecran in mod aleator */
{
    int c,culoare,fond,i,inalt,j,lung,s,stanga,sus;
    struct text_info info,crt;
    struct time ora_crt;

    istiva=0;
    clrscr();

    /* pastreaza tot ecranul */
    if((stiva[istiva]=(ELEM *)farmalloc(sizeof(ELEM)))==0){
        printf("memorie insuficienta\n");
        exit(1);
    }
    if((stiva[istiva]->zonfer=farmalloc(2*80*25))==0){
        printf("memorie insuficienta\n");
        exit(1);
    }
    if(gettext(1,1,80,25,stiva[istiva]->zonfer)==0){
        printf("nu se poate salva ecranul\n");
        exit(1);
    }
    istiva++;

    /* salveaza parametrii ecranului */
    gettextinfo(&info);

    /* citeste dimensiunile ferestrelor: lungimea si inaltimea */
    if(pcit_int_lim("lungime:",8,70,&lung)==0){
        printf("s-a tastat EOF\n");
        exit(1);
    }
    if(pcit_int_lim("inaltime:",5,15,&inalt)==0){
        printf("s-a tastat EOF\n");
        exit(1);
    }

    /* coordonatele maxime pentru coltul din stinga sus a ferestrelor */
    i=79-lung;
    j=25-inalt;
}

```

```

/* seteaza saminta pentru sirul de numere pseudo-alatoare care definesc
parametrii ferestrelor:
    - coltul din stinga sus;
    - atributul de culoare. */
gettime(&ora_crt);
s=(3600L*ora_crt.ti.hour+60*ora_crt.ti_min+
    ora_crt.ti_sec)%65535;
 srand(s);
printf("Actionati o tasta pentru a continua\n");
printf("cu ESC se revine la ecranul precedent\n");
for(;;){
    c=getch();
    if(c!=ESC){
        /* s-a tastat un caracter diferit de ESC */
        /* se genereaza parametrii ferestrei */
        stanga=random(i)+1;
        sus=random(j)+1;
        fond=random(8);
        while((culoare=random(15)+1)==fond)
            ;
        fereastra(stanga,sus,stanga+lung,sus+inalt,
                   fond,culoare,istiva%3,istiva);
        continue;
    }

    /* s-a tastat ESC */
    if(--istiva>0){
        /* se reface zona din ecran eliminind fereastra activa */
        puttext(stiva[istiva]->x,stiva[istiva]->y,
                stiva[istiva]->u,stiva[istiva]->v,
                stiva[istiva]->zonfer);
        farfree(stiva[istiva]);
    }
    else
        /* se interupe executia programului */
        break;
}
puttext(info.winleft,info.wintop,info.winright,
        info.winbottom,stiva[0]);
window(1,1,80,25);
farfree(stiva[0]);
textattr(info.attribute);

/* afiseaza cursorul */
_AH=1;
_CH=6;
_CL=7;
geninterrupt(0x10);
clrscr();
}

```

19. GESTIUNEA ECRANULUI ÎN MOD GRAFIC

Modul *grafic* presupune ca ecranul este format din "puncte luminoase" (*pixeli*). Numărul acestora depinde de adaptorul grafic și se numește *rezoluție*. O rezoluție este cu atit mai bună cu cit este mai mare.

Adaptorul CGA are o rezoluție de 200 de rinduri a 640 de coloane, iar adaptorul EGA oferă o rezoluție de tot atâtea coloane, dar de 350 de rinduri.

Adaptorul VGA oferă o rezoluție de 480 de rinduri a 640 de coloane sau chiar mai mare, de pînă la 768 de rinduri, a 1024 de coloane.

Pentru gestiunea ecranului în mod grafic se pot utiliza peste 60 de funcții standard aflate în biblioteca sistemului. Aceste funcții au prototipul în fișierul *graphics.h*.

În acest capitol indicăm funcțiile mai importante care permit gestiunea ecranului în mod grafic.

ACESTE FUNCȚII FOLΟEȘC PIONTERI DE TIP *far* (32 de biți).

19.1. Setarea modului grafic

Modul grafic se setează cu ajutorul funcției *initgraph*. Aceasta funcție poate fi folosită singură sau împreună cu o altă funcție numită *detectgraph* care determină parametrii adaptorului grafic. Prototipul ei este:

```
void far detectgraph(int far *gd, int far *gm);
```

unde:

- În zona spre care pointeză *gd* se păstrează una din valorile:

Constantă simbolică	Valoare
CGA	1
MCGA	2
EGA	3
EGA64	4
EGAMONO	5
IBM8514	6
HERCMONO	7
ATT400	8
VGA	9
PC3270	10

- În zona spre care pointeză *gm* se memorează una din valorile:

Pentru CGA:

Constantă simbolică	Valoare
CGAC0	0
CGAC1	1
CGAC2	2
CGAC3	3

Toate aceste valori corespund unei rezoluții de 320*200 puncte și permit maximum 4 culori.

CGAHI	4
-------	---

Permite o rezoluție de 640*200 puncte și lucrează numai în alb/negru.

Pentru EGA:

Constantă simbolică	Valoare
EGALO	0
EGAHI	1
EGAMED	2

Are o rezoluție de 640*200 puncte și permite maximum 16 culori.

Are o rezoluție de 640*350 puncte.

Pentru VGA:

Constantă simbolică	Valoare
VGALO	0
VGAMED	1
VGAHI	2

Are o rezoluție de 640*200 puncte.

Are o rezoluție de 640*350 puncte.

Are o rezoluție de 640*480 puncte.

Valorile spre care pointeză *gd* definesc niște funcții standard corespunzătoare adaptorului grafic. Aceste funcții se numesc *drive*. Ele se află în subdirectorul BGI.

Funcția *detectgraph* detectează adaptorul grafic prezent la calculator și păstrează valoarea corespunzătoare acestuia în zona spre care pointeză *gd*.

Modul grafic se definește în aşa fel încît el să fie cel mai performant pentru adaptorul grafic curent.

Cele mai utilizate adaptoare sunt cele de tip EGA și VGA. De aceea, în cele ce urmează, vom presupune că adaptorul curent este de tip EGA. În felul acesta, toate exercițiile din acest capitol pot fi rulate la un calculator echipat cu un adaptor EGA sau VGA.

Apelul funcției *detectgraph* trebuie să fie urmat de apelul funcției *initgraph*. Aceasta setează modul grafic în conformitate cu parametri stabiliți de apelul prealabil al funcției *detectgraph*.

Funcția *initgraph* are prototipul:

```
void far initgraph(int far *gd, int far *gm, int far *cale);
```

unde:

gd și gm

- Sunt pointeri care au aceeași semnificație ca în cazul funcției *detectgraph*.

cale

- Este pointer spre sirul de caractere care definește calea subdirectorului BGI care conține driverele. De exemplu, dacă BGI este subdirector al directorului BORLANDC, atunci vom folosi sirul de caractere:
"c:\\BORLANDC\\BGI"

Exemplu:

Pentru setarea în mod implicit a modului grafic, putem utiliza secvența de mai jos:

```
int driver, mod_grafic;
...
detectgraph(&driver,&mod_grafic);
initgraph(&driver,&mod_grafic,"c:\\BORLANDC\\BGI");
```

După apelul funcției *initgraph* se pot utiliza celelalte funcții standard de gestiune grafică a ecranului.

Din modul grafic se poate ieși apelând funcția *closegraph* de prototip:

```
void far closegraph(void);
```

Funcția *initgraph* poate fi apelată folosind secvența de mai jos:

```
int driver, mod_grafic;
...
driver = DETECT;
initgraph(&driver,&mod_grafic,"c:\\BORLANDC\\BGI");
```

Constanta simbolică DETECT este definită în fișierul *graphics.h* alături de celelalte constante simbolice care definesc driverul. Aceasta are valoarea zero.

Prin apelul de mai sus, funcția *initgraph* apelează funcția *detectgraph* pentru a defini parametrii implicați ai adaptorului grafic.

Utilizatorul poate defini el însuși parametrii pentru inițializarea modului grafic. De exemplu, secvența:

```
int driver = 1; /* CGA */
int mod_graf = 0; /* CGAC0 */
initgraph(&driver,&mod_graf, "c:\\BORLANDC\\BGI");
```

setează modul grafic corespunzător unui adaptor grafic CGA cu rezoluția 320*200 puncte.

În afara acestor funcții, utilizatorul mai poate utiliza și funcția *setgraphmode* care selectează un mod grafic diferit de cel activat implicit prin *initgraph*. Această funcție are prototipul:

```
void far setgraphmode(int mode);
```

unde:

mode

- Are valorile:

- | | |
|-------|-------------|
| 0 - 4 | pentru CGA; |
| 0 - 1 | pentru EGA; |
| 0 - 2 | pentru VGA. |

Ea poate fi utilizată împreună cu funcția *restorecrtmode* de prototip:

```
void far restorecrtmode(void);
```

Această funcție permite revenirea la modul precedent, iar *setgraphmode* realizează trecerea inversă.

Alte funcții din această categorie sunt:

```
void far graphdefaults(void);
```

repune parametrii grafici la valorile implicate;

```
int far getgraphmode(void);
```

returnează codul modului grafic;

```
char *far getmodename(int mod);
```

returnează pointerul spre numele modului grafic definit de codul numeric *mod*;

```
char *far getdrivername(void);
```

returnează pointerul spre numele driverului corespunzător adaptorului curent;

```
void far getmoderange(int grafdrv, int far *min, int far *max);
```

definește valorile minime și maxime ale modului grafic utilizat;

grafdrv are ca valoare una din valorile 1-10 indicate mai sus la funcția *detectgraph*.

Valoarea minimă a modului grafic este păstrată în zona spre care pointează *min*, iar cea maximă în zona spre care pointează *max*.

Exerciții:

19.1 Să se scrie un program care setează modul grafic în două feluri:

- cu ajutorul funcției *detectgraph*;
- fără această funcție.

Programul afișează rezultatele setării.

PROGRAMUL BXIX1

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

main() /*seteaza modul grafic si afiseaza parametrii setarii */
{
    int gd, gm;
    int mod,min,max;
```

```

/* setare prin apelul functiei detectgraph */
detectgraph(&gdriv,&gmod);
printf("valori dupa apelul functiei detectgraph\n");
printf("driver=%d\ntmod grafic=%d\n",gdriv,gmod);
printf("Actionati o tasta pentru a continua\n");
getch();
initgraph(&gdriv,&gmod,"c:\\borlandc\\bgi");
printf("valori dupa initgraph\n");
printf("driver=%d\ntmod grafic=%d\n",gdriv,gmod);
printf("Actionati o tasta pentru a continua\n");
getch();
closegraph();

/* setare fara apelul lui detectgraph */
gdriv=DETECT;
initgraph(&gdriv,&gmod,"c:\\borlandc\\bgi");
printf("initializare fara detectgraph\n");
printf("driver=%d\ntmod grafic=%d\n",gdriv,gmod);
printf("Actionati o tasta pentru a continua\n");
getch();

/* afiseaza numele adaptorului grafic curent */
printf("adaptor grafic: %s\n",getdrivername());
mod=getgraphmode();
printf("cod mod=%d\ntmod grafic=%s\n",mod,getmodename(mod));
getmoderange(gdriv,&min,&max);
printf("domeniul pentru adaptorul grafic=%d,%d\n",min,max);
printf("Actionati o tasta pentru a continua\n");
getch();
closegraph();
}

```

19.2. Gestiunea culorilor

Adptoarele grafice sint prevazute cu o zonă de memorie în care se păstrează date specifice gestiunii ecranului. Aceasta zonă de memorie poartă denumirea de *memorie video*.

În mod grafic, ecranul se consideră format din puncte luminoase numite *pixeli*. Poziția pe ecran a unui pixel se definește printr-un sistem binar:

(x,y)

unde:

- x - Definește coloana în care este afișat pixelul.
- y - Definește linia în care este afișat pixelul.

În cazul adaptoarelor color, unui pixel îi corespunde o culoare.

Culoarea pixelilor se păstrează pe biți în memoria video. Memoria video necesară pentru a păstra starea ecranului setat în mod grafic, se numește *pagina video*. Adptoarele pot conține mai multe pagini video.

Gestiunea culorilor este dependentă de tipul de adaptor grafic existent la

microprocesor.

În cele ce urmează vom avea în vedere adaptoarele grafice de tip EGA/VGA.

În mod concret, ne vom referi la facilitățile oferite de adaptorul EGA, deoarece adaptorul VGA, având performanțe superioare, permite utilizarea acestor facilități.

Numărul maxim al culorilor care pot fi afișate cu ajutorul unui adaptor EGA este de 64.

Culorile se codifică prin numere întregi din intervalul [0,63].

Cele 64 de culori nu pot fi afișate simultan pe ecran. În cazul adaptorului EGA se pot afișa simultan pe ecran cel mult 16 culori. Mulțimea culorilor care pot fi afișate simultan pe ecran se numește *paletă*. Culorile din compoziția unei palete pot fi modificate de utilizator prin intermediul funcțiilor standard. La inițializarea modului grafic se setează o paletă *implicită*.

Paleta se definește cu ajutorul unui tablou de 16 elemente pentru adaptorul EGA. Elementele acestui tablou au valori din intervalul [0,63]. Fiecare element din acest tablou reprezintă codul unei culori.

Codurile culorilor din paleta implicită au denumiri simbolice definite în fișierul *graphics.h*. Ele au prefixul *EGA_*.

În tabela de mai jos se indică codurile culorilor pentru paleta implicită.

Funcțiile de gestiune a culorilor pot avea ca parametri nu numai codurile culorilor, ci și indecsă în tabloul care definește culorile unei palete. De aceea, indicații din intervalul [0,15] pot fi referiți prin constante simbolice definite în fișierul *graphics.h*. Aceste denumiri sugerează culoarea din compunerea paletei.

Tabela paletei implice

indice		codul culorilor	
denumire simbolica	valoare	denumire simbolica	valoare
BLACK	0	EGA_BLACK	0
BLUE	1	EGA_BLUE	1
GREEN	2	EGA_GREEN	2
CYAN	3	EGA_CYAN	3
RED	4	EGA_RED	4
MAGENTA	5	EGA_MAGENTA	5
BROWN	6	EGA_BROWN	20
LIGHTGRAY	7	EGA_LIGHTGRAY	7
DARKGRAY	8	EGA_DARKGRAY	56
LIGHTBLUE	9	EGA_LIGHTBLUE	57
LIGHTGREEN	10	EGA_LIGHTGREEN	58
LIGHTCYAN	11	EGA_LIGHTCYAN	59
LIGHTRED	12	EGA_LIGHTRED	60
LIGHTMAGENTA	13	EGA_LIGHTMAGENTA	61
YELLOW	14	EGA_YELLOW	62
WHITE	15	EGA_WHITE	63

Culoarea fondului (*background*) este totdeauna cea corespunzătoare indicelui zero.

Culoarea pentru desenare (*foreground*) este cea corespunzătoare indicelui 15.

Culoarea de fond poate fi modificată cu ajutorul funcției *setbkcolor*. Aceasta are prototipul:

```
void far setbkcolor(int culoare);
```

unde:

culoare - Este index în tabloul care definește paleta.

De exemplu, dacă se utilizează apelul:

```
setbkcolor(BLUE);
```

atunci culoarea de fond devine albastră.

Pentru a cunoaște culoarea de fond curentă se poate apela funcția *getbkcolor* de prototip:

```
int far getbkcolor(void);
```

Ea returnează indexul în tabloul care definește paleta pentru culoarea de fond.

Culoarea pentru desenare poate fi modificată folosind funcția *setcolor* de prototip:

```
void far setcolor(int culoare);
```

unde:

culoare - Este index în tabloul care definește paleta.

De exemplu, dacă se utilizează apelul:

```
setcolor(YELLOW);
```

atunci culoarea pentru desenare este galbenă.

Culoarea pentru desenare se poate determina apelind funcția *getcolor* de prototip:

```
int far getcolor(void);
```

Ea returnează indexul în tabloul care definește paleta relativ la culoarea pentru desenare.

Paleta curentă poate fi modificată folosind funcțiile *setpalette* și *setallpalette*.

Prima se folosește pentru a modifica o culoare din paleta curentă. Ea are prototipul:

```
void far setpalette(int index, int cod);
```

unde:

index - Este un întreg din intervalul [0,15] și reprezintă indexul în tabloul care definește paleta pentru culoarea care se modifică.

cod - Este un întreg din intervalul [0,63] și reprezintă codul culorii care o înlocuiește în paletă pe cea veche.

De exemplu, apelul:

```
setpalette(DARKGRAY,45);
```

modifică culoarea corespunzătoare indicelui DARKGRAY (adică 8), prin culoarea de cod 45.

Cealaltă funcție permite modificarea simultană a mai multor culori din compunerea paletei. Ea are prototipul:

```
void far setallpalette(struct palettetype far *paleta);
```

unde:

palettetype - Este un tip definit în fișierul *graphics.h* ca mai jos:

```
struct palettetype {  
    unsigned char size;  
    signed char colors[MAXCOLORS + 1];  
};
```

unde:

size - Este dimensiunea paletei.

colors - Este un tablou ale căruia elemente au ca valori codurile culorilor componente ale paletei care se definește.

Modificarea paletei curente cu ajutorul funcțiilor *setpalette* sau *setallpalette* conduce la schimbarea corespunzătoare a culorilor afișate pe ecran în momentul apelului funcțiilor respective.

Pentru a determina codurile culorilor componente ale paletei curente vom folosi funcția *getpalette* de prototip:

```
void far getpalette(struct palettetype far *paleta);
```

Exemplu:

```
struct palettetype pal;  
...  
getpalette(&pal);  
...
```

După apelul funcției *getpalette* se atribuie componentelor structurii datele corespunzătoare din paleta curentă.

Paleta implicită poate fi determinată folosind funcția *getdefaultpalette* de prototip:

```
struct palettetype *far getdefaultpalette(void);
```

Numărul culorilor dintr-o paletă poate fi obținut apelind funcția *getmaxcolor* de prototip:

```
int far getmaxcolor(void);
```

Funcția returnează numărul maxim de culori diminuat cu 1. Deci, în cazul adaptorului EGA funcția returnează valoarea 15.

O altă funcție care determină dimensiunea paletei este funcția *getpalettesize*. Ea are prototipul:

```
int far getpalettesize(void);
```

Functia returneaza numarul colorilor componente ale paletei. In cazul adaptorului EGA functia returneaza valoarea 16.

Exercitii:

19.2 Să se scrie un program care afisează codurile colorilor pentru paleta implicită.

PROGRAMUL BXIX2

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>

main() /* afiseaza codurile colorilor pentru paleta implicita */
{
    int gd=DETECT,gm,i;
    struct palettetype far *pal=(void *)0;

    initgraph(&gd,&gm,"c:\\borland\\bgi");
    pal=getdefaultpalette();
    for(i=0;i<16;i++){
        printf("color[%d]=%d\n",i,pal->colors[i]);
        getch();
    }
    closegraph();
}
```

19.3. Starea ecranului

In mod grafic, ecranul se compune din $n \times m$ puncte luminoase (pixeli). Aceasta inseamnă că pe ecran se pot afisa m linii a n pixeli fiecare.

Pozitia unui pixel se defineste printr-un sistem binar de intregi:

(x,y)

numite coordonatele pixelului. Coordonata x defineste coloana pixelului, iar y defineste linia acestuia.

Pixelul aflat in colțul din stanga sus are coordonatele (0,0).

Coloanele se numerotează de la stanga spre dreapta, iar linile de sus in jos.

Biblioteca grafică a sistemului conține 4 funcții care permit utilizatorului să obțină următoarele informații relativ la ecran:

- coordonata maximă pe orizontală;
- coordonata maximă pe verticală;
- poziția curentă (pixel current).

Prototipurile acestor funcții sunt:

```
int far getmaxx(void);
```

functia returneaza coordonata maximă pe orizontală (abscisa maximă);

```
int far getmaxy(void);
```

functia returneaza coordonata maximă pe verticală (ordonata maximă);

```
int far getx(void);
```

functia returneaza pozitia pe orizontală (abscisa) a pixelului curent;

```
int far gety(void);
```

functia returneaza pozitia pe verticală (ordonata) a pixelului curent.

Exercitii:

19.3 Să se scrie un program care afisează următoarele informații:

- culoarea fondului;
- culoarea pentru desenare;
- coordonatele maxime;
- coordonatele pixelului curent.

PROGRAMUL BXIX3

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>

main() /* afiseaza:
         - culoarea fondului;
         - culoarea pentru desenare;
         - coordonatele maxime;
         - coordonatele pixelului curent. */
{
    int gd=DETECT,gm,culf,culd,crtx,cry,maxx,maxy;

    initgraph(&gd,&gm,"c:\\borland\\bgi");
    culf=getbkcolor();
    culd=getcolor();
    maxx=getmaxx();
    maxy=getmaxy();
    crtX=getx();
    cry=gety();
    closegraph();
    printf("culoarea fondului=%d\n",culf);
    printf("culoarea pentru desenare=%d\n",culd);
    printf("abscisa maxima=%d\n",maxx);
    printf("ordonata maxima=%d\n",maxy);
    printf("abscisa curenta=%d\n",crtX);
    printf("ordonata curenta=%d\n",cry);
    printf("Actionati o tasta pentru a continua\n");
    getch();
}
```

```

    closegraph();
}

```

19.4. Gestiunea textelor

Afișarea textelor presupune definirea unor parametri care pot fi gestionati prin funcțiile descrise mai jos.

În mod grafic dispunem de mai multe seturi de caractere. Setul de caractere se alege prin intermediul parametrului numit *font*. Pentru acest parametru se pot utiliza următoarele valori:

Constantă simbolică	Valoare
DEFAULT_FONT	0
TRIPLEX_FONT	1
SMALL_FONT	2
SANS_SERIF_FONT	3
GOTHIC_FONT	4

Alți parametri utilizati în definirea caracterelor sunt:

- înălțimea și lățimea caracterelor;
- direcția de scriere a caracterelor:
 - de la stînga la dreapta: HORIZ_DIR;
 - de jos în sus: VERT_DIR;
- cadrul caracterelor față de poziția curentă:
 - pe orizontală; poziția curentă se află:
 - » în stînga: LEFT_TEXT;
 - » în centru: CENTER_TEXT;
 - » în dreapta: RIGHT_TEXT.
 - pe verticală; poziția curentă se află în:
 - » marginea inferioară: BOTTOM_TEXT;
 - » centru: CENTER_TEXT;
 - » marginea superioară: TOP_TEXT.

Acești parametri se setează cu ajutorul a două funcții *settextstyle* și *settextjustify*. Prima are prototipul:

```
void far settextstyle(int font, int direction, int charsize);
```

unde:

- | | |
|------------------|---|
| <i>font</i> | - Definește setul de caractere. |
| <i>direction</i> | - Definește direcția de scriere a textului. |
| <i>charsize</i> | - Definește dimensiunea caracterului în pixeli. |

Dimensiunea se definește astfel:

Valoarea parametrului	Matricea folosită pentru afișarea caracterului (in pixeli)
1	8*8
2	16*16
3	24*24
...	...
10	80*80

Dimensiunea poate fi stabilită de utilizator folosind funcția *setusercharsize*. În acest caz, parametrul *charsize* are valoarea zero.

Cea dea două funcție definește cadrul textului. Ea are prototipul:

```
void far settextjustify(int oriz, int vert);
```

unde:

- | | |
|-------------|-----------------------------------|
| <i>oriz</i> | - Definește cadrul pe orizontală; |
| <i>vert</i> | - Definește cadrul pe verticală. |

Valorile acestor parametri pot fi determinate cu ajutorul funcției *gettextsettings* de prototip:

```
void far gettextsettings(struct textsettingstype far *textinfo);
```

Tipul *textsettingstype* este definit în fișierul *graphics.h* astfel:

```
struct textsettingstype {
    int font;
    int direction;
    int charsize;
    int horiz;
    int vert;
};
```

Funcția *gettextsettings* atribuie componentelor structurii de tip *textsettingstype* valorile curente.

Amintim valorile numerice ale constantelor simbolice indicate mai sus.

Constantă simbolică	Valoare
LEFT_TEXT	0
CENTER_TEXT	1
RIGHT_TEXT	2
BOTTOM_TEXT	0
TOP_TEXT	2
HORIZ_DIR	0
VERT_DIR	1

După setarea parametrilor de mai sus se pot afișa texte folosind funcțiile *outtext* și *outtextxy*. Prima afișează textul începând cu poziția curentă de pe ecran. Cea de a doua funcție permite afișarea textului începând cu un punct al căruia coordonate sint definite prin primii doi parametri efectivi ai funcției.

Funcțiile afișează caractere colorate folosind culoarea de desenare curentă.

Funcția *outtext* are prototipul:

```
void far outtext(char far *sir);
```

unde:

- sir - Este pointer spre o zona de memorie in care se păstreaza caracterele de afisat.

Se afiseaza caracterele respective pina la intilnirea caracterului NUL.

Funcția *outtextxy* are prototipul:

```
void far outtextxy(int x, int y, char far *sir);
```

unde:

- (x,y) - Definește poziția punctului de pe ecran începind cu care se afiseaza textul.

- sir - Are aceeași semnificație ca în cazul funcției *outtext*.

Dimensiunile in pixeli a unui șir de caractere se pot determina folosind funcțiile *textheight* și *textwidth*. Acestea au prototipurile:

```
int far textheight(char far *sir);
```

funcția returneaza inalțimea in pixeli a șirului pastrat in zona spre care pointeaza *sir*:

```
int far textwidth(char far *sir);
```

funcția returneaza lațimea in pixeli a șirului aflat in zona spre care pointeaza *sir*.

Utilizatorul poate defini dimensiunea caracterelor apelind funcția *setusercharsize*. Ea are prototipul:

```
void far setusercharsize(int multx, int divx, int multy, int divy);
```

Dimensiunea caracterelor se definește prin inmulțirea lațimii lor cu multx/divx și a inalțimii cu multy/divy.

Modificarea dimensiunii caracterelor in acest mod este posibila pentru fonturile diferite de fontul *DEFAULT_FONT*.

Observație:

Funcția *printf* poate fi folosita pentru a afișa caractere in mod obișnuit. Ea ignora parametrii indicați mai sus.

Exerciții:

- 19.4 Să se scrie un program care afișează texte folosind toate cele 5 fonturi, caracterele avind pe rînd dimensiunile 1, 2, 3 și 4. În cazul fonturilor diferite de *DEFAULT_FONT*, se vor afișa texte ale caror caractere se vor afla in rapoartele:

- 4/3 in lațime;
- 2/1 in inalțime.

PROGRAMUL BXIX4

```
#include <graphics.h>
#include <conio.h>
#include <stdio.h>
#include <string.h>

main() /* afiseaza texte cu diferite fonturi si dimensiuni */
{
    int gdriver=DETECT, gmod;
    char *denfont[] = {
        "Val=0 DEFAULT_FONT",
        "Val=1 TRIPLEX_FONT",
        "Val=2 SMALL_FONT",
        "Val=3 SANS_SERIF_FONT",
        "Val=4 GOTHIC_FONT"
    };
    int stil,x=0,y=0;
    int dim;
    char dimensiune[30];

    for(dim=1;dim<5;dim++){
        y=0;
        initgraph(&gdriver,&gmod, "c:\\borlandc\\bgi");
        setTextStyle(stil,HORIZ_DIR,dim);
        sprintf(dimensiune,"dim=%d font:",dim);
        outtextxy(x,y,dimensiune);
        x += textwidth(dimensiune); /* avans la coloana libera de pe aceiasi linie */
        outtextxy(x,y,denfont[stil]);
        y += textheight(denfont[stil]); /* avans la inceputul liniei urmatoare */
        x=0;
        /* se defineste dimensiunea de catre utilizator raport: 4/3 in latime; 2/1 in inaltime. */
        if(stil!=DEFAULT_FONT){
            setusercharsize(4,3,2,1); /* defineste raporturile pentru dimensiunile caracterelor */
            strcpy(dimensiune,"dim utilizator:4/3,2/1");
            outtextxy(x,y,dimensiune);
            y+=textheight(dimensiune);
            x=0;
        }
        getch();
    }
    closegraph();
    printf("Actionati o tasta pentru a continua\n");
    getch();
}
```

- 19.5 Să se scrie un program care afișează texte cadratice in toate variantele definite de funcția *settextjustify*.

PROGRAMUL BXIX5

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

main() /* afiseaza texte cadrate in toate variantele definite de functia settextjustify */
{
    int gdriver=DETECT,gmod;
    char *hjust[]={"LEFT_TEXT","CENTER_TEXT","RIGHT_TEXT",};
    char *vjust[]={"BOTTOM_TEXT","CENTER_TEXT","TOP_TEXT"};
    int x=200,y=100,hj,vj;

    for(hj=LEFT_TEXT;hj<=RIGHT_TEXT;hj++) {
        initgraph(&gdriver,&gmod,"c:\\borlandc\\bgi");
        outtextxy(x,y,hjust[hj]);
        y+=textheight(hjust[hj])+8;
        for(vj=BOTTOM_TEXT;vj<=TOP_TEXT;vj++) {
            settextjustify(hj,vj);
            outtextxy(x,y,"N");
            x+=textwidth("N");
            outtextxy(x+100*vj+100,y,vjust[vj]);
            getch();
        }
        closegraph();
    }
}
```

19.5. Gestiunea imaginilor

În paragrafele precedente s-a arătat că ecranul în mod grafic se compune din $n \times m$ puncte luminoase numite *pixeli*. Un pixel are o poziție definită prin coordonatele sale și este colorat în cazul adaptoarelor color.

La adaptoarele monocrom pixelul are o nuanță de gri.

Utilizatorul poate afișa pe ecran un pixel cu ajutorul funcției *putpixel*. Aceasta are prototipul:

```
void far putpixel(int x, int y, int culoare);
```

unde:

- (x,y) - Defineste pozitia punctului.
culoare - Defineste culoarea punctului.
- Este un intreg din intervalul [0,15] si reprezinta indicele culorii in tabela care defineste paleta curenta.

Functia *getpixel* permite stabilirea culorii unui pixel afisat pe ecran. Ea are prototipul:

```
unsigned far getpixel(int x, int y);
```

unde:

(x,y) - Definește poziția punctului.

Funcția returnează un întreg din intervalul [0,15]. Acesta definește culoarea pixelului de coordonate (x,y), fiind index în tabloul care definește paleta curentă.

Ecranul poate fi partajat în mai multe părți care pot fi gestionate independent. Aceste părți le vom numi *ferestre grafice*.

În continuare, prin ferestru vom înțelege o fereastră grafică. O fereastră se definește cu ajutorul funcției *setviewport* de prototip:

```
void far setviewport(int st, int sus, int dr, int jos, int d);
```

unde:

- (st,sus) - Sunt coordonatele colțului stînga sus al ferestrei.
(dr,jos) - Sunt coordonatele colțului dreapta jos al ferestrei.
d - Indicator cu privire la decuparea desenului (vezi mai jos).

Fereastra definită în urma apelului funcției *setviewport* devine fereastra activă. Inițial (imediat după setarea modului grafic), fereastra activă este tot ecranul.

O fereastră activă se poate șterge cu ajutorul funcției *clearviewport*. Ea are prototipul:

```
void far clearviewport(void);
```

După apelul funcției *clearviewport*, toți pixelii ferestrei active au aceeași culoare și anume culoarea de fond curentă.

Pozitia curentă după apelul funcției *clearviewport* este pixelul de coordonate relative (0,0), adică chiar colțul din stînga sus al ferestrei.

O altă funcție utilizată pentru a șterge tot ecranul este funcția *cleardevice*. Ea are prototipul:

```
void far cleardevice(void);
```

După apelul funcției *cleardevice* se șterge tot ecranul și pixelul curent devine cel din colțul stînga sus al ecranului.

Parametri ferestrei active se pot determina apelind funcția *getviewsettings*. Aceasta are prototipul:

```
void far getviewsettings(struct viewporttype far *fereastra);
```

unde:

- viewporttype - Este un tip definit în fișierul *graphics.h* astfel:
struct viewporttype {
 int left;
 int top;
 int right;
 int bottom;
 int clip;
};

Dupa apelul funcției `getviewsetting`, la componentele structurii de tip `viewporttype` de la apel li se atribuie valorile corespunzatoare ale ferestrei active.

Parametrul `clip` are două valori:

`CLIP_ON` (valoarea 1)

și

`CLIP_OFF` (valoarea zero).

Daca `clip` are valoarea 1, atunci funcțiile de afișare a textelor și de desenare nu pot scrie sau desena în afara limitelor ferestrei active. În caz contrar, se pot depăși limitele ferestrei active. Deci textele și figurile care nu încap în fereastra activă se trunchiază dacă:

`clip = CLIP_ON.`

Imaginea ecranului se păstrează în memoria video a adaptorului grafic și formează o pagină. În cazul adaptoarelor de tip EGA/VGA, adaptorul dispune de o memorie video capabilă să memoreze 8 pagini. Acestea se numerotează de la 0 la 7.

Funcțiile de desenare și scriere de texte acționează asupra unei singure pagini. Aceasta se numește pagina activă. Utilizatorul poate activa o pagină folosind funcția `setactivepage` de prototip:

```
void far setactivepage(int nrapag);
```

unde:

`nrapag` - Este numărul paginii care se activează.

De obicei, pagina activă este vizualizată pe ecran. Cu toate acestea, programatorul are posibilitatea să vizualizeze o altă pagină decit cea activă. Aceasta se realizează utilizând funcția `setvisualpage` de prototip:

```
void far setvisualpage(int nrapag);
```

unde:

`nrapag` - Este numărul paginii care se vizualizează.

Această funcție poate fi utilă pentru animație.

Imaginea unei zone dreptunghiulare de pe ecran poate fi salvată în memorie folosind funcția `getimage`. Ea are prototipul:

```
void far getimage(int st, int sus, int dr, int jos, void far *zt);
```

unde:

`(st,sus)` - Definește coordonatele colțului stînga sus a zonei de pe ecran care se salvează.

`(dr,jos)` - Definește coordonatele colțului dreapta jos a zonei de pe ecran care se salvează.

`zt` - Pointer spre zona de memorie în care se salvează imaginea de pe ecran.

Dimensiunea zonei de memorie spre care pointeză `zt` trebuie să fie suficient de mare pentru a putea salva datele care definesc imaginea de pe ecran, care se salvează. Această dimensiune se poate determina folosind funcția `imagesize` de prototip:

```
unsigned far imagesize(int st, int sus, int dr, int jos);
```

unde:

`(st,sus)`

- Definește coordonatele colțului din stînga sus a zonei dreptunghiulare de pe ecran.

`(dr,jos)`

- Definește coordonatele colțului din dreapta jos a zonei dreptunghiulare de pe ecran.

Funcția `imagesize` se apelează înainte de a apela funcția `getimage` pentru a stabili dimensiunea zonei de memorie necesară pentru a salva o imagine dreptunghiulară de pe ecran. Zona de memorie respectivă se poate rezerva în memoria heap îninind seama de valoarea returnată de funcția `imagesize`.

Imaginea de pe ecran salvată cu ajutorul funcției `getimage`, poate fi afișată pe ecran în orice parte a acestuia, cu ajutorul funcției `putimage`. Cu această ocazie se pot face anumite operații asupra datelor care definesc imaginea. Funcția are prototipul:

```
void far putimage(int st, int sus, void far *zt, int op);
```

unde:

`(st,sus)`

- Definește coordonatele colțului stînga sus a zonei de pe ecran în care se afișează imaginea.

`zt`

- Pointer spre zona în care se păstrează datele care formează imaginea de afișat.

`op`

- Aceste date au fost păstrate în zona respectivă prin intermediul funcției `getimage`.

`op`

- Definește operația între datele aflate în zona spre care pointeză `zt` și cele existente pe ecran în zona dreptunghiulară definită de parametri `st, sus` (colțul din stînga sus; colțul din dreapta jos rezultă din datele existente în zona spre care pointeză `zt`).

Parametrul `op` se definește ca mai jos:

Constantă simbolică	Valoare	Aceiune
COPY_PUT	0	copiază imaginea din memorie pe ecran
XOR_PUT	1	"sau exclusiv" între datele de pe ecran și cele aflate în memorie
OR_PUT	2	"sau" între datele de pe ecran și cele din memorie
AND_PUT	3	"și" între datele de pe ecran și cele din memorie

(continuare)

Constantă simbolică	Valoare	Acțiune
NOT_PUT	4	copiază imaginea din memorie pe ecran complementând datele aflate în memorie.

Pentru a defini pixelul curent se utilizează funcția *moveto*. Aceasta are prototipul:

void far moveto(int x, int y);

După apelul funcției *moveto*, pixelul curent devine cel de coordonate (x,y). O funcție înrudită cu aceasta este funcția *moverel*. Aceasta are prototipul:

void far moverel(int dx, int dy);

Dacă notăm cu (x,y) coordonatele pixelului curent, atunci după apelul funcției *moverel*, pixelul curent are coordonatele:

(x+dx, y+dy)

Observație:

Majoritatea funcțiilor care au ca parametri coordonate de pixel, interpretează aceste coordonate ca fiind relative față de fereastra activă al cărui colț stanga sus sunt coordonatele(0,0). Așa de exemplu, funcțiile:

outtextxy, putpixel, getpixel, moveto

au ca parametri coordonate relative la pixelul din colțul stanga sus al ferestrei active.

Funcțiile *getx* și *gety* returnează abscisa, respectiv ordonata, relative la pixelul din colțul stanga sus al ferestrei active.

În schimb funcțiile *getmaxx* și *getmaxy* returnează totdeauna abscisa, respectiv ordonata, maximă pentru tot ecranul în conformitate cu modul grafic.

De asemenea, funcțiile *setviewport* și *getviewsettings* gestionează coordonate care sunt relative față de colțul stanga sus al ecranului și nu față de fereastra activă. Astfel de coordonate le vom numi în continuare *absolute*.

Exerciții:

19.6 Să se scrie un program care realizează următoarele:

- Afișează, într-o zonă de dimensiune 50*50, pixeli colorați folosind toate culorile din paletă.
Zona se află în colțul din stanga sus al ecranului.
- Definește fereastra de coordonate:
(60,0) - colțul stanga sus;
(120,60) - colțul dreapta jos.
Afișează în fereastra activă pixeli colorați folosind toate culorile din paletă.

- c. Definește fereastra de coordonate:

(20,100) - colțul stanga sus;
(100,180) - colțul dreapta jos.

Afișează în fereastra activă pixeli colorați folosind toate culorile din paletă.
Afișează în fereastra activă textul "abcdefg" începând cu poziția de coordonate(1,1).

Se utilizează culoarea de index 6.

- d. Se șterge fereastra activă.

Se afișează textul "abcdefg" începând cu poziția curentă din fereastră și folosind culoarea de index 14.

- e. Se șterge tot ecranul.

- f. Se revine din modul grafic.

După realizarea fiecărui punct de mai sus se vizualizează ecranul apelind funcția *getch*. Pentru a continua, se acționează o tastă oricare.

PROGRAMUL BXIX6

```
#include <graphics.h>
#include <conio.h>

main() /* - afișează în trei zone ale ecranului pixeli colorați folosind toate culorile paletei;
          - în una din zone se afișează și textul "abcdefg";
          - în final se sterg zonele afișate. */
{
    int gd=DETECT,gm;
    int i,j,c;

    initgraph(&gd,&gm, "c:\\borlandc\\bgi");

    /* zona din colțul stanga sus al ecranului */
    for(i=0;i<50;i++){
        c=i;
        for(j=0;j<50;j++,c++) {
            c=c%16;
            putpixel(i,j,c);
        }
    }
    getch();

    /* definește o fereastra și afișează în ea pixeli colorați */
    setviewport(60,0,120,60,1);
    for(i=0;i<50;i++){
        c=0;
        for(j=0;j<50;j++,c++) {
            c=c%16;
            putpixel(i,j,c);
        }
    }
    getch();
}
```

```

/* definește ultima zonă și afișează în ca pixeli colorați */
setviewport(20,100,100,180,1);
for(i=0;i<40;i++){
    c=10;
    for(j=0;j<250;j++,c++) {
        c=c&16;
        putpixel(i,j,c);
    }
}

/* afișează textul "abcdefg" */
setcolor(6);
outtextxy(1,1,"abcdefg");
getch();

/* sterge ultima zonă și apoi afișează același text folosind culoarea de index 14 */
clearviewport();
setcolor(14);
outtext("abcdefg");
getch();

/* sterge tot ecranul */
cleardevice();
getch();

/* ieșire din modul grafic */
closegraph();
getch();
}

```

19.7 Să se scrie un program care realizează următoarele:

- Afișează un pixel în punctul de coordonate absolute (20,30), apoi textul:
(x,y)
unde x și y sunt coordonatele returnate de funcțiile `getx` și respectiv `gety`.
- Se deplasează poziția curentă cu valoarea relativă 30, atât pe abscisă, cât și pe ordonată, apoi se afișează pixelul curent și textul:
(x,y)
unde x și y sunt valorile returnate de funcțiile `getx` și respectiv `gety`.
- Se definește fereastra de coordonate:
(200,0,420,70);
apoi se repetă secvențele a-b de mai sus.
- Se sterg fereastra activă.
- Se sterg ecranul și se ieșe din modul grafic.

Culoarea pentru afișare este culoarea de index maxim.

PROGRAMUL BXIX7

```

#include <graphics.h>
#include <conio.h>
#include <stdio.h>

```

```

main() /* - afișează:
        a- un pixel în punctul de coordonate absolute (20,30), apoi textul
            (x,y)
            unde x și y sunt coordonatele returnate de getx și respectiv gety;
        b- se deplasează poziția curentă cu valoarea relativă 30 atât pe abscisă
            cât și pe ordonată, apoi se afișează pixelul curent și textul
            (x,y)
            unde x și y se definesc ca mai sus;
        c- se definește fereastra
            200,0,420,70
            și se repetă secvențele a-b de mai sus;
            se sterg afișările și apoi se ieșe din modul grafic . */

{
    int gd=DETECT,gm;
    int x,y,i;
    char msg[80];

    initgraph(&gd,&gm, "c:\\borlandc\\bgi");
    for(i=0;i<2;i++)
        moveto(20,30);
    putpixel(getx(),gety(),getmaxcolor());
    sprintf(msg, "%d %d", getx(), gety());
    outtext(msg);

    /* deplasare relativă față de poziția curentă */
    moverel(30,30);
    putpixel(getx(),gety(),getmaxcolor());
    sprintf(msg, "%d %d", getx(), gety());
    outtext(msg);
    getch();
    setviewport(200,0,420,70,1);

    clearviewport();
    getch();
    cleardevice();
    getch();
    closegraph();
}

```

19.8 Să se scrie un program care realizează următoarele:

- Afișează pixeli colorați într-o zonă dreptunghiulară în colțul din stînga sus al ecranului, apoi o afișează pe ecran în zone ce au poziții definite aleator.
- La afișarea imaginii se folosesc toate operațiile dintre imagini oferite de funcția `putimage`.
- Culoarea de fond se schimbă folosind toate culorile din paletă.

După afișările indicate mai sus execuția se poate termina actionând tasta zero; orice altă tastă acționată permite continuarea programului cu un nou set de imagini afișate aleator.

Se observă efectul operațiilor dintre imagini cînd acestea se suprapun.

PROGRAMUL BXIX8

```
#include <graphics.h>
#include <conio.h>
#include <alloc.h>
#include <stdlib.h>
#include <dos.h>
#include <stdio.h>

main() /* - afiseaza pixeli colorati intr-o zona dreptunghiulara in coltul din stanga sus al ecranului, apoi afiseaza imaginea respectiva pe ecran in zone ce au pozitii definite aleator;
         - la afisarea imaginii se folosesc toate operatiile oferite de functia putimage;
         - de asemenea, se schimba culoarea de fond folosind toate culorile din paleta. */
{
    int gd=DETECT,gm;
    int i,j,c,k;
    unsigned dim;
    void far *buf;
    struct time t;
    int rx,ry,x,y;

    initgraph(&gd,&gm, "c:\\borlandc\\bgi");
    /* afiseaza pixeli colorati */
    for(i=0;i<50;i++){
        c=i;
        for(j=0;j<50;j++,c++) {
            c=c%16;
            putpixel(i,j,c);
        }
    }
    getch();
    /* salveaza zona afisata pe ecran */
    dim=imagesize(0,0,50,50);
    if((buf=farmalloc(dim))==0){
        closegraph();
        printf("Memorie insuficienta\n");
        exit(1);
    }
    getimage(0,0,50,50,buf);

    /* seteaza saminta */
    gettime(&t);
    srand(t.ti_hour*3600L + t.ti_min*60+t.ti_sec);

    /* se determina valorile maxime pentru abscisa si ordonata coltului sting al imaginilor */
    x=getmaxx()-50;
    y=getmaxy()-50;

    /* afisarea aleatoare a imaginilor */
    do{
        for(k=0;k<16;k++) {
            setbkcolor(k);
```

```
for(i=COPY_PUT;i<=NOT_PUT;i++) {
    rx=random(x);
    ry=random(y);
    setviewport(0,0,getmaxx(),getmaxy(),1);
    putimage(rx,ry,buf,i);
    getch();
}
setcolor(1); /* culoarea pentru afisarea textului in partea de jos a ecranului */
setviewport(1,340,639,349,1); /* fereastra pentru text */
outtextxy(0,0,"Pentru a termina tastati zero; se continua cu orice alta tasta");
if(getch()=='0') break;
clearviewport(); /* sterge textul afisat */
}while(1);
closegraph();
}
```

19.6. Tratarea erorilor

Erorile survenite în gestiunea în mod grafic a ecranului pot fi puse în evidență cu ajutorul funcției *graphresult*. Ea are prototipul:

```
int far graphresult(void);
```

Funcția returnează codul ultimei erori care a apărut înaintea apelului funcției *graphresult* sau valoarea constantei simbolice *grOk* dacă nu au fost erori.

Constanta *grOk* este definită în fișierul *graphics.h*.

Mesajul de eroare poate fi decodificat cu ajutorul funcției *grapherrmsg*. Aceasta are prototipul:

```
char far *far grapherrmsg(int coderoare);
```

unde:

coderoare — Este valoarea returnată de funcția *graphresult*.

Funcția *grapherrmsg* returnează un pointer spre textul de eroare.

Constanta *grOk* are valoarea zero. Codurile de eroare au valori negative.

Exerciții:

19.9 Să se scrie un program care afișează parametri implicați ai modului grafic setat prin apelul funcției *initgraph*:

- numele adaptorului grafic;
- numele modului grafic;
- rezoluția;
- fereastra curentă;
- decupajul;
- poziția curentă;
- numărul culorilor disponibile;

- culoarea curentă;
- setul de caractere;
- direcția textului;
- dimensiunea caracterelor;
- cadrul textelor.

PROGRAMUL BXIX9

```

#include <conio.h>
#include <graphics.h>
#include <stdio.h>
#include <stdlib.h>

int grafdriver, grafmod;
struct palettetype paleta;
int maxculori;
int xmax, ymax;
int posx, posy;
char *Fonts[]={"DefaultFont", "TriplexFont",
               "SmallFont", "SansSerifFont", "GothicFont"};
char *TextDirect[]={"HorizDir", "VertDir"};
char *HorizJust[]={"LeftText", "CenterText", "RightText"};
char *VertJust[]={"BottomText", "CenterText", "TopText"};

void inimodgrafic() /* initializeaza modul grafic */
{
    int eroare;

    grafdriver=DETECT;
    initgraph( &grafdriver, &grafmod, "c:\\borland\\bgi" );
    eroare=graphresult();
    if(eroare!= grOk){
        printf ("eroare la initializarea modului grafic: %s\n",
               grapherrmsg(eroare));
        exit(1);
    }
    getpalette(&paleta);
    maxculori=getmaxcolor()+1;
    xmax=get maxx();
    ymax=get maxy();
    posx=get x();
    posy=get y();
}

void fereastraprincipala(char *antet)
/* afisaza antetul in fereastra principala */
{
    int inalt;
    /* sterge ecranul */
    cleardevice();
    /* setaza fereastra principala pe tot ecranul */
    setviewport (0, 0, xmax,ymax, 1);

    /* determina inaltimea textului */
    inalt=textheight ("H");

    /* afisarea textului */
    outtextxy(xmax/2, 2, antet );

    /* setaza o fereastra pentru a continua afisarea */
    setviewport(1, inalt+5, xmax-1,ymax-(inalt+ 5),1);
}

void afisinit()
/* afiseaza datele de la initializarea modului grafic */
{
    struct viewporttype infview;
    struct textsettingstype inftext;
    char *driver, *mod;
    int x,y;
    char sir[130];

    getviewsettings(&infview);

    gettextsettings(&inftext);
    driver=getdrivername();
    mod=getmodename(grafmod);

    x=10;
    y=4;

    /* se scrie textul rezultatelor initializarii */
    fereastraprincipala("Rezultatele initializarii");

    /* cadrul textului */
    settextjustify(LEFT_TEXT, TOP_TEXT);

    /* afisaza adaptorul grafic */
    sprintf(sir, "adaptorul grafic: %-20s(%d)", driver, grafdriver);
    outtextxy(x,y,sir);
    y+=8;

    /* afisaza modul grafic */
    sprintf(sir, "modul grafic: %-20s (%d)", mod,grafmod);
    outtextxy(x,y,sir);
    y+=8;

    /* afisaza rezolutia */
    sprintf(sir, "rezolutia: (0,0,%d,%d)", xmax, ymax);
    outtextxy( x,y,sir);
    y+=8;

    /* fereastra grafica curenta */
    sprintf(sir, "fereastra curenta:(%d, %d, %d, %d)",
```

```

infview.left, infview.top,
infview.right, infview.bottom);
outtextxy(x,y,sir);
y+=8;

sprintf(sir,"decupaj: %s",infview.clip ? "ON" : "OFF");
outtextxy(x,y, sir );
y+=8;
/* pozitia curenta */
sprintf(sir, "pozitia curenta:( %d, %d )",posx,posy);
outtextxy( x, y, sir );
y+=8;

/* numar culori disponibile */
sprintf(sir, "numar culori: %d", maxculori);
outtextxy(x, y, sir );
y+=8;

/* culoarea curenta */
sprintf(sir, "culoarea curenta: %d",getcolor());
outtextxy(x, y, sir );
y+=8;

/* setul de caractere */
sprintf(sir, "setul de caractere: %s",Fonts[inftext.font]);
outtextxy(x,y,sir);
y+=8;

/* directia textului */
sprintf (sir,"directia textului: %s",
TextDirect[inftext.direction]);
outtextxy(x,y,sir);
y+=8;

/* dimensiunea caracterului */
sprintf (sir,"dimensiunea caracterului: %d",inftext.charsize);
outtextxy(x,y,sir);
y+=8;

/* cadraj orizontal */
sprintf(sir, "cadraj orizontal: %s",HorizJust[inftext.horiz]);
outtextxy(x,y,sir);
y+=8;

/* cadraj vertical */
sprintf(sir, "cadraj vertical: %s",VertJust[inftext.vert]);
outtextxy(x,y,sir);
}

main() /* afiseaza starea curenta a unor parametri ai modului grafic */
{
    inimodgrafic ();
    afisajit();
    outtextxy(8,ymax-50,"Actionati o tasta pentru a termina");
}

```

```

getch ();
closegraph();
}

```

19.7. Desenare și colorare

Biblioteca standard a sistemului pune la dispoziția utilizatorului o serie de funcții care permit desenarea și colorarea unor figuri geometrice.

Amintim că un punct colorat (pixel) se afișază cu ajutorul funcției *putpixel* de prototip (vezi paragraful 19.5):

```
void far putpixel(int x, int y, int culoare);
```

unde:

(x,y)

- Sunt coordonatele punctului care se afișează și ele sunt relative la fereastra activă.

culoare

- Este index în tabloul care definește paleta curentă.

Indexul respectiv definește codul culorii pentru afișarea pixelului pe ecran.

Menționăm că în acest paragraf prin parametrul *culoare* vom înțelege un index în tabloul care definește paleta curentă.

Acest index definește codul culorii pentru desenarea și colorarea figurilor.

Pentru trasarea linilor se pot folosi trei funcții: *line*, *lineto* și *linerel*.

Funcția *line* are prototipul:

```
void far line(int xstart, int ystart, int xfin, int yfin);
```

Funcția trasează un segment de dreaptă ale cărui capete sunt punctele de coordonate:

(xstart,ystart)

și

(xfin,yfin).

Funcția *lineto* are prototipul:

```
void far lineto(int x, int y);
```

Ea trasează un segment de dreaptă care are ca origine poziția curentă, iar ca și punct final cel de coordonate (x,y).

Punctul final devine poziția curentă.

Amintim că funcția *moveto* permite definirea poziției curente (vezi paragraful 19.5).

Cea de a treia funcție utilizată la trasarea dreptelor are prototipul:

```
void far linerel(int x, int y);
```

Dacă notăm cu *x crt* și *y crt* coordonatele poziției curente, atunci funcția *linerel* trasează un segment de dreaptă ale cărui capete sunt punctele de coordonate:

(*x crt*,*y crt*)

și
(xcentru+x,ycentru+y).

Alte funcții care permit trăsuri de figuri geometrice utilizate frecvent sunt:

void far arc(int xcentru, int ycentru, int unghistart, int unghifin, int raza);
trasează un arc de cer; unghiurile sint exprimate în grade sexagesimale.

void far circle(int xcentru, int ycentru, int raza);

trasează un cer; (xcentru,ycentru) sint coordonatele centrului arcului de cer și respectiv cercului trasat de aceste funcții; parametrul *raza* definește mărimea razei curbelor respective.

**void far ellipse(int xcentru, int ycentru,
int unghistart, int unghifin,
int semiaxamare, int semiaxamica);**

trasează un arc de elipsă cu centrul în punctul de coordonate (xcentru, ycentru) avind semiaxa mare definită de parametrul *semiaxamare*, iar semiaxa mică de parametrul *semiaxamica*.

void far rectangle(int st, int sus, int dr, int jos);

trasează un dreptunghi definit de colțurile sale opuse:

- (*st,sus*) - Colțul din stînga sus.
- (*dr,jos*) - Colțul din dreapta jos.

void far drawpoly(int nr, int far *tabpct);

trasează o linie poligonala:

- nr* - Numarul laturilor.
- tabpct* - Este un pointer spre intregi care definesc coordonatele virfurilor liniei poligonale.
- Acestea sint pastrate sub forma: *abscisa_i,ordonata_i* unde *i* are valorile 1, 2, ..., *nr*+1.

Linia poligonala este inchisă dacă primul punct coincide cu ultimul (au același coordonate).

Coordonatele utilizate ca parametri la apelul acestor funcții sunt relative la fereastra activă.

Culoarea de trăsare este cea curentă.

La trăsarea linilor cu ajutorul funcțiilor:

line, lineto și *linerel*

se poate defini un stil de trăsare folosind funcția *setlinestyle*. Această funcție are prototipul:

void far setlinestyle(int stil, unsigned sablon, int grosime);

unde:

stil

- Este întreg din intervalul [0,4] care definește stilul liniei conform tabelei de mai jos:

Constantă simbolică	Valoare	Stil
SOLID_LINE	0	linie continuă
DOTTED_LINE	1	linie punctată
CENTER_LINE	2	linie intreruptă formată din liniuțe de două dimensiuni
DASHED_LINE	3	linie intreruptă formată din liniuțe de același dimensiune
USERBIT_LINE	4	stil definit de utilizator prin şablon

sablon

- Definește stilul liniei.
- Are sens numai cînd primul parametru (stil) are valoarea 4.
- În rest este neglijat și de aceea poate avea valoarea zero.

grosime

- Definește lățimea liniei în pixeli; pot fi două lățimi:
NORM_WIDTH (valoarea 1 pixel)
- și
- THICK_WIDTH (valoarea 3 pixeli).

Stilul curent poate fi determinat apelind funcția *getlinesettings* de prototip:

void far getlinesettings(struct linesettingstype far *linieinfo);

unde:

linesettingstype - Este definit în fișierul *graphics.h* astfel:

```
struct linesettingstype {
    int linestyle;
    unsigned upattern;
    int thickness;
};
```

După apelul funcției *getlinesettings*, componentelor structurii de tip *linesettingstype* de la apel li se atribuie valorile curente ale stilului de trăsare după cum urmează:

linestyle - Are ca valoare stilul definit mai sus.

upattern - Are ca valoare şablonul definit de utilizator. Această valoare are semnificație numai dacă *linestyle* are valoarea 4.

thickness - Are valoarea 1 sau 3 și reprezintă grosimea de trăsare a dreptelor prin funcțiile *line*, *lineto* sau *linerel*.

Din cele de mai sus rezultă că pentru trăsarea linilor se pot folosi 4 stiluri *standard* (definite de valorile SOLID_LINE, DOTTED_LINE, CENTER_LINE și DASHED_LINE), precum și unul *nestandard*, definit de utilizator (valoarea USERBIT_LINE).

Stilul *nestandard* se precizează cu ajutorul parametrului *sablon*. Aceasta este o dată de tip *int* (16 biți). Fiecare bit din şablon reprezintă un pixel al liniei. Fiecare

bit din şablon setat reprezintă un pixel colorat cu culoarea curentă, biţii din şablon de valoarea zero reprezintă pixeli coloraţi cu culoarea de fond.

Alte funcţii din biblioteca standard a sistemului permit trasarea de figuri geometrice împreună cu colorarea interiorului acestora.

Indicăm mai jos prototipurile funcţiilor mai importante din această clasă:

- **void far bar(int st, int sus, int dr, int jos);**

unde:

st, sus, dr și jos - Au aceeaşi semnificaţii ca în cazul funcţiei *rectangle*.

Funcţia desenează un domeniu dreptunghiular colorat (fără a avea o frontieră scoasă în evidenţă).

void far bar3d(int st, int sus, int dr, int jos, int profunzime, int ind);

Funcţia desenează o prismă dreptunghiulară colorată pentru *ind* diferit de zero. Pentru *ind*=0, nu se trasează partea de sus a prismei.

void far pieslice(int xcentru, int ycentru, int unghistart, int unghifin, int raza);

Funcţia desenează un sector de cerc colorat.

void far fillpoly(int nr, int far *tabpct);

unde:

nr și tabpct - Au aceeaşi semnificaţii ca în cazul funcţiei *drawpoly*.

Funcţia desenează un poligon colorat.

void far fillellipse(int xcentru, int ycentru, int semiamaxare, int semiamaxica);

Funcţia desenează o elipsă colorată.

Mentionăm că figurile desenate cu funcţiile indicate mai sus se colorează folosind o culoare definită în prealabil cu ajutorul funcţiei *setfillstyle*. De asemenea, o figură poate fi colorată în mai multe feluri:

- folosind culoarea de fond;
- colorare uniformă (toţi pixelii din interiorul figurii au aceeaşi culoare) folosind culoarea definită prin funcţia *setfillstyle*;
- colorare prin haşură.

Haşura poate fi standard sau definită de utilizator.

Modul de colorare al unei figuri se defineşte tot cu ajutorul funcţiei *setfillstyle*. Ea are prototipul:

void far setfillstyle(int hasura, int culoarea);

unde:

hasura - Defineşte modul de colorare conform tabelului de mai jos.

culoarea - Defineşte culoarea pentru colorarea figurilor.

Parametrul *hasura* are următoarele valori:

Constantă simbolică	Valoare
EMPTY_FILL	0
SOLID_FILL	1
LINE_FILL	2
LTSLASH_FILL	3
SLASH_FILL	4
BKSLASH_FILL	5
LTBKSLASH_FILL	6
HATCH_FILL	7
XHATCH_FILL	8
INTERLEAVE_FILL	9
WIDE_DOT_FILL	10
CLOSE_DOT_FILL	11
USER_FILL	12

Valoarea EMPTY_FILL colorează figura folosind culoarea de fond.

Valoarea SOLID_FILL realizează colorarea uniformă a figurii (toţi pixelii din interiorul figurii au aceeaşi culoare).

Valorile de la LINE_FILL(2), pînă la CLOSE_DOT_FILL(11) definesc diferite haşuri standard.

Valoarea USER_FILL se utilizează cînd haşura se defineşte de către utilizator. Pentru a defini o haşură nestandard se utilizează funcţia *setfillpattern*. Aceasta are prototipul:

void far setfillpattern(char far *h_utilizator, int culoare);

unde:

h_utilizator - Este un pointer spre o zonă de memorie în care se defineşte, pe 8 octeţi, haşura.

culoare - Defineşte culoarea de haşurare.

Şablonul se defineşte printr-un şir de 8 octeţi. Fiecare octet defineşte 8 pixeli, deci un şablon defineşte 64 pixeli.

Modul curent utilizat la colorare se poate determina cu ajutorul funcţiei *getfillsettings*. Aceasta are prototipul:

void far getfillsettings(struct fillsettingstype far *stilcolorare);

unde:

fillsettingstype - Este definit în fişierul *graphics.h* astfel:

```
struct fillsettingstype {
    int pattern;
    int color;
};
```

unde:

pattern - Este componenta care defineşte modul de colorare utilizat la colorare (intreg din intervalul [0,12]).

color - Defineşte culoarea utilizată la colorare.

Funcția `getfillpattern` determină şablonul definit de utilizator cu ajutorul funcției `setfillpattern`. Ea are prototipul:

```
void far getfillpattern(char far *sablon);
```

La apelul funcției `getfillpattern` se atribuie, la 8 octeți din zona spre care pointează pointerul `sablon`, valorile care stabilesc hașura definită de utilizator prin apelul prealabil al funcției `setfillpattern`.

Utilizatorul poate trasa și alte figuri decit cele pentru care au fost prevăzute funcții standard de felul celor indicate mai sus. În acest scop se pot folosi funcțiile `putpixel`, `line`, `lineto`, `linerel`, `arc` etc., pentru a trasa elemente componente ale figurii de trasat.

Pentru a colora un domeniu inchis obținut în acest fel se folosește funcția `floodfill`. Aceasta are prototipul:

```
void far floodfill(int x, int y, int culoare_frontiera);
```

unde:

(x,y)

- Sunt coordonatele unui punct interior figurii care se colorează.

`culoare_frontiera`

- Definește culoarea utilizată la trasarea conturului figurii.

Interiorul figurii se colorează în conformitate cu parametrii curent setați prin apelul funcției `setfillstyle`.

La trasarea figurilor trebuie să se țină seama de faptul că pixelii nu sunt puncte ideale. Elă au forme dreptunghiulare. Din această cauză, figurile afișate pe ecran vor avea un aspect deformat față de forma lor ideală. Pentru a evita acest lucru se utilizează niște factori de corecție numiți coeficienți de *aspect*. Aceștia sint doi: unul relativ la abscisă și unul relativ la ordonata. Valorile lor se pot seta cu ajutorul funcției `setaspectratio` de prototip:

```
void far setaspectratio(int xaspect, int yaspect);
```

Valorile acestor coeficienți depind de adaptorul grafic.

Raportul $(double)xaspect/yaspect$ se folosește adesea la trasarea curbelor definite analitic.

Valorile curente ale acestor coeficienți pot fi găsite apelind funcția `getaspectratio` de prototip:

```
void far getaspectratio(int far *xaspect, int far *yaspect);
```

După apel, cei doi coeficienți de aspect curenti se află memorati în zonele spre care pointează parametrii `xaspect` și `yaspect`.

Exerciții:

19.10 Să se scrie un program care trasează linii orizontale folosind cele 4 stiluri standard și ambele grosimi.

PROGRAMUL BXIX10

```
#include <graphics.h>
#include <conio.h>

main() /* traseaza linii orizontale folosind cele 4 stiluri standard si ambele grosimi */
{
    char *stil[]={
        "SOLID_LINE = 0",
        "DOTTED_LINE =1",
        "CENTER_LINE =2",
        "DASHED_LINE =3"
    };
    char *grosime[]={
        "NORM_WIDTH =1",
        "THICK_WIDTH =3"
    };
    int gd=DETECT,gm;
    int i,j;
    int x,y;

    initgraph(&gd,&gm, "c:\\borlandc\\bgi");
    x=getmaxx()/4;
    y=getmaxy()/8;
    settextstyle(DEFAULT_FONT,HORIZ_DIR,1);
    settextjustify(LEFT_TEXT,CENTER_TEXT);
    for(i=SOLID_LINE;i<=DASHED_LINE;i++)
        for(j=0;j<2;j++){
            outtextxy(x,y,stil[i]);
            outtextxy(x+textwidth(stil[i])+4,y,grosime[j]);
            y+=textheight(grosime[j])+4;
            if(j) setlinestyle(i,0,3);
            else setlinestyle(i,0,1);
            line(x,y,3*x,y);
            y+=20;
        }
    getch();
    closegraph();
}
```

9.11 Să se scrie un program care trasează următoarele figuri:

- cerc;
- elipsă;
- dreptunghi;
- patrulater.

Tipul figurii se stabilește aleator. Figurile sint trasate folosind în mod aleator culoarea de desenare.

PROGRAMUL BXIX11

```
#include <graphics.h>
#include <conio.h>
#include <stdlib.h>
```

```

main() /* traseaza figuri geometrice in mod aleator */
{
    int gd=DETECT,gm;
    int i;
    int xmax,ymax,x,y;
    int poligon[10];
    initgraph(&gd,&gm,"c:\\borlandc\\bgi");
    outtextxy(0,0,"Pentru a termina actionati tasta zero");
    setviewport(0,8,getmaxx(),getmaxy(),1);

    srand(1993); /* seteaza saminta sirului de numere pseudo-aleatoare */
    xmax=getmaxx();
    ymax=getmaxy();
    for (;;){
        setcolor(random(15)+1);
        switch(random(4)){
            case 0: /* cerc */
                x=random(xmax/2)+xmax/4;
                y=random(ymax/2)+ymax/4;
                circle(x,y,random(80)+1);
                break;
            case 1: /* elipsa */
                x=random(xmax/2)+xmax/8;
                y=random(ymax/2)+ymax/8;
                ellipse(x,y,0,359,random(80)+1,
                        random(60)+1);
                break;
            case 2: /* dreptunghi */
                x=random(xmax/2)+xmax/10;
                y=random(ymax/2)+ymax/10;
                rectangle(x,y,x+random(xmax/4)+1,
                           y+random(ymax/4)+1);
                break;
            case 3: /* patrulater */
                for(i=0;i<8; i+=2){
                    poligon[i]=random(xmax);
                    poligon[i+1]=random(ymax);
                }
                poligon[8]=poligon[0];
                poligon[9]=poligon[1];
                drawpoly(5,polygon);
                break;
        } /* sfarsit switch */
        if(getch()=='0') break;
    }
    closegraph();
}

```

- 19.12 Să se scrie un program care colorează figurile geometrice trasate în programul precedent.

PROGRAMUL BXIX12

```

#include <graphics.h>
#include <conio.h>
#include <stdlib.h>

main() /* traseaza si coloreaza figuri geometrice in mod aleator */
{
    int gd=DETECT,gm;
    int i;
    int xmax,ymax,x,y;
    int poligon[8];

    initgraph(&gd,&gm,"c:\\borlandc\\bgi");
    outtextxy(0,0,"Pentru a termina actionati tasta zero");
    setviewport(0,8,getmaxx(),getmaxy(),1);

    srand(1993); /* seteaza saminta sirului de numere pseudo-aleatoare */
    xmax=getmaxx();
    ymax=getmaxy();
    setcolor(getmaxcolor());
    for (;;){
        setfillstyle(SOLID_FILL,random(15)+1);
        switch(random(4)){
            case 0: /* cerc */
                x=random(xmax/2)+xmax/4;
                y=random(ymax/2)+ymax/4;
                pieslice(x,y,0,360,random(80)+1);
                break;
            case 1: /* elipsa */
                x=random(xmax/2)+xmax/8;
                y=random(ymax/2)+ymax/8;
                fillellipse(x,y,random(80)+1,
                            random(60)+1);
                break;
            case 2: /* dreptunghi */
                x=random(xmax/2)+xmax/10;
                y=random(ymax/2)+ymax/10;
                bar(x,y,x+random(xmax/4)+1,
                     y+random(ymax/4)+1);
                break;
            case 3: /* patrulater */
                for(i=0;i<8; i+=2){
                    poligon[i]=random(xmax);
                    poligon[i+1]=random(ymax);
                }
                fillpoly(4,polygon);
                break;
        } /* sfarsit switch */
        if(getch()=='0') break;
    }
    closegraph();
}

```

- 19.13 Să se scrie un program care afișează 15 dreptunghiuri colorate, pe trei rinduri, folosind toate cele 15 culori ale paletei diferite de culoarea de fond.

Sub fiecare dreptunghi se listează indicele culorii dreptunghiului, în tabloul care definește paleta curentă de culori.

După afișarea celor 15 dreptunghiuri se poate regla monitorul aşa încit fiecare dreptunghi să fie vizibil.

Acest program poate fi utilizat ori de câte ori se constată că monitorul este dereglat.

PROGRAMUL BXIX13

```
#include <stdio.h>
#include <graphics.h>
#include <stdlib.h>
#include <conio.h>

main() /* afiseaza 15 dreptunghiuri colorate */
{
    int Adaptorgrafic,Modgrafic;
    int lat,inalt,x,y,i,j,culoare;
    char asculoare[5];

    Adaptorgrafic = DETECT;
    initgraph(&Adaptorgrafic,&Modgrafic,"c:\\borlandc\\bgi");
    culoare = 1;
    lat = 78;
    inalt = 64;
    x = lat/2;
    y = inalt/2;
    for(j = 0; j < 3; j++) {
        for( i = 0 ; i < 5 ; i++ ){
            setfillstyle(SOLID_FILL,culoare);
            setcolor(culoare);
            bar(x,y,x+lat,y+inalt);
            itoa(culoare,asculoare,10);
            outtextxy(x+lat/2,y+inalt+4,asculoare);
            ++culoare;
            x += (lat/2)*3;
        }
        y += (inalt/2)*3;
        x = lat/2;
    }
    getch();
    closegraph();
}
```

- 19.14 Să se scrie un program care afișează dreptunghiuri utilizând toate hașurile standard.

PROGRAMUL BXIX14

```
#include <stdio.h>
#include <graphics.h>
#include <stdlib.h>
#include <conio.h>

main() /* afiseaza dreptunghiuri cu toate hașurile standard */
{
    int Adaptorgrafic,Modgrafic;
    int lat,inalt,x,y,i,j,hasura;
    char ashas[5];

    Adaptorgrafic = DETECT;
    initgraph(&Adaptorgrafic,&Modgrafic,"c:\\borlandc\\bgi");
    lat = 78;
    inalt = 64;
    x = lat/2;
    y = inalt/2;
    for(j = 0,hasura=EMPTY_FILL; j < 3; j++) {
        for( i = 0 ; i < 5 ; i++ ){
            if(hasura >=12) break; /* nu mai sunt hașuri */
            setfillstyle(hasura,WHITE);
            bar(x,y,x+lat,y+inalt);
            rectangle(x,y,x+lat,y+inalt);
            itoa(hasura,ashas,10);
            outtextxy(x+lat/2,y+inalt+4,ashas);
            ++hasura;
            x += (lat/2)*3;
        }
        if(hasura >= 12) break;
        y += (inalt/2)*3;
        x = lat/2;
    }
    getch();
    closegraph();
}
```

- 19.15 Să se scrie un program care trasează un cerc folosind ecuațiile parametrice ale cercului:

$$(1) \begin{aligned} x &= r \cos(g) + x_{\text{centru}}; \\ y &= r \sin(g) + y_{\text{centru}}; \end{aligned}$$

unde:

$(x_{\text{centru}},y_{\text{centru}})$ - Sunt coordonatele centrului cercului.

r - Este raza cercului.

g - Este unghiul în radiani dintre axa Ox și raza OM, O fiind originea axelor de coordonate, iar M este punctul de pe cerc de coordonate (x,y) .

Cind unghiul g variază de la 0 la 2π , punctul $M(x,y)$ descrie cercul de rază r și cu centru în punctul de coordonate $(x_{\text{centru}},y_{\text{centru}})$.

Programul afișează două curbe pentru a pune în evidență importanța

coeficienților de aspect. Prima curbă se afișează folosind ecuațiile (1), iar cea de a doua curbă se trasează corectind ordonata conform relațiilor de mai jos:

$$(2) \begin{aligned} x &= r * \cos(g) + x_{centru}; \\ y &= r * (\text{xaspect}/\text{yaspect}) * \sin(g) + y_{centru}; \end{aligned}$$

unde:

x_{aspect} și y_{aspect} - Sunt coeficienții de aspect obținuți prin apelul funcției *getaspectratio*.

PROGRAMUL BXIX15

```
#include <stdio.h>
#include <graphics.h>
#include <stdlib.h>
#include <conio.h>
#include <math.h>

#define PI 3.14159265

main()
{
    int gd=DETECT,gm;
    int xaspect,yaspect;
    double FactorAspect,draza,rad;
    int x,y,g;
    int xcentru,ycentru;

    initgraph(&gd,&gm,"c:\\borlandc\\bgi");
    getaspectratio(&xaspect,&yaspect);
    FactorAspect=(double)xaspect/yaspect;
    draza=150;
    FactorAspect *=draza;
    xcentru=getmaxx()/2;
    ycentru=getmaxy()/2;

    /* traseaza un cerc fara a tine scama de coeficientii de aspect */
    for(g=0;g<360;g++){
        rad=g*PI/180.0;
        x=draza*cos(rad)+xcentru;
        y=draza*sin(rad)+ycentru;
        putpixel(x,y,WHITE);
    }

    getch();

    /* traseaza un cerc tinind scama de coeficientii de aspect */
    for(g=0;g<360;g++){
        rad=g*PI/180.0;
        x=draza*cos(rad)+xcentru;
        y=FactorAspect*sin(rad)+ycentru;
        putpixel(x,y,LIGHTBLUE);
    }

    getch();
}
```

```
closegraph();
}
```

19.16 Sa se scrie un program care deplasează o imagine pe ecran. Acest program a fost propus și realizat de Negrescu Dan și Filimon Ovidiu. Programul a apărut și în cărțile [13] și [21].

PROGRAMUL BXIX16

```
#include <graphics.h>
#include <conio.h>
#include <alloc.h>
#include <dos.h>

main() /* deplaseaza un vapor pe ecran */
{
    void desen_vapor();
    int grmode,grdrive;
    void *ptr;
    void *vapor1,*vapor2;
    int linia,coloana;

    grdrive=DETECT;
    initgraph(&grdrive,&grmode,"c:\\borlandc\\bgi");
    grmode=EGAHI;
    setgraphmode(grmode);
    setbkcolor(BLACK);
    cleardevice();
    desen_vapor();
    setfillstyle(SOLID_FILL,LIGHTRED);
    bar(10,0,25,10);
    vapor1 = malloc(imagesize(0,0,50,50));
    getimage(0,0,50,50,vapor1);
    cleardevice();
    desen_vapor();
    setfillstyle(SOLID_FILL,YELLOW);
    bar(25,0,40,10);
    vapor2 = malloc(imagesize(0,0,50,50));
    getimage(0,0,50,50,vapor2);
    setbkcolor(LIGHTBLUE);
    cleardevice();
    linia=coloana=0;
    for(;){
        putimage(coloana,linia,vapor1,XOR_PUT);
        delay(100);
        putimage(coloana,linia,vapor1,XOR_PUT);
        coloana+=25;
        if(coloana>570){
            putimage(coloana,linia,vapor2,XOR_PUT);
            delay(100);
            putimage(coloana,linia,vapor2,XOR_PUT);
            coloana+=25;
            if(coloana>570) coloana=0;
        }
    }
}
```

```

        else{
            linia+=60 ;
            coloana=0 ;
            if(linia>300) linia=0 ;
        }
        if(kbhit()) break ;
    }
    getch() ;
    closegraph() ;
    free(vapor1) ;
    free(vapor2) ;
}

void desen_vapor() /* deseneaza un vapor */
{
    int puncte[]={
        25,10,
        10,30,
        0,30,
        10,50,
        40,50,
        50,30,
        40,30,
        25,10
    } ;

    setcolor(CYAN) ;
    drawpoly(8,puncte) ;
    line(10,30,40,30) ;
    setfillstyle(SOLID_FILL,LIGHTMAGENTA) ;
    floodfill(25,45,CYAN) ;
    setfillstyle(SOLID_FILL,WHITE) ;
    floodfill(25,20,CYAN) ;
    setcolor(BROWN) ;
    line(25,10,25,30) ;
}

```

BIBLIOGRAFIE

1. C.Bohn, G.Jacopini
FlowDiagrams, Turning Machines and Languages with Only Two Formation Rules
 Comm ACM, 9, 5, pag. 366 - 371, 1966
2. Brian W. Kernighan, Dennis M. Ritchie
The C Programming Language
 Prentice-Hall, Inc, Englewood Cliffs
 New Jersey 1978
3. Donald E. Knuth
Tratat de programarea calculatoarelor, vol. 1 - 3
 Editura Tehnică
 Bucureşti 1974
4. Valentin Cristea și alții
Dictionar de informatică
 Editura științifică și enciclopedică
 Bucureşti 1981
5. Narian Gehani
C: An Advanced Introduction
 AT&T Bell Laboratories Murray Hill
 New Jersey 1985
6. Joel E. Richardson, Michel J. Carey, Daniel T. Schuh
The Design of the E Programming Language
 Computer Sciences Department
 University of Wisconsin Madison
 WI53706 1989
7. * * *
BORLAND INTERNATIONAL TURBO C++: Getting Started
 1990
8. * * *
BORLAND INTERNATIONAL TURBO C++: Programmer's Guide
 1990
9. Frederic Lung
Le Langage C++
 CNIT Paris - La Defense, 1990

10. Setrag Khoshafian, Razmik Abnous
Object Orientation Concepts, Languages, Databases, User Interfaces
John Wiley&Sons, Inc. New York
Chichester Brisbone Toronto
Singapore 1990
11. Aaron M. Tenenbaum, Yedidyah Langsam, Moshe J. Augenstein
Data Structures Using C
Prentice - Hall International, Inc.
1990
12. Clara Ionescu, Ioan Zsako
Structuri arborescente cu aplicațiile lor
Editura Tehnică
București 1990
13. Liviu Negrescu
Introducere în limbajul C vol. 1 - 2
Colecția GLOB
1990
14. Stroustrup Bjarne
The C++ Programming Language
Second Edition
Copyright 1991 by AT&T
Bell Telephone Laboratories, Incorporated
15. Liviu Negrescu
Limbajul C - Culegere de programe
Fasciculele 1 - 2
Cluj-Napoca 1991
16. Liviu Negrescu
Limbajul TURBO C
Editura LIBRIS
Cluj-Napoca 1992
17. Liviu Negrescu
Introducere în mediul de dezvoltare integrat TURBO C
Editura LIBRIS
Cluj-Napoca 1992
18. Ionuț Mușlea
Inițiere în C++ - Programare orientată pe obiecte
Editura MicroInformatica
Cluj-Napoca 1992
19. Donald L. Shell
CACM 2(iulie), pag. 30-32
1959
20. C. A. R. Hoare
Quicksort
Comp. j., 5, No. 1
1962
21. Liviu Negrescu
Introducere în limbajul C
Editura MicroInformatica
Cluj-Napoca 1993