

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

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



9 789736 500411

I.S.B.N. 973-650-042-X



9 789736 500428



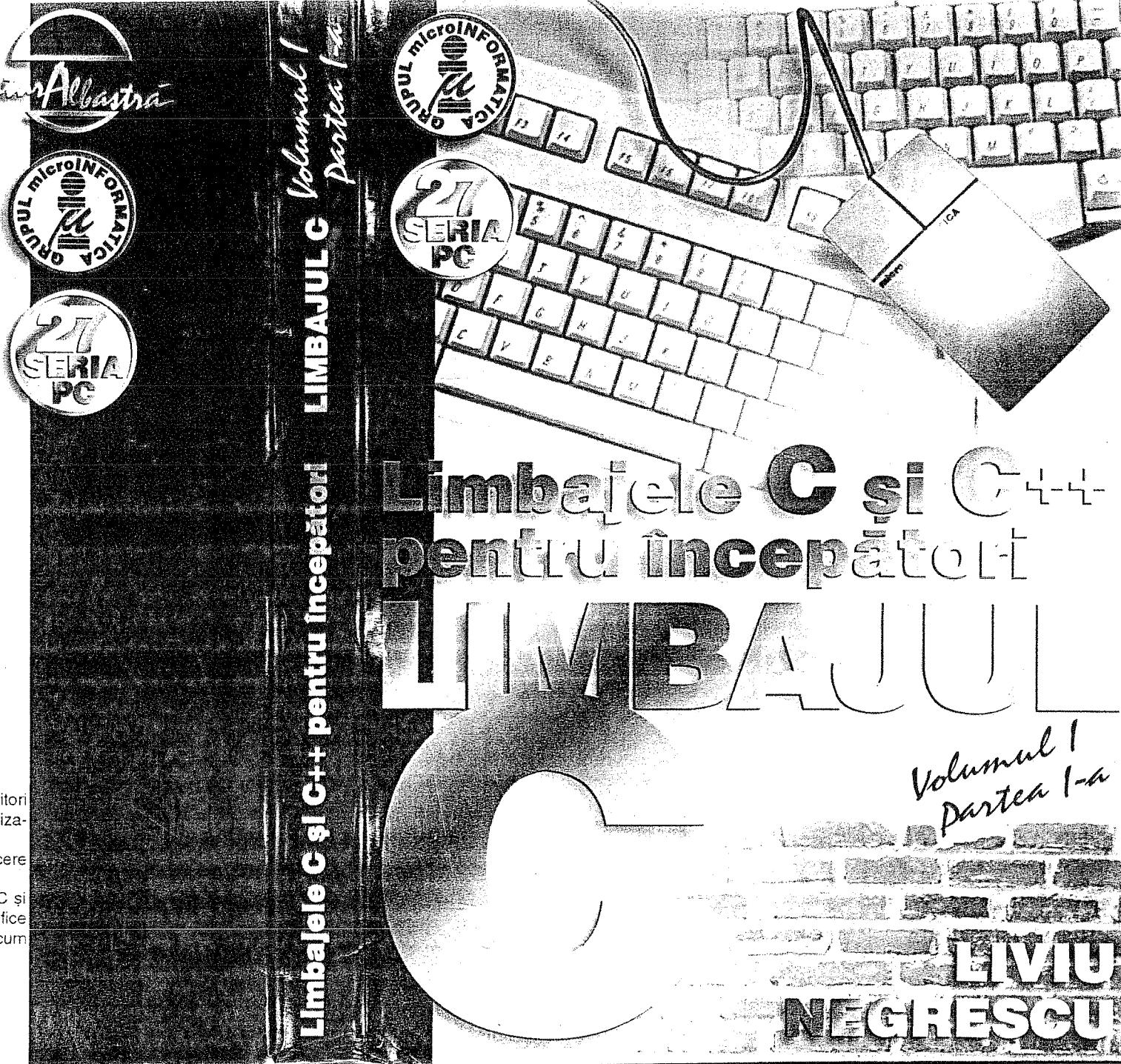
0 0 2 6 0

Limbajele C și C++ pentru începători

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

Prințele 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-ingineresci, în prelucrările de date precum și în scrierea programelor de sistem.





CLUJ-NAPOCA
2001

Păncosca

Florin

LIMBAJUL
Reeditare

C

**LIMBAJELE
C SI C++
PENTRU
ÎNCEPĂTORI
VOLUMUL I
Partea I-a**

Autor:

LIVIU NEGRESCU

A confruntat cu originalul
Irina Mitrov

Editura Albastră

Director editură
Smaranda Derveșteanu

Coordonator serie
Codruța Poenaru

Coperta
Liviu Derveșteanu

Tiraj: 1000 exemplare

Tipărit
EDITURA ALBASTRĂ
comanda 142 / 2001



CUPRINS

PREFĂTĂ	7
INTRODUCERE	11
1. NOTIUNI DE BAZĂ	23
1.1. Nume	23
1.2. Cuvinte cheie	24
1.3. Tipuri de date de bază	24
1.4. Structura unei funcții	26
1.5. Comentariu	29
1.6. Constante	30
1.6.1. Constante intregi	30
1.6.2. Constante flotante	32
1.6.3. Constantă caracter	33
1.6.4. Sir de caractere	36
1.7. Caractere sau spații albe (white spaces)	37
1.8. Variabile simple, tablouri și structuri	38
1.9. Declarația de variabilă simplă	40
1.10. Declarația de tablou	41
1.11. Apelul și prototipul funcțiilor	42
1.12. Preprocesare	45
1.12.1. Includeri de fișiere cu texte sursă	46
1.12.2. Substituiri de succesiuni de caractere la preprocesare	47
2. INTRĂRI/IEȘIRI STANDARD	50
2.1. Funcțiile getch și getche	50
2.2. Funcția putch	51
Exerciții	51
2.3. Macroulurile getchar și putchar	53
Exerciții	54
2.4. Funcțiile gets și puts	55
Exerciții	56
2.5. Funcția printf	57
2.5.1. Litera c	60
2.5.2. Litera s	60
2.5.3. Litera d	61
2.5.4. Litera o	61
2.5.5. Literele x și X	62
2.5.6. Litera u	62
2.5.7. Litera l	62

2.5.8.	Litera f	62	3.2.15.	Alți operatori ai limbajului C	112
2.5.9.	Literele e și E	63	3.2.16.	Tabela cu prioritățile operatorilor limbajului C	112
2.5.10.	Literele g și G	64	4.	INSTRUCȚIUNI	113
2.5.11.	Litera L	64	4.1.	Instrucțiunea vidă	114
	Exerciții	64	4.2.	Instrucțiunea expresie	114
2.6.	Funcția scanf	69	4.3.	Exerciții	115
2.6.1.	Litera c	71	4.4.	Instrucțiunea compusă	116
2.6.2.	Litera s	71	4.5.	Instrucțiunea if	117
2.6.3.	Litera d	73	Exerciții	118	
2.6.4.	Litera o	75	4.6.	Funcția standard exit	124
2.6.5.	Literele x sau X	75	Exerciții	125	
2.6.6.	Litera u	75	4.7.	Instrucțiunea while	126
2.6.7.	Litera f	75	Exerciții	127	
2.6.8.	Litera l	75	4.8.	Instrucțiunea for	144
2.6.9.	Litera L	75	Exerciții	147	
	Exerciții	76	4.9.	Instrucțiunea do-while	150
3.	EXPRESII	79	Exerciții	151	
3.1.	Operand	79	4.10.	Instrucțiunea continue	155
3.2.	Operatori	80	4.11.	Funcțiile standard sscanf și sprintf	156
3.2.1.	Operatorii aritmetici	80	Exerciții	157	
	Exerciții	82	4.12.	Instrucțiunea break	159
3.2.2.	Regula conversiilor implicate	86	Exerciții	159	
	Exerciții	87	4.13.	Instrucțiunea switch	162
3.2.3.	Operatorii de relație	89	Exerciții	164	
	Exerciții	90	4.14.	Instrucțiunea goto	166
3.2.4.	Operatorii de egalitate	90	Exerciții	168	
3.2.5.	Operatorii logici	91	4.15.	Programarea procedurală, funcții, apelul și revenirea din ele	172
	Exerciții	93	Exerciții	179	
3.2.6.	Operatorii logici pe biți	95	Apel prin valoare și apel prin referință	179	
3.2.7.	Operatorii de atribuire	100	Exerciții	181	
	Exerciții	101	5.	CLASE DE MEMORIE	197
3.2.8.	Operatorii de incrementare și decrementare	103	5.1.	Variabile globale	197
3.2.9.	Operatorul de forțare a tipului sau de conversie explicită (expresie cast)	104	5.2.	Variabile locale	199
	Exerciții	104	5.3.	Alocarea parametrilor	202
3.2.10.	Operatorul dimensiune	105	5.4.	Utilizarea parametrilor și a variabilelor globale	202
3.2.11.	Operatorul adresă	107	5.5.	Variabile registru	203
3.2.12.	Operatorii paranteză	107	Exerciții	204	
3.2.13.	Operatorii condiționali	108	6.	INITIALIZARE	211
	Exerciții	109	6.1.	Inițializarea variabilelor simple	211
3.2.14.	Operatorul virgulă	110		Exerciții	212
	Exerciții	111	6.2.	Inițializarea tablourilor	213

Exerciții	217
---------------------	-----

7. PROGRAMAREA MODULARĂ 225

Exerciții	226
---------------------	-----

8. POINTERI 235

Declarația de pointer și tipul pointer	237
--	-----

8.2. Realizarea apelului prin referință utilizând parametri de tip pointer 241

Exerciții	242
---------------------	-----

8.3. Legătura dintre pointeri și tablouri 250

Operări cu pointeri	251
-------------------------------	-----

8.4.1. Operări de incrementare și decrementare a pointerilor 252

8.4.2. Adunarea și scăderea unui întreg dintr-un pointer 253

Compararea a doi pointeri	254
-------------------------------------	-----

8.4.4. Diferența a doi pointeri 255

Exerciții	255
---------------------	-----

8.5. Modificatorul const 258

Funcții standard utilizate la prelucrarea sirurilor de caractere	262
--	-----

8.6.1. Lungimea unui sir de caractere 263

Copierea unui sir de caractere	264
--	-----

8.6.3. Concatenarea sirurilor de caractere 265

Compararea sirurilor de caractere	266
---	-----

Exerciții	268
---------------------	-----

8.7. Expresie lvalue 273

Alocarea dinamică a memoriei	275
--	-----

Exerciții	278
---------------------	-----

8.9. Utilizarea tablourilor de pointeri la prelucrări de date de tip sir de caractere 282

Exerciții	283
---------------------	-----

8.10. Tratarea parametrilor din linia de comandă 286

Exerciții	287
---------------------	-----

8.11. Pointeri spre funcții 289

Exerciții	291
---------------------	-----

8.12. Tablouri de pointeri 294

Exerciții	298
---------------------	-----

9. RECURSIVITATE 301

Exerciții	304
---------------------	-----

PREFĂTĂ

Anii '90 sunt consacrați *programării orientate spre obiecte* (în engleză Object Oriented Programming sau prescurtat OOP) la fel cum anii '70 au fost numiți anii *programării structurate*.

Programarea orientată spre obiecte este un *stil* de programare care permite abordarea eficientă a aplicațiilor complexe. Aceasta se asigură pe baza:

- elaborării de componente reutilizabile, extensibile și ușor modificabile, fără a reprograma totul de la început;
- construirii de biblioteci de module extensibile;
- implementării simple a interacțiunilor programelor cu mediul de calcul prin utilizarea unor elemente standardizate.

Pe scurt, putem afirma că, programarea orientată spre obiecte permite să se implementeze sisteme complexe cu ajutorul unor componente individuale, standardizate, reutilizabile, extensibile și ușor modificabile.

Această concepție a influențat dezvoltarea limbajelor de programare. Ea s-a manifestat începând din anii '60 cind s-a introdus conceptul de *obiect* în limbajul *Simula*.

În principiu, un obiect are o structură de date bine precizată și totodată sunt definite operațiile care pot utiliza componentele lui.

Simula se consideră că se află la baza limbajelor care permit programarea orientată spre obiecte.

În cele ce urmează vom spune despre un limbaj de programare că este *orientat spre obiecte* dacă permite programarea orientată spre obiecte.

În anii '70, programarea orientată spre obiecte s-a utilizat mai ales prin intermediul limbajului *Smalltalk*. Așa cum s-a amintit mai sus, în acel deceniu s-au pus bazele programării structurate care a avut un succes deosebit prin utilizarea limbajului *Pascal*.

În anul 1972 a apărut *limbajul C*. Aceasta este un limbaj de programare care are o destinație universală. Autorii acestui limbaj sunt Dennis M. Ritchie și Brian W. Kernighan de la Bell Laboratories.

Limbajul C a fost proiectat în ideea de a asigura implementarea portabilă a sistemului de operare *UNIX*. Un rezultat direct al acestui fapt este acela că programele scrise în limbajul C au o *portabilitate* foarte bună.

În mod intuitiv, spunem că un program este portabil dacă el poate fi transferat ușor de la un tip de calculator la altul. În principiu, toate limbajele de nivel înalt asigură o anumită portabilitate a programelor. La ora actuală se afirmă că programele scrise în C sunt cele mai portabile.

Prin anii '80 interesul pentru programarea orientată spre obiecte a crescut, ceea ce a condus la apariția de limbaje care să permită utilizarea ei în scrierea programelor. Limbajul C a fost și el dezvoltat în această direcție. Astfel, în anul

1980 s-a dat publicația *limbajul C++*.

Limbajul C++ a fost elaborat de Bjarne Stroustrup de la AT&T. El este un superset al limbajului C și permite utilizarea principalelor concepte ale programării orientate spre obiecte.

Limbajul Pascal a fost și el dezvoltat în aceasta direcție.

La ora actuală, majoritatea limbajelor de programare moderne au fost dezvoltate în direcția programării orientate spre obiecte.

În prezent, anii '80 sunt considerați ca fiind decada care a lansat programarea orientată spre obiecte.

Limbajul C++, ca și limbajul C, se bucură de o portabilitate mare și este implementat pe o gamă largă de calculatoare începând cu microcalculatoare și pînă la cele mai mari supercalculatoare. Între diferitele implementări ale limbajului C++ există diferențe legate de specificul calculatoarelor, dar aceste diferențe nu sunt esențiale.

Limbajul C++ a fost implementat pe microcalculatoarele compatibile IBM PC în mai multe variante. Cele mai importante implementări ale limbajului C++ pe aceste calculatoare sunt cele realizate de firmele Microsoft și Borland. Aceste firme au implementat limbajul C++ pe microcalculatoarele respective în mai multe versiuni.

În cartea de față se prezintă facilitățile de bază ale limbajului C++ care sunt comune diferitelor implementări ale sale. Exercițiile prezentate în cartea de față au fost rulate folosind compilatorul TURBO C++ v2.0 (acest compilator este marcat înregistrată a firmei Borland).

Concepțele programării orientate spre obiecte au influențat în mare măsură dezvoltarea limbajelor de programare din ultimii ani. De obicei, multe limbaje au fost extinse astfel încît ele să admită concepțele mai importante ale programării orientate spre obiecte. Uneori s-au făcut chiar mai multe extensii ale același limbaj.

De exemplu, în prezent există extensiile limbajului C++. O astfel de extensie o constituie *limbajul E* ai căruia autori sunt Joel E. Richardson, Michael J. Carey și Daniel T. Schuh de la universitatea Wisconsin Madison.

Acest limbaj a fost proiectat pentru a permite exprimarea simplă a tipurilor și operațiilor interne sistemelor de gestiune a bazelor de date.

Printre altele, limbajul E permite crearea și gestiunea *obiectelor persistente*, lucru deosebit de important pentru sistemele de gestiune a bazelor de date.

O altă extensie a limbajului C++ este *limbajul O* dezvoltat la Bell Laboratories. Cele două limbaje (E și O) sunt în esență echivalente.

Despre limbajul E se afirmă că este în principal orientat spre scrierea programelor sistem, în particular pentru implementarea sistemelor de gestiune a bazelor de date.

Despre limbajul O se spune că încearcă să imbine facilitățile de nivel înalt cu cele ale programării sistem.

O altă extensie a lui C++ este limbajul Avalon/C++. Ea a fost realizată de

Detlefs D., Herlihy M. și Wing J. Acest limbaj este destinat pentru a suporta calcul distribuit.

Printre altele, aceste ultime două extensii admit și ele obiecte persistente.

În prezent, concepțele programării orientate spre obiecte au aplicații importante nu numai în ce privește dezvoltarea limbajelor de programare, ci și la implementarea altor sisteme complexe.

Cele mai mari realizări obținute pînă în momentul de față sunt legate de implementarea *sistemelor de gestiune a bazelor de date* și a *interfețelor utilizator*.

Limbajul E a fost utilizat, de către autorii lui, la implementarea proiectului EXODUS care este un sistem de gestiune a unei baze de date "extensibile".

În prezent există numeroase proiecte de acest fel implementate cu ajutorul concepției programării orientate spre obiecte. Bazele de date respective se numesc *baze de date orientate spre obiecte*.

Interfețele utilizator au atins o dezvoltare mare datorită facilităților oferite de componente hardware ale diferitelor tipuri de calculatoare. În principiu ele simplifică interacțiunea dintre programe și utilizatorii acestora. Astfel, diferite comenzi, date de intrare sau rezultate pot fi exprimate simplu și natural utilizând diferite standarde care conțin ferestre, bare de meniuri, cutii de dialoguri, butoane etc. Un dispozitiv primar de selectare utilizat foarte frecvent și cu mult succes este așa numitul "șoarece" (mouse). Toate acestea conduc la interfețe simple și vizuale, accesibile pentru utilizatorii finali.

Implementarea interfețelor este mult simplificată prin utilizarea limbajelor orientate spre obiecte. Aceasta mai ales datorită posibilităților de a utiliza componente standardizate aflate în biblioteci specifice.

Importanța aplicării conceptului de reutilizare prin intermediul limbajelor orientate spre obiecte rezultă din faptul că interfețele utilizator adesea ocupă 40% din codul total al aplicației.

Limbajele utilizate frecvent la implementarea interfețelor utilizator sunt limbajele C++ și Pascal.

Firma Borland comercializează o bibliotecă de componente standardizate care pot fi utilizate la implementarea interfețelor utilizator folosind unul din limbajele C++ și Pascal.

Produsul respectiv se numește *Turbo Vision*. Utilizarea lui pentru implementarea de interfețe utilizator presupune cunoașterea a cel puțin unuia din aceste limbaje de programare.

Interfețele utilizator implementate cu ajutorul limbajelor orientate spre obiecte le vom numi *interfețe utilizator orientate spre obiecte*.

De obicei, interfețele utilizator gestionează ecranul în mod grafic. O astfel de interfață utilizator se numește *interfață utilizator grafică*.

Una dintre cele mai populare interfețe utilizator grafice pentru calculatoarele compatibile IBM PC este produsul *Windows* oferit de firma Microsoft. Produsul a fost lansat în noiembrie 1985 și pînă acum a fost vîndut în cîteva milioane de

exemplare. În prezent este lansat în exploatare Windows 2000.

Windows este un sistem de operare care amplifică facilitățile sistemului de operare MS-DOS. Aplicațiile Windows se pot dezvolta folosind diferite medii de dezvoltare ca: Turbo C++ pentru Windows, Pascal pentru Windows, Microsoft C++ 7.0, Microsoft Visual Basic, Visual C și Visual C++.

Componentele Visuale permit specificarea în mod grafic a interfeței utilizator, a unei aplicații, folosind șoarecele, iar aplicația propriu-zisă se programează într-un limbaj de tip Basic, C sau C++.

Din cele de mai sus, putem afirma că limbajul C++ se bucură de o popularitate crescută fiind utilizat în măsură mare la implementarea de aplicații complexe.

Dacă prin anii '70 se consideră că o singură persoană este rezonabil să se ocupe cu programe de pînă la 4-5 mii de instrucțiuni, în prezent, în condițiile folosirii limbajelor de programare orientate spre obiecte, această medie este de 25 mii de instrucțiuni.

Carta de față se adresează tuturor celor care doresc să se familiarizeze cu limbajul C++. Din motive de spațiu, în carte nu sunt date detaliile cu privire la noțiunile de bază ale programării, arhitectura sau modul de funcționare al unui calculator, considerindu-se că cititorul este familiarizat cu cel puțin un limbaj de programare. Tot din lipsă de spațiu nu este descris mediul de dezvoltare integrat TURBO C++ el fiind foarte asemănător cu mediile integrate de dezvoltare ale celorlalte compilatoare furnizate de firma Borland. Cititorul care nu este familiarizat cu astfel de medii poate consulta alte cărți care descriu astfel de medii (vezi de exemplu [17]).

În încheiere exprim multumiri D-nei Irina Mitrov pentru observațiile și sugestiile care mi le-a făcut în legătură cu continutul cărții și al exercițiilor, precum și cu privire la îmbunătățirea clarității și exprimării diferitelor noțiuni. Menționez activitatea intensă depusă de D-na Irina Mitrov și fiica mea, Lavinia Negrescu, în legătură cu munca de introducere pe calculator a textului cărții. Tot pe această cale doresc să-i mulțumesc fiului meu, Dan Negrescu, pentru ajutorul pe care mi l-a dat în legătură cu formularea și rezolvarea unor exerciții care sunt legate de facilitățile sistemului de operare MS-DOS.

De asemenea, mulțumesc editurii Casa de Editură Albastră pentru efortul depus în vederea publicării ingrijite a cărții de față.

Autorul

INTRODUCERE

Limbajul C++ este un superset al limbajului C având o utilizare universală și care permite scrierea de programe orientate spre obiecte.

Faptul că limbajul C++ este un superset al limbajului C înseamnă că orice program scris în limbajul C este în același timp și un program scris în limbajul C++. Această afirmație este adevărată cu mici excepții, de exemplu, compilatorul C++ face unele controale suplimentare față de compilatorul C. Aceste controale se referă în primul rînd la tipurile parametrilor funcțiilor, precum și la tipurile valorilor returnate de ele.

Faptul că limbajul C++ permite scrierea de programe orientate spre obiecte înseamnă că el permite utilizarea unor concepții de bază ale acestui stil de programare. Acest fapt reprezintă un salt calitativ față de limbajul C, ceea ce se exprimă sugestiv și prin numele limbajului (C++). El nu a fost numit limbajul D deoarece este o extensie a lui C.

Tinând seama că operatorul "++", din limbajul C, reprezintă operația de incrementare, numele C++ se poate citi "C incrementat".

În principiu, putem afirma că limbajul C++ este un "C mai bun", care oferă atât facilitățile limbajului C cât și programarea orientată spre obiecte.

Facilitățile principale ale limbajului C sunt:

- portabilitatea mare a programelor;
- flexibilitatea în programare;
- programe compacte;
- lucru pe biți;
- calcul de adrese.

Menționăm că limbajul C inițial a fost proiectat în ideea de a asigura o portabilitate bună a programelor.

Ca o consecință a acestui deziderat amintim lipsa, din limbajul C, a instrucțiunilor de intrare/ieșire. Aceste operații se realizează prin funcții de bibliotecă. Eliminarea din limbaj a instrucțiunilor de intrare/ieșire rezulta din faptul că operațiile de intrare/ieșire sunt dependente de particularitățile hardware ale calculatorelor.

Flexibilitatea în programare rezultă, printre altele, din numărul mai redus de *controale* pe care îl face compilatorul C, față de alte compilatoare, de exemplu față de compilatorul limbajului Pascal. Acest fapt a provocat o serie de disperții, întrucît absența unor controale severe la compilare se consideră că îngreunează depistarea unor erori în programare. Cu toate acestea, flexibilitatea în programare obținută în acest fel s-a dovedit a fi adesea utilă și trebuie considerată o facilitate în plus a limbajului decit o parte negativă a lui.

Alte facilități care măresc flexibilitatea în programare sunt legate de prezența în limbaj a unor instrucții care asigură revenirea simplă dintr-o funcție, ieșirea

din cicluri etc.

Compilatorul C++ realizează unele controale suplimentare față de compilatorul C. Cu toate acestea programarea în limbajul C++ se bucură de o flexibilitate mare.

O altă facilitate importantă a limbajului C este posibilitatea de a compacta programele sursă. Limbajul C permite o compactare bună a programelor sursă pe seama unor construcții specifice, cum sunt: expresia de atribuire, expresia condițională, operatorii de incrementare și decrementare etc.

Limbajul C oferă facilități specifice limbajelor de asamblare, cum sunt lucrul pe biți și calculul cu adrese. Acest fapt permite adesea scrierea de programe optime, atât în ceea ce privește memoria, cât și timpul de execuție.

Limbajul C se consideră că este un intermediar între limbajele de nivel înalt și cele de asamblare. El a cîștigat o popularitate mare în decursul anilor datorită facilităților amintite mai sus.

La ora actuală există un standard ANSI al limbajului C, dar numeroasele sale implementări se abată de la acest standard.

În carteaua de față ne vom referi la versiunea TURBO C v2.0 care este marca înregistrată a firmei BORLAND și este implementată pe calculatoarele compatibile IBM PC.

Compilatorul TURBO C poate fi apelat prin intermediul *mediului integrat de dezvoltare* (Integrated Development Environment) sau printr-o linie de comandă.

Fără a intra în detaliu cu privire la mediul integrat de dezvoltare TURBO C, indicăm mai jos o secvență simplă de comenzi pentru a lansa în execuție un program C nou:

- după setarea pe directorul corespunzător se tastează:
 - tc;
 - <Alt>-F;
 - N.
- se editează programul sursă folosind editorul mediului TURBO C;
- se activează tasta F2;
- se indică numele fișierului pentru salvarea programului sursă pe disc;
- se realizează compilarea, link-editarea și lansarea în execuție a programului tastând:
<Ctrl>-F9
- pentru a vizualiza rezultatele execuției programului se tastează:
<Alt>-F5
- se revine în fereastra de editare a mediului acționând o tastă oarecare.

Tasta F1 poate fi acționată pentru a obține informații suplimentare. Se revine în fază precedentă acționând tasta ESC.

Pentru a ieși din mediul integrat de dezvoltare TURBO C se poate tastea:

<Alt>-X

Compilatorul TURBO C++ poate fi și el apelat prin intermediul mediului integrat de dezvoltare TURBO C++ sau printr-o linie de comandă.

Mediul integrat de dezvoltare TURBO C++ este asemănător cu cel al compilatorului TURBO C. De exemplu, secvența de mai sus poate fi utilizată și în cazul compilatorului TURBO C++ v2.0 cu singura diferență că în loc de *tc* se va tastea *bc*. Menționăm că fișierele cu programe sursă C au extensia .C, iar cele cu programele sursă C++ au extensia .CPP. Programele sursă C din această carte au fost compilate cu compilatorul C++ și ele au extensia .CPP.

Limbajul C, alături de limbajul Pascal și celelalte limbiage precedente lor sunt adecvate *programării procedurale*.

Programarea procedurală este un model de programare utilizat frecvent încă din fază inițială de existență a limbajelor de programare.

La baza acestei programări se află *procedura*. Aceasta poate fi considerată ca un proces de *abstractizare*. Acest proces se realizează pe baza *parametrilor* care permit realizarea procedurii făcând *abstracție de valorile concrete* pentru care să se execute procedura.

De exemplu, la realizarea unei proceduri pentru calculul funcției *sin* se face abstracție de valoarea concretă a unghiului. Procedura are un parametru care ne permite să neglijăm valoarea concretă a unghiului. Procedura se definește în așa fel încât ea să calculeze valoarea funcției *sin* în funcție de valoarea parametrului ei, valoare care se concretizează la execuție în momentul apelului procedurii respective.

Acest proces de abstractizare prin intermediul parametrilor procedurilor se numește *abstractizare procedurală*.

Un alt aspect al acestui proces de abstractizare este acela că procedura poate fi privită ca o "cutie neagră". La apelul ei nu se au în vedere detaliile de realizare ale procedurii. De exemplu, la apelul procedurii de calcul a valorii funcției *sin* sunt neglijate detaliile cu privire la algoritmul de calcul al funcției respective. La această fază interesează "tipul" parametrului și eventual precizia de calcul a funcției respective.

Abstractizarea procedurală a fost folosită pe scară largă la rezolvarea multor probleme aplicative sau de sistem. Importanța ei crește odată cu creșterea complexității problemelor de rezolvat. De obicei, problemele complexe se descompun în subprobleme mai simple. Acestea, la rîndul lor, pot fi și ele descompuse în continuare și aşa mai departe, pînă cînd se ajunge la componente suficient de simple. Acest proces de descompunere se realizează printr-un proces de abstractizare care permite neglijarea diferitelor detaliu. În felul acesta pot fi pusă în evidență procese de calcul care prin utilizarea de parametri pot fi exprimate printr-o procedură.

Pasul următor programării procedurale a fost *programarea modulară*. Deși nu există o definiție riguroasă pentru noțiunea de modul, citindu-l pe Bjarne

Stroustrup, se poate afirma că *modulul* este un set de proceduri înrudite împreună cu datele pe care le manevreză (vezi [14]).

Ceea ce trebuie să mai precizăm este faptul că datele manevrate de aceste proceduri, care compun un modul, sunt de obicei "ascunse" în modulul respectiv. Aceasta înseamnă că în afara modulului nu se are acces direct la datele ascunse de modul. Se are acces la ele numai indirect, prin intermediul procedurilor modulului respectiv. Limitarea accesului la date prin ascunderea lor într-un modul conduce la o protejare a lor, ceea ce prezintă importanță mai ales în cazul programelor complexe.

Un exemplu simplu de modul este modulul pentru implementarea unei *stive*. În forma cea mai simplă, stiva poate fi considerată ca fiind o zonă de memorie în care pot fi păstrate date, de obicei de un același tip. Datele pot fi scoase din stivă numai în ordinea inversă păstrării lor. Ultima dată pusă în stivă se spune că este în virful ei. Data scoasă din stivă este totdeauna aceea aflată în virful stivei. Totodată, după scoaterea unei date din stivă, în virful stivei se va afla data care a fost pusă pe stivă imediat înaintea celei scoase din stivă. În felul acesta, totdeauna se scoate din stivă ultima dată păstrată (pusă) în stivă. Se obișnuiește să se spună că stiva este o zonă de memorie gestionată după principiul: ultimul pus pe stivă este primul scos din stivă (last in-first out sau pe scurt LIFO). Este important ca acest principiu să fie respectat. De exemplu, nu este admis să se scoată din stivă sau să se aibă acces la un element care nu este în virful stivei. De asemenea, cind se pune un element pe stivă, acesta poate fi pus numai după cel aflat în virful stivei și prin aceasta el devine elementul *din* virful stivei. Din această cauză, zona de memorie prin care se implementează stiva trebuie "ascunsă", accesul la ea realizându-se numai prin intermediul a două proceduri:

- una care să pună o dată pe stivă (procedură care de obicei se numește *push*); și
- una care să scoată o dată din stivă (procedură care de obicei se numește *pop*).

De obicei, la aceste proceduri se mai adaugă o procedură pentru inițializarea stivei. Această procedură are drept scop vidarea stivei. În urma apelului ei, stiva devine vidă (această procedură de obicei se numește *clear*).

Cele trei proceduri amintite mai sus se consideră "înrudite" și împreună cu zona de memorie utilizată pentru implementarea stivei se grupează într-un modul. Utilizatorul nu poate gestiona direct această zonă de memorie decit numai prin intermediul celor 3 proceduri amintite mai sus.

Un modul similar se poate defini pentru a gestiona o zonă de memorie conform principiului: primul pus în zona respectivă este și primul scos (first in-first out sau pe scurt FIFO). O astfel de zonă de memorie se spune că formează o *coadă*.

Adesea, ascunderea datelor în modul se extinde și asupra procedurilor, limitându-se accesul din afara modulului la unele proceduri din compunerea lui.

Programarea modulară corespunde mai bine ideii amintite mai sus cu privire la descompunerea problemelor complexe în subprobleme mai simple. Ea permite un grad mai înalt de abstractizare pe seama ascunderii datelor și a procedurilor.

Limbajul C a fost proiectat în aşa fel încât să permită realizarea de module cu ascunderea datelor și a procedurilor. Astfel, datele și procedurile declarate cu *static* au o valabilitate limitată în cadrul fișierului sursă în care sunt declarate.

Programarea modulară are anumite limite. Așa de exemplu, dacă este nevoie de mai multe stive, atunci programarea modulară este o soluție destul de complexă pentru a gestiona stivele respective.

O rezolvare mai simplă și naturală s-ar obține dacă am putea defini date de tip stivă, așa cum putem defini, de exemplu, date de tip întreg, flotant sau caracter.

Toate limbajele de nivel înalt pun la dispoziția utilizatorului un anumit număr de tipuri predefinite. De exemplu, limbajul FORTRAN are tipurile predefinite *integer* și *real*. Limbajul C are și el aceste tipuri predefinite (întreg și flotant) și în plus mai pune la dispoziția utilizatorului tipul caracter.

Un *tip de date* descrie un set de date pe care le mai numim și obiecte, care au aceeași reprezentare. De asemenea, există un număr de operații asociat cu un tip de date. De exemplu, cu tipul întreg se asociază cele patru operații aritmétice și eventual și altele.

Ulterior s-a constatat că este util ca limbajul să permită utilizatorului să definească el însuși tipuri diferite de cele predefinite în limbaj. În acest scop, în limbajul Pascal s-a introdus noțiunea de *înregistrare* (record) care permite definiri de grupe de date care nu neapărat au un același tip. O astfel de grupă de date reprezintă un tip nou de date. O facilitate analogă există și în limbajul C. În acest caz se utilizează noțiunea de *structură* (struct). Atât construcția record din Pascal cit și struct din limbajul C definesc *reprezentarea* tipului nou de date.

Utilizatorul definește, de asemenea, operații cu date de acest tip prin intermediul unor proceduri sau funcții.

Tipurile definite în acest fel se numesc *tipuri definite de utilizator* sau mai scurt *tipuri utilizator*. Un exemplu de tip utilizator folosit frecvent poate fi tipul de număr complex. Acesta are o reprezentare care se definește simplu în limbajul C folosind construcția struct. Numărul complex este o grupă formată din două numere flotante, primul reprezentând partea reală a numărului complex, iar celălalt partea imaginară a lui.

În continuare, se pot defini variabile care să aibă tipul complex, exact la fel cum se definesc variabile de tipul întreg, flotant sau caracter.

Pentru a realiza operații cu numere complexe, utilizatorul definește funcții corespunzătoare. Operațiile care se execută asupra datelor care au tipuri predefinite au diferență facilități care lipsesc în cazul tipurilor utilizator. Așa de exemplu, pentru datele de tip întreg sau flotant se utilizează operatorii obișnuiți pentru operațiile aritmétice (+, -, * și /). Acest lucru nu este posibil și pentru numerele de tip complex. De asemenea, o serie de controale și conversii se

realizează automat în cazul utilizării datelor de tipuri predefinite.

Un alt neajuns al tipurilor utilizator, definite ca mai sus, este faptul că nu se asigură nici o protecție asupra componentelor unei date.

Aceste neajunsuri rezultă mai ales din faptul că la definirea tipurilor utilizator de fapt se definesc numai reprezentările datelor respective. Nu se precizează nici acțiunile posibile asupra datelor respective. Chiar dacă astfel de acțiuni sunt definite prin proceduri sau funcții, compilatoarele nu au informații în acest sens. De aceea, pasul următor constă în posibilitatea de a preciza în limbaj procedurile sau funcțiile care au acces la componentele unei date de tip utilizator. O astfel de extensie s-a realizat foarte simplu, de exemplu în cazul construcției struct din limbajul C, alături de definirea reprezentării componentelor se indică și lista funcțiilor care definesc acțiunile asupra lor. Aceasta însă, nu înseamnă că numai funcțiile respective au acces la datele componente ale structurii. Cu alte cuvinte, prin enumerarea funcțiilor în cadrul construcției struct nu se oferă nici o protecție a componentelor structurii respective, ci numai se indică faptul că datele componente ale structurii sunt prelucrate prin funcțiile enumerate, dar la ele putem avea acces și prin alte funcții.

Mai târziu s-a constatat că adesea anumite componente date și eventual și componente funcții atașate unui tip definit de utilizator este bine să fie protejate pentru a înălța accesul neautorizat. În felul acesta s-a ajuns la noțiunea de *tip abstract de date*. Aceasta noțiune depășește posibilitățile limbajului C. Tipurile abstrakte de date pot fi definite în limbajul C++ folosind noțiunea de *clasa*. O clasă, deci, definește atât reprezentarea datelor tipului respectiv cât și funcțiile care au acces și pot prelucra datele respective.

La definirea unei clase se indică componentele date și funcție la care au acces utilizatorii clasei (componente *publice*). Componentele care nu sunt publice permit un acces limitat și în felul acesta ele sunt protejate față de accesurile neautorizate. Se obișnuiește să se spună că, clasele "incapsulează" datele. Acest proces de incapsulare este deosebit de important permitând definirea de componente *protejate și reutilizabile*. Deosebirea dintre clasă și construcția struct constă chiar în ideea de protecție. Un tip utilizator definit printr-o construcție struct poate fi definit printr-o clasa care are toate componente date și funcție, publice.

O dată de un tip abstract (adică tip definit printr-o clasă) se spune că este un *obiect*. De asemenea, se obișnuiește să se spună că obiectul este o *instanțiere* a clasei care definește tipul său.

Reluind exemplul cu privire la definirea și gestiunea stivelor, observăm că de data aceasta putem introduce tipul abstract de date *stivă* prin intermediul unei clase. Zona de memorie care se organizează ca stivă este o componentă dată protejată a clasei respective. O altă componentă dată protejată este cea care definește virful stivei. Componentele funcție, publice, ale clasei sunt funcțiile *push*, *pop* și *clear*. Ele pot fi apelate din diferite funcții ale programului. Deoarece componente date sunt protejate, la ele nu se are acces decit prin

intermediul celor trei funcții publice indicate mai sus. Prin instanțierea acestei clase se obține un obiect de tip *stivă*. În felul acesta se pot obține atâtea obiecte de tip *stivă*, cite sunt necesare, exact la fel cum se pot defini atâtea date de un tip predefinit, cite sunt necesare.

În mod analog, se poate defini, printr-o clasă, tipul abstract de date pentru numere complexe. În acest caz sunt necesare două componente date, una pentru partea reală și una pentru partea imaginară. Ambele componente sunt protejate. Cele patru operații aritmetice asupra numerelor complexe se definesc prin patru componente funcție care sunt publice.

Un număr complex este o instanțiere a acestei clase. Se pot instanția atâtea numere complexe cite sunt necesare.

Mai mult decit atât, componentele funcție care definesc cele patru operații aritmetice cu numere de tip *complex* pot fi definite în aşa fel încit operatorii acestor operații să poată să folosească numai pentru date de tipuri predefinite ale limbajului, ci și pentru datele de tip *complex*. Așa de exemplu, funcția pentru adunarea a două numere complexe poate fi definită în aşa fel ca expresia $x+y$ să fie valabilă nu numai pentru cazurile în care x și y sunt de tip întreg sau flotant, ci și cind ei sunt de tip *complex*.

Cu alte cuvinte, este posibil ca valabilitatea de aplicare a unui operator al limbajului să fie extinsă și pentru tipuri abstracte de date. Această posibilitate trebuie privită ca un mijloc de extindere a valabilității operatorilor existenți. Acest mijloc îl vom numi în continuare *supraincărcarea* operatorilor.

Menționăm că, clasele au de obicei și alte componente funcție care permit realizarea unor operații specifice tipurilor abstracte de date. De exemplu, obiectele pot fi inițializate la instanțierea lor printr-o componentă funcție specială, care se numește *constructor*.

De asemenea, o altă componentă funcție specială realizează "distrugerea" obiectului. Ea se numește *destructor*.

Alte componente funcție se definesc pentru a copia obiecte, a face diferite conversii etc.

Toate aceste facilități apropiie mult comportamentul obiectelor instanțiate prin clase de cele care au tipuri predefinite.

Programarea care utilizează tipuri abstracte de date este un stil de programare superior programării modulare. Un astfel de stil de programare se numește *programare prin abstractizarea datelor*. În esență, ea constă în definirea de tipuri abstracte de date pentru fiecare *concept* necesar la rezolvarea unei probleme concrete, concept care nu este predefinit în limbajul de programare utilizat (mai sus s-au indicat conceptele de *stivă* și *număr complex*). În felul acesta programarea devine mai simplă și mai naturală.

De aici și pînă la stilul programării orientate spre obiecte mai este un singur pas.

De obicei, diferite tipuri abstracte de date au elemente comune. Precizarea lor conduce la o "ierarhizare" a tipurilor abstracte de date. În virful ierarhiei se află

clasa ce conține elementele comune celorlalte clase. Ea este clasa cea mai generală. Pe nivelul următor al ierarhiei se află alte clase care conțin elementele clasei din virful ierarhiei, precum și alte componente specifice tipurilor abstracte pe care le definesc. În general, dacă o clasă conține și alte componente decât cele care sunt comune pentru una sau mai multe clase, atunci aceasta se află la un nivel mai inferior al ierarhiei decât clasa care definește elementele comune lor.

Ierarhizarea claselor are o importanță mare, deoarece atributele unei clase rămân valabile pentru clasele de pe nivelul următor ei din ierarhie. Această proprietate se numește *moștenire*.

Pe scurt, moștenirea permite definirea de tipuri abstracte de date noi prin adăugarea de componente dată și/sau funcție la un tip abstract de date deja existent. Prin aceasta se pot elabora simplu componente extensibile.

O clasă care se definește adăugind componente noi unei clase date se numește *clasă derivată*, iar clasa compusă din elementele comune (la care s-au adăugat componentele noi) se numește *clasă de bază*. Această terminologie este utilizată în C++.

Uneori clasa derivată se mai numește și subclasa a clasei de bază, iar clasa de bază se numește superclasa a unei clase deriveate a ei.

Notiunea de moștenire se aplică cu succes la prelucrarea conceptelor ierarhice din lumea reală.

Modelarea ierarhiilor din lumea reală conduce la o ierarhie de tipuri abstracte de date care în limbajul C++ se definește printr-o ierarhie de clase bazată pe proprietatea de *moștenire*.

Un exemplu simplu de ierarhie din lumea reală este o comunitate de persoane împreună cu conducătorii acestei comunități. În acest exemplu simplu, există două concepte și anume conceptul *persoană* și conceptul de *conducător*.

Conceptul de persoană se modeleză printr-o clasă care conține datele unei persoane:

- nume;
- prenume;
- data nașterii;
- localitate;
- domiciliu;
- profesiune;
- studii;
- stare civilă;
- număr copii;
- incadrare;
- vechime în muncă;
- salarizare;

- cod.

Conceptul de conducător deriva din cel de persoana deoarece orice conducător are toate atributele specifice unei persoane. De aceea, conceptul de conducător se modeleză printr-o clasă care derivă din clasa persoana.

Conceptul de conducător presupune elemente noi, cum ar fi de exemplu cunoștințe despre conducere, specializare, funcție etc.

Un alt exemplu simplu este ales din geometrie și se referă la patrulater.

Dacă se consideră clasa care definește conceptul de paralelogram (patrulater cu laturile opuse paralele), atunci conceptul de dreptunghi poate fi modelat printr-o clasă derivată din cea a paralelogramului. Într-adevar, dreptunghiul este un paralelogram la care mai adăugăm condiția ca să aibă un unghi drept. Deci clasa dreptunghi este o clasă derivată a clasei paralelogram, iar aceasta din urmă este o clasă de bază a clasei dreptunghi. Ierarhia poate fi continuată cu pătratul. Într-adevar, pătratul este un dreptunghi cu toate laturile egale, deci clasa corespunzătoare pătratului este derivată din cea a dreptunghiului. De asemenea, clasa pentru dreptunghi este clasa de bază pentru clasa corespunzătoare pătratului.

Deoarece rombul este un paralelogram cu toate laturile egale, clasa corespunzătoare acestuia este și ea o clasă derivată din clasa corespunzătoare paralelogramului. Deci, clasa pentru paralelogram este clasa de bază atât pentru clasa corespunzătoare dreptunghiului cât și pentru cea corespunzătoare rombului.

La rîndul ei, clasa corespunzătoare paralelogramului poate fi și ea derivată dintr-o clasă mai generală corespunzătoare patrulaterelor.

Numărul exemplelor poate fi marit ușor, deoarece lumea reală este plina de ierarhii. De aici decurge importanța mare a conceptului de moștenire care distanțează substanțial stilul programării orientate spre obiecte de cel al programării prin abstractizarea datelor.

În concluzie, programarea procedurală este cea mai veche și ea se află la nivelul cel mai de jos din punctul de vedere al stilului de programare. La nivelul următor se află programarea modulară care permite ascunderea datelor și a procedurilor în module. Ea este suficientă pentru programarea problemelor relativ simple care nu implică utilizări multiple ale conceptelor modelate prin module. De exemplu, dacă conceptul de stivă este realizat printr-un modul, atunci programarea modulară este suficientă dacă în program nu se utilizează în același timp mai multe stive de felul celei definite prin modulul respectiv.

Ambele stiluri (programarea procedurală și cea modulară) sunt suportate de limbajul C.

Nivelul următor, programarea prin abstractizarea datelor, înălță inconvenientele programării modulare pe baza modelării conceptelor ce intervin în rezolvarea problemelor prin tipuri abstracte de date. De exemplu, introducând tipul abstract de date "stivă", se pot defini și utiliza simplu mai multe stive simultan în același program.

Programarea prin abstractizarea datelor nu este suportată de limbajul C.

Nivelul urmator îl constituie programarea orientată spre obiecte. Acest stil de programare, pe lîngă faptul că permite modelarea conceptelor prin tipuri abstracte de date, ea permite în plus și alte facilități dintre care mai sus s-a amintit exprimarea ierarhiilor din cadrul conceptelor prin facilitatea de moștenire.

Programarea prin abstractizarea datelor este suficientă pentru probleme în care intervin concepte individuale care nu au elemente comune și care deci nu conduce la existența unei ierarhii.

Limbajul C++ a fost proiectat pentru a permite definirea și utilizarea tipurilor abstracte de date, precum și ierarhizarea lor prin folosirea conceptului de moștenire.

Programarea orientată spre obiecte implica și alte facilități care însă nu sunt suportate direct de limbajul C++. De altfel, la ora actuală nu există un limbaj care să suporte toate concepțiile programării orientate spre obiecte.

În carte se descriu elementele de bază ale limbajului C++ și totodată se dau exemple de utilizare a lor în programare.

Cartea apare în patru volume intitulate:

Limbajul C - volumul 1;

Limbajul C++ - volumul 2;

Limbajele C și C++ în aplicații - volumul 3;

Probleme de optimizare, grafică, programe de sistem -volumul 4.

În primul volum se face o descriere elementară a limbajului C. Această descriere este însoțită de o serie de exemple și programe simple care permit o mai bună înțelegere și fixare a elementelor limbajului C.

Majoritatea exemplelor au un caracter didactic. Cu toate acestea, cititorul va întâlni unele exemple și programe sau funcții care pot interveni în practica de zi cu zi a programării în limbajul C.

Autorul insistă asupra importanței rulării programelor prezentate în carte și propune ca ele să fie rulate și înțelese pe măsură ce se avansează cu citirea cărții.

Programele din primul volum pot fi compilate și executate folosind mediile integrate de dezvoltare Turbo C sau Turbo C++ implementate de firma BORLAND.

În cazul în care se folosește mediul Turbo C se va folosi extensia .C la fișierele sursă, iar în cazul lui Turbo C++ extensia .CPP.

Programele din volumul 2 se pot compila și executa numai sub mediul Turbo C++ sau Borland C++ și fișierele sursă vor avea extensia .CPP.

În volumul 3 se întâlnesc atât programe care nu utilizează elemente specifice limbajului C++ cit și programe care utilizează astfel de elemente. Programele din prima categorie pot fi compilate și executate folosind ambele medii amintite mai sus. Evident cele din categoria a doua implică utilizarea mediului Turbo C++ sau Borland C++.

Volumul 2, conține o descriere elementară a facilităților oferite de limbajul C++, care este însoțită de numeroase exemple și programe menite să faciliteze înțelegerea și fixarea cunoștințelor respective.

Abordarea programelor din volumele 3 și 4 este posibilă numai după înșușirea temeinică a cunoștințelor și facilităților limbajelor C și C++ prezentate în primele două volume ale cărții.

Autorul recomandă programatorilor incepători să parcurgă volumele 1 - 2 de mai multe ori înainte de a trece la studierea programelor din volumele 3 și 4.

Limbajele C și C++ având un caracter universal, volumele 3 și 4 conțin programe pentru rezolvări de probleme de sistem, probleme orientate spre calcule științifice sau prelucrări de date.

Limbajul C se utilizează în aplicații care nu implică elemente specifice programării orientate spre obiecte:

- tipuri abstracte de date;
- moștenire;
- supraincărcarea operatorilor etc.

Majoritatea problemelor de acest fel sunt orientate spre calcule științifice, ca de exemplu:

- rezolvări de ecuații;
- sisteme de ecuații liniare;
- probleme de optimizări etc.

În schimb, aplicațiile de sistem și de grafică se abordează utilizând facilitățile programării orientate spre obiecte.

Volumul I care descrie **limbajul C** apare împărțit în două părți:

Partea 1 conține:

- noțiuni de bază;
- intrări/ieșiri standard;
- expresii;
- instrucțiuni;
- clase de memorie;
- pointeri;
- recursivitate.

Partea a 2-a conține:

- tipul utilizator;
- liste;
- arbori;
- tabele;
- sortare;
- procesare;
- intrări ieșiri;
- funcții standard;
- gestiunea ecranului în mod text și grafic.

1. NOIȚIUNI DE BAZĂ

Un program conține una sau mai multe *funcții*. Dintre acestea, una este funcția *principală*.

Fiecare funcție are un *nume*. Numele funcției principale este *main*. Celelalte funcții au nume definite de utilizator.

Programul se păstrează într-un *fișier* sau mai multe. Fișierele au extensia *.c* pentru limbajul C și *.cpp* pentru limbajul C++.

Un fișier care conține un program scris în C sau C++ sau care conține numai o parte a acestuia se va numi *fișier sursă*. Prin compilarea unui fișier sursă rezultă un *fișier obiect*. Acesta are extensia *.obj*.

Fișierele sursă care intră în compunerea unui program pot fi compilate împreună sau separat. În urma unei compilări rezultă un fișier obiect. Fișierele obiect, corespunzătoare unui program, pot fi reunite într-un program executabil prin *ediția de legături (link-editare)*. În urma link-editării rezultă un *fișier executabil*. Acesta are extensia *.exe*.

1.1. Nume

Un *nume* este o succesiune de litere și eventual și cifre, primul caracter fiind literă. În calitate de *litere* se pot utiliza *literele mici și mari* ale alfabetului englez, precum și caracterul *subliniere* (_).

Numărul de caractere care intră în compunerea unui nume nu este limitat.

În mod implicit, numai primele 32 de caractere dintr-un nume sunt luate în seamă. Aceasta înseamnă că două nume diferă între ele numai dacă ele diferă în primele 32 de caractere ale lor.

Mentionăm că sistemele integrate de dezvoltare Turbo C și Turbo C++ permit utilizatorului să modifice această limită de 32.

Exemple de nume:

```
x  
i  
al  
a_1  
a1b2c3  
A×Y  
__ (caracterul subliniere)  
_1  
a_  
acesta_este_un_nume  
Acesta_Este_Tot_un_NUME
```

Se recomandă ca numele să fie sugestiv, adică el să sugereze pe cît posibil scopul alegerii lui sau a datei pe care o reprezintă.

(continuare)

La scrierea numelor se folosesc frecvent literele mici. Adesea, cind un nume se formează din concatenarea mai multor prescurtări de cuvinte, fiecare cuvint începe în numele respectiv cu o literă mare.

Exemple:

prodScal sau ProdScal - pentru produs scalar;
prodMat sau ProdMat - pentru produsul a două matrice.

1.2. Cuvinte cheie

Există un număr de cuvinte imprumutate din limba engleză care au o utilizare predefinită. Utilizatorul nu poate să dea o altă utilizare acestor cuvinte. Ele se numesc *cuvinte cheie*.

Cuvintele cheie se scriu cu litere mici. Deci cuvintele cheie sunt nume cu destinații speciale.

Exemple:

```
if  
while  
for  
break  
class etc.
```

Sensul cuvintelor cheie va fi explicitat pe măsură ce se vor descrie construcțiile în care ele apar.

1.3. Tipuri de date de bază

Orice limbaj de programare oferă programatorului un număr de tipuri de date de bază.

Tipurile de bază se specifică prin cuvinte cheie. Acestea, în ordine alfabetică sunt:

```
char double float int long short signed unsigned
```

În tabela de mai jos se indică reprezentarea tipurilor de bază.

Reprezentarea tipurilor de bază - Tabela 1

Specificarea tipului	Dimensiune în biți	Modul de reprezentare
int	16	întreg reprezentat prin complement față de doi
short	16	idem

Specificarea tipului	Dimensiune în biți	Modul de reprezentare
long	32	idem
unsigned	16	întreg fără semn
unsigned long	32	idem
char	8	codul ASCII al caracterului
float	32	reprezentare flotantă în simplă precizie
double	64	reprezentare flotantă în dublă precizie
long double	80	reprezentare flotantă în dublă precizie

Datele de tip caracter pot fi specificate prin:

unsigned char

sau

signed char

În primul caz, data se presupune că este un întreg în intervalul [0,255] (întreg fără semn).

În cel de al doilea caz, data se presupune că aparține intervalului [-128,127] (întreg cu semn).

Datele specificate numai prin *char* au o interpretare implicită. Această interpretare poate fi definită prin intermediul mediului integrat de dezvoltare Turbo C sau Turbo C++. În mod normal se consideră că datele de tip *char* sunt implicit numere fără semn.

Pentru a preîmpinge apariția unor erori prin interpretări implice nedorite ale datelor de tip caracter, se recomandă să se utilizeze specificările explicate *unsigned char* și *signed char*.

În tabela 2 se indică intervalele de valori ale tipurilor de bază.

Intervalele de valori pentru tipurile de bază - Tabela 2

Specificarea tipului	Intervalul de valori
int	[-32768,32767]
short	[-32768,32767]
long	[-2147483648,2147483647]

(continuare)

Specificarea tipului	Intervalul de valori
unsigned	[0,65535]
unsigned long	[0,4294967295]
unsigned char	[0,255]
signed char	[-128,127]
float	valoarea absolută a unei date diferite de zero apartine intervalului [3,4*10**(-38);3,4*10**(38)]
double	valoarea absolută a unei date diferite de zero apartine intervalului [1,7*10**(-308);1,7*10**(308)]
long double	valoarea absolută a unei date diferite de zero apartine intervalului [3,4*10**(-4932);1,1*10**(4932)]

Observație:

Tipul predefinit *short* este identic cu tipul predefinit *int* la calculatoarele compatibile IBM PC. La alte calculatoare ele pot fi diferite, de exemplu tipul *short* rămâne reprezentat pe 16 biți, iar tipul *int* poate să fie la fel ca și tipul *long*.

Tipul *unsigned long* poate fi scris inversind cuvintele cheie:

long unsigned.

1.4. Structura unei funcții

O funcție are următoarea structură:

```
tip nume(lista declarațiilor parametrilor formali)
{
    declarații
    instrucțiuni
}
```

Primul rind din formatul de mai sus reprezintă *antetul* funcției. Partea inclusă între acolade, împreună cu acoladele, formează *corpu* funcției.

În cazul tipurilor predefinite, *tip* din antetul funcției este un cuvint cheie. El definește tipul valorii returnate de funcție.

În limbajul C există două categorii de funcții. O primă categorie conține

funcțiile care la revenirea din ele returneză o valoare în punctul de apel. Tipul acestei valori se definește prin tip din antetul funcției. Cealaltă categorie de funcții conține funcțiile care nu returneză nici o valoare la revenirea din ele.

Pentru aceste funcții se va folosi cuvintul cheie *void* în calitate de *tip*. El semnifică lipsa unei valori returnate la revenirea din funcție.

O funcție poate avea zero sau mai mulți parametri. Lista declarațiilor parametrilor formali este vidă în cazul în care funcția nu are parametri formali. În acest caz antetul funcției se reduce la:

tip nume()

Menționăm că absența parametrilor poate fi indicată explicit folosind cuvintul cheie *void*. Astfel, antetul de mai sus poate fi scris și sub forma:

tip nume(void)

Exemplu:

1. *void f(void)*
{
...
}

Funcția *f* nu are parametri. Ea nu returnează nici o valoare la revenirea din ea.

2. *void f()*
{
...
}

Format identic cu cel din exemplul precedent.

3. *int g()*
{
...
}

Funcția *g* nu are parametri. La revenirea din ea se returnează o valoare întreagă de tip *int*.

4. *double h(void)*
{
...
}

Funcția *h* nu are parametri. La revenirea din ea se returnează o valoare flotantă în dublă precizie.

În cazul în care funcția are parametri, declarațiile parametrilor formali se includ între parantezele rotunde prezente după numele funcției și se separă prin virgulă dacă sunt mai multe.

Parametri se utilizează pentru a permite transferuri de date la o funcție în momentul apelului ei. Acest mecanism de transfer al datelor prin intermediul parametrilor ne permite construirea de funcții facind abstracție de valorile concrete care vor fi prezente abia la execuția programului. În momentul compilării este necesara numai cunoașterea *tipurilor* valorilor pe care le vor primi parametri la execuție. Aceste tipuri sunt definite prin declarațiile parametrilor respectivi, declarații care, așa cum am văzut mai sus, se indică în antetul funcției.

Parametri declarați în antetul unei funcții și care apoi se utilizează în corpul funcției, se numesc *formali* pentru a sublinia faptul că ei nu reprezintă valori concrete, ci numai în locul acestora pentru a putea exprima procesul de calcul realizat prin funcție. Ei se concretizează la execuție prin apelurile funcției. Valorile parametrilor formali se definesc la fiecare apel al unei funcții prin așa numiți parametri *reali, efectivi sau concreți*.

Utilizarea parametrilor formali la implementarea funcțiilor și atribuirea de valori concrete pentru ei la execuție, reprezintă un prim nivel de abstractizare în programare. Acest mod de programare este cel mai vechi. El se numește *programare procedurală* și realizează un proces de *abstractizare prin parametri*.

Declarațiile parametrilor formali sunt asemănătoare cu cele ale variabilelor și vor fi definite într-un paragraf ulterior.

Observații:

1. Inițial antetul unei funcții a avut următorul format:

tip nume(lista parametrilor formali)
declarațiile parametrilor formali

unde *tip* este prezent numai pentru funcții care returnează o valoare la revenirea din ele. Cuvintul cheie *void* a fost introdus ulterior. De asemenea, cuvintul cheie *int* nu era nevoie să fie prezent, considerindu-se că orice funcție care returnează o valoare a carui tip nu este specificat, returnează o valoare de tip *int*.

Ulterior s-a constatat că această libertate în omiterea tipului este o sursă de erori și de aceea se recomandă ca tipul să fie totdeauna prezent în antetul unei funcții. În cazul limbajului C++ controalele cu privire la tipul valorii au fost întărite chiar pentru a elimina posibilitățile de apariție a erorilor la revenirea din funcții.

Lista parametrilor formali este fie vidă, cind funcția nu are parametri, fie se compune dintr-un nume sau mai multe separate prin virgulă. În acest caz, declarațiile parametrilor formali, dacă ei există, se dau imediat după paranteza închisă. Acest format poate fi folosit, atât în limbajul C, cât și în limbajul C++, dar compilatorul C++ remarcă într-un avertisment că acest format este învechit.

De asemenea, amintim că cele două formate pot fi folosite împreună, adică

putem indica în parantezele rotunde o parte din parametri prin numele lor, iar restul prin declarațiile lor.

În continuare se dău declarațiile parametrilor care în parantezele rotunde au fost prezente numai prin numele lor.

Cu toate acestea, se recomandă utilizarea formatului în care toate declarațiile parametrilor sunt incluse în parantezele rotunde (formatul de la începutul paragrafului).

2. Pentru funcția principală se pot utiliza antetele:

```
int main()
int main(void)
void main()
void main(void)
main()
main(void)
```

Primele două antete presupun că funcția *main* returnează o valoare întreagă la revenirea din ea în sistemul de operare.

Adesea se obișnuiește utilizarea formatului fără tip:

main()

Amintim că funcția *main* poate avea și parametri. Aceștia, cind sunt prezenti, permit utilizarea de către program a unor valori definite la lansarea programului. Ulterior vom preciza modul de utilizare al acestor parametri.

1.5. Comentariu

În limbajul C, ca și în alte limbaje de programare se pot folosi comentarii. Un comentariu începe cu succesiunea de caractere

*/**

și se termină cu

**/*

El se compune din orice caracter admis în setul de caractere al limbajului. Evident, în interiorul unui comentariu nu se va folosi succesiunea

**/*

care termină un comentariu.

În limbajul C++ s-a mai introdus o convenție pentru a insera comentarii. Astfel, în C++ un comentariu poate începe prin succesiunea

//

Un astfel de comentariu se termină pe același rind (la sfîrșitul lui) pe care se află și începutul lui.

Comentariile sunt explicații pentru programatori. Ele nu au nici un efect asupra compilatorului și sunt omise la compilare.

Un comentariu se poate insera oriunde în program unde este legal să apară un spațiu, un tabulator sau caracterul de rind nou.

În general, se recomandă introducerea de comentarii după antetul funcției, care să precizeze:

- acțiunea sau acțiunile realizate de funcție;
- formatele datelor de intrare și ieșire;
- algoritmii codificați prin funcția respectivă dacă sunt complecsi;
- diferite limite impuse datelor de intrare etc.

De asemenea, se pot insera comentarii în corpul funcției unde se consideră că sunt necesare unele explicații.

Comentariile trebuie să fie exprimări clare care să nu conducă la ambiguități și să nu conțină afirmații eronate.

1.6. Constante

O constantă are un *tip* și o *valoare*. Atât tipul, cât și valoarea, sunt determinate de caracterele care intră în compunerea constantei. Valoarea unei constante nu poate fi schimbată în timpul execuției programului în care a fost utilizată.

1.6.1. Constante întregi

Constantele întregi pot fi scrise în sistemul de numerație cu baza 8, 10 sau 16.

O *constantă zecimală întreagă* este un sir de cifre zecimale care are prima cifră diferită de zero. Constantele zecimale se reprezintă prin complement față de doi pe 16 biți sau pe 32 de biți dacă nu încap pe 16 biți.

Constantele întregi reprezentate pe 16 biți sunt de tip *int*, iar cele reprezentate pe 32 biți sunt de tip *long*.

În cazul în care noi dorim să reprezentăm o constantă zecimală pe 32 de biți, chiar dacă ea se poate reprezenta pe 16 biți, constanta respectivă trebuie să o terminăm prin *L* sau *l*.

Exemple:

Reprezentare externă	Reprezentare internă în binar
12345	0011000000111001
123456789	0000011101011011100110100010101
12345L	000000000000000001100000111001

O constantă zecimală are tipul *unsigned* dacă se termină prin litera *U* sau *u*. O

astfel de constantă se reprezintă pe 16 biți dacă nu este mai mare decât 65535 și pe 32 de biți în caz contrar. În cazul în care dorim ca o constantă întreagă fără semn mai mică decât 65536 să se reprezinte pe 32 de biți, constantă respectivă se va termina prin una din următoarele succesiuni de litere:

ul
lu
LU
UL

Constantele întregi fără semn pot fi utilizate pentru a economisi memorie. Astfel, constantele de tip *int* din intervalul [32768,65535] se păstrează pe 32 de biți, în schimb constantele de tip *unsigned* din același interval se reprezintă pe 16 biți.

Exemple:

40000u	constantă întreagă de tip <i>unsigned</i> reprezentată pe 16 biți
40000U	idem
4294967295	constantă întreagă de tip <i>unsigned</i> reprezentată pe 32 de biți
40000lu	idem
40000ul	idem
40000LU	idem
40000UL	idem

O *constantă octală* întreagă este o succesiune de cifre octale (0 - 7) precedată de un zero nesemnificativ. O astfel de constantă se păstrează pe 16 biți dacă aceștia îi sunt suficienți și pe 32 de biți în caz contrar. În cazul în care o constantă octală se termină prin *l* sau *L*, ea se păstrează pe 32 de biți chiar dacă sunt suficienți 16 biți pentru reprezentarea ei.

Constantele octale sunt de tip *unsigned* dacă se reprezintă pe 16 biți și *unsigned long* dacă se reprezintă pe 32 de biți.

O *constantă hexazecimală* întreagă este o succesiune de cifre hexazecimale precedată de

0x
sau
0X.

În rest, ea are aceleași proprietăți ca și o constantă octală.

Cifrele hexazecimale se obțin extinzând cifrele zecimale cu literele mici sau mari de la *A* la *F*:

Litera care reprezintă o cifră hexazecimală	Valoare
a sau A	10
b sau B	11
c sau C	12
d sau D	13
e sau E	14
f sau F	15

Exemple:

Constantă	Tipul constantei	Lungimea reprezentării
123	constantă zecimală de tip <i>int</i>	16 biți
0123	constantă octală de tip <i>unsigned</i>	16 biți
40000	constantă zecimală de tip <i>long</i>	32 biți
040000	constantă octală de tip <i>unsigned</i>	16 biți
0123456	idem	idem
123L	constantă zecimală de tip <i>long</i>	32 biți
0123l	constantă octală de tip <i>unsigned long</i>	32 biți
0x123	constantă hexazecimală de tip <i>unsigned</i>	16 biți
0xa1b2c3	constantă hexazecimală de tip <i>long unsigned</i>	32 biți
0XAFCFL	constantă hexazecimală de tip <i>long unsigned</i>	32 biți

1.6.2. Constante flotante

O constantă flotantă reprezintă un număr *rational*. Ea se compune din:

- o parte întreagă care poate fi și vidă;
- o parte fracționară care poate fi și vidă;
- un exponent care poate fi și vid.

Evident, nu pot fi vîde toate părțile indicate mai sus. La scrierea unei constante flotante este necesar să fie prezentă fie partea fracționară, fie exponentul precedat de partea întreagă.

Partea întreagă este o constantă zecimală.

Partea fracționară se compune din caracterul punct după care urmează o succesiune de cifre zecimale. Succesiunea respectivă poate fi vidă numai în cazul în care partea întreagă este prezentă.

Exponentul începe cu litera e mică sau mare, după care poate fi prezent un semn optional (plus sau minus) și un sir de cifre zecimale. Exponentul definește un factor care exprimă o putere a lui 10.

În exemplele de mai jos și în continuare vom folosi notația ****** pentru operația de ridicare la putere. Deci a la puterea b se va nota prin

$a^{**}b$

Exemple:

Constantă flotantă	Valoare
123.	123
123.7	123,7
.25	0,25
78e4	$78*10^{**4}$
.1E-3	$0,1*10^{**(-3)}$
123.456e2	12345,6
1234.567e-4	0,1234567

Constanțele flotante se reprezintă în dublă precizie (64 biți).

Pentru a reprezenta o constantă flotantă în simplă precizie este suficient să terminăm constanta respectivă prin litera *f* sau *F*.

O constantă flotantă terminată prin *l* sau *L* se reprezintă pe 80 de biți și are tipul *long double*.

1.6.3. Constantă caracter

Prelucrarea datelor cu ajutorul calculatoarelor arc în vedere, printre altele, posibilitatea lucrului pe caractere. În acest scop, caracterele se codifică folosind coduri numerice. Cele mai utilizate coduri sunt codurile EBCDIC (Extended Binary Coded Decimal Interchange Code) și ASCII (American Standard Code for Information Interchange).

La calculatoarele compatibile IBM PC se utilizează codul ASCII. Caracterele acestui cod le împărțim în:

- | | |
|-------------------|---|
| caractere | - Codul lor este în intervalul [0,31] la care se adaugă și codul negrafcice |
| spațiu | - Are codul 32. |
| caractere grafice | - Codul lor este în intervalul [33,126]. |

Caracterele grafice împreună cu spațiul formează setul de caractere imprimabile.

O constantă caracter are ca valoare codul ASCII al caracterului pe care-l reprezintă. Ea are tipul *int*.

O constantă caracter corespunzătoare unui caracter imprimabil se reprezintă prin caracterul respectiv inclus între caractere apostrof.

Exemple:

Constantă caracter	Valoare
'a'	97
'A'	65

(continuare)

Constantă caracter	Valoare
'0'	48
'1'	32
'*'	42

O excepție de la regula de mai sus o reprezintă caracterele apostrof și bara oblică inversă (*backslash*). Astfel, caracterul *backslash* se reprezintă prin două caractere *backslash* incluse între caractere apostrof:

\\"

La reprezentarea caracterului apostrof se utilizează caracterul *backslash* urmat de un caracter apostrof:

'\'

De aceea, constanta caracter apostrof se poate reprezenta prin:

'\\'

Caracterul *backslash* se poate utiliza pentru a defini constante caracter și pentru caractere negrafcice. Așa de exemplu, pentru a defini constanta caracter corespunzătoare tabulatorului se folosește notația cu *backslash*:

'\t'

Deci

'\t'

definește constanta caracter tabulator orizontal. Ea are valoarea 9.

În mod analog, constanta caracter rind nou (*newline*) se reprezintă prin:

'\n'

Ea are valoarea 10.

Se obișnuiește să se spună că *backslash* introduce o secvență *escape*.

Mai jos se indică reprezentarea unor constante caracter prin secvențe *escape*.

Constantă caracter	Codul ASCII	Denumirea caracterului	Utilizare
'\a'	7	BEL	activare sunet
'\b'	8	BS	revenire cu un spațiu (<i>Backspace</i>)
'\t'	9	HT	tabulator orizontal
'\n'	10	LF	rind nou (<i>Line Feed</i>)
'\v'	11	VT	tabulator vertical
'\f'	12	FF	salt de pagină la imprimantă (<i>Form Feed</i>)
'\r'	13	CR	retur de car - poziționează cursorul în coloana 1 din rindul curent (<i>Carriage Return</i>)

La aceste notății adăugăm și utilizările secvenței *escape* pentru a reprezenta caracterele *backslash*, apostrof și ghilimele:

Constantă caracter	Codul ASCII	Denumirea caracterului
'\"	34	ghilimele
'\'	39	apostrof
'\\'	92	backslash

Secvența *escape* poate fi folosită pentru a defini constante caracter pentru orice caracter al codului ASCII. Aceasta se realizează folosind codul caracterului respectiv. Astfel, construcția:

'\ddd'

unde:

d

- Este o cifră octală.

- Reprezintă constanta caracter corespunzătoare codului ASCII egal cu valoarea întregului octal ddd.

Exemple:

'\a' și '\7'

- Reprezintă aceeași constantă caracter corespunzătoare caracterului BEL.

'\b' și '\10'

- Reprezintă constanta caracter corespunzătoare caracterului backspace (8 din sistemul decimal se reprezinta prin numărul 10 în sistemul cu baza opt).

Se observă că în cadrul unei secvențe escape numerele octale nu mai trebuie să fie precedate de un zero nesemnificativ. În acest caz numarul se consideră automat în sistemul cu baza 8.

'\" și '\42'

- Reprezintă constanta caracter care corespunde caracterului ghilimele.

La calculatoarele compatibile IBM PC se utilizează un cod *ASCII extins*. Acesta conține 256 de coduri care sunt valori în intervalul [0,255].

Codurile din intervalul [0,127] corespund caracterelor codului ASCII obișnuit, iar cele din intervalul [128,255] corespund unor caractere care au o utilizare specială.

Constantele caracter corespunzătoare caracterelor de cod ASCII din intervalul [128,255] se definesc prin secvențe escape în care se indică codul acestora. De exemplu, constanta caracter

'\377'

corespunde caracterului care are codul ASCII egal cu 255, adică codul ASCII maxim.

Deoarece o constantă caracter este de tip *int*, ea se păstrează pe 16 biți. Pentru constantele caracter mai mari decât 127, se pune problema extensiei semnului. Programatorul poate opta pentru constantă caracter cu semn sau fără semn. Pentru a folosi în mod implicit constantă caracter fără semn se va alege, în mediul integrat de dezvoltare Turbo C sau Turbo C++, alternativa *Unsigned Characters* în submeniu *Code Generation* al submeniului *Compile* din meniu *Options*.

1.6.4. Sir de caractere

O succesiune de zero sau mai multe caractere incluse între ghilimele formează o constantă sir sau un sir de caractere.

La scrierea caracterelor din compunerea unui sir de caractere se pot utiliza secvențe escape.

Exemple:

"Acesta este un sir de caractere"
"Prin secvența escape \" se reprezinta ghilimele"
"Prin secvența escape \\ se reprezinta backslash"
"Apostroful se reprezinta obisnuit"
"s'a"
"alte\\secvențe\\escape\\nintr-un sir"

Un sir poate fi continuat pe rindul următor folosind caracterul backslash. În acest scop se tastează backslash la sfârșitul rindului care se continuă, se trece pe rindul următor (acționând tastă Enter) și se continuă cu tastarea caracterelor sirului respectiv.

Caracterul care precede pe backslash se va concatena cu primul caracter de pe rindul următor.

Caracterele unui sir de caractere se păstrează în memorie într-o zonă contiguă, prin codurile lor ASCII. După ultimul caracter al sirului se păstrează caracterul *NUL*, adică valoarea zero. Aceasta joacă rolul de marcaj de sfârșit al oricărui sir de caractere. Din cauza acestui marcaj, trebuie să facem distincție între o constantă caracter care corespunde unui caracter și sirul de caractere care este format din același caracter. Astfel, constantă caracter

'A'

se păstrează în memorie într-un octet prin valoarea 65, pe cind sirul de caractere

"A"

ocupa o zonă de doi octeți: în primul octet se păstrează valoarea 65, iar în al doilea caracterul *NUL*, adică valoarea zero.

În concluzie, un sir de caractere se păstrează în memorie într-o succesiune de octeți al cărui numar este egal cu numarul caracterelor sirului respectiv mărit cu

1, deoarece sirul se termină totdeauna prin caracterul NUL.

Observații:

1. Fie sirul

"a\b"

Acest sir are în compunerea sa:

- caracterul a de cod ASCII 97;
- caracterul SOH de cod ASCII 1;
- caracterul b de cod ASCII 98;
- caracterul NUL de cod ASCII 0.

Dacă în locul sirului de mai sus se dorește un sir în care b să fie înlocuit prin caracterul 3, atunci scrierea:

"a\3"

nu este corectă. Într-adevăr, acest sir are în compunerea sa:

- caracterul a de cod ASCII 97;
- caracterul VT de cod ASCII 11;
- caracterul NUL de cod ASCII 0.

Pentru a reprezenta caracterul 3 în acest caz, va fi nevoie de încă o secvență escape. Secvența escape pentru caracterul 3 este \63, deci sirul de caractere respectiv se scrie astfel:

"a\1\63"

2. Sirul de caractere

"\1751"

are în compunerea sa:

- caracterul } de cod ASCII 125 (175 în octal);
- caracterul 1 de cod ASCII 49;
- caracterul NUL de cod ASCII 0.

Ei poate fi scris mai simplu astfel:

"}1"

3. Caracterul NUL nu poate fi utilizat decât la sfârșitul unui sir de caractere. Aceasta, deoarece un sir de caractere totdeauna se termină la apariția caracterului NUL.

1.7. Caractere sau spații albe (white spaces)

În cele ce urmează, prin caracter sau spațiu alb vom înțelege unul din următoarele caractere:

- spațiu (' ');
- tabulator orizontal ('\t');
- caracterul de rind nou (newline) ('\n').

Menționăm că setul caracterelor albe diferă pentru diferite implementări ale limbajului C. Setul de caractere indicat mai sus este un set minimal comun tuturor implementărilor.

Amintim că un comentariu poate fi inserat într-un program, oriunde este legal să apară un spațiu alb.

1.8. Variabile simple, tablouri și structuri

Într-un program utilizăm alături de date constante și date variabile care își schimbă valorile în timpul execuției programului.

Dacă la o dată constantă ne putem referi folosind caracterele din compunerea ei, la o dată variabilă trebuie să ne referim altfel. Cel mai simplu mod este acela de a denumi data respectivă. Numele datei ne permite accesul la valoarea ei, precum și schimbarea valorii dacă este necesar.

În cazul în care o dată nu are legături cu alte date (de exemplu de ordine), vom spune că ea este o dată *izolată*. Numele unei date izolate se spune că reprezintă o *variabilă simplă*.

Unei date izolate îi corespunde un *tip*. În cursul execuției programului se pot schimba valorile unei date variabile dar nu și tipul ei.

Corespondența dintre numele unei date variabile și tipul ei se definește printr-o *declarație*.

Adesea, într-un program este util să considerăm grupe de date. Gruparea datelor se poate face în mai multe moduri.

Un mod simplu de a grupa date este acela de a considera date de *același tip*, în așa fel încât grupa respectivă să formeze o *mulțime ordonată* de elemente la care să ne putem referi folosind *indici*. O astfel de grupă se spune că formează un *tablou*. Unui tablou îi se dă un nume. Tipul comun al elementelor unui tablou este și tipul tabloului respectiv. De exemplu, o mulțime ordonată de întregi reprezintă un tablou de tip *întreg*.

În cazul în care elementele care se grupează într-un tablou sunt ele însele tablouri, vom avea nevoie de mai mulți indici, pentru a ne referi la ele. În cazul în care se utilizează un singur indice pentru a ne referi la elementele tabloului, spunem că tabloul este *unidimensional*. Dacă se folosesc n indici, se spune că tabloul este *n-dimensional*.

Exemple simple de tablouri unidimensionale sunt vectorii care au componente de același tip. Așa de exemplu, un vector de componente întregi este un tablou unidimensional de tip *întreg*.

O matrice de elemente întregi este un exemplu de tablou bidimensional de tip *întreg*.

Referirea la elementele unui tablou se face printr-o *variabilă cu indici*.

O variabilă cu indici se compune din numele tabloului urmat de valorile indiciilor, fiecare indice fiind reprezentat printr-o expresie inclusă între paranteze pătrate.

Valoarea *inferioară* a indicilor este egală cu *zero*. De exemplu, dacă *vect* este un tablou unidimensional de 10 elemente, atunci ne referim la elementele lui cu ajutorul variabilelor cu indici:

<i>vect[0]</i>	primul element
<i>vect[1]</i>	al doilea element
...	
<i>vect[9]</i>	ultimul element

Dacă *mat* este un tablou bidimensional care definește o matrice de 3 linii și 2 coloane fiecare, atunci elementele acestui tablou pot fi referite prin:

<i>mat[0][0]</i>	<i>mat[0][1]</i>	elementele primei linii
<i>mat[1][0]</i>	<i>mat[1][1]</i>	elementele celei de a doua linii
<i>mat[2][0]</i>	<i>mat[2][1]</i>	elementele celei de a treia linii

Un alt mod de a grupa date are în vedere prezența unor *relații* între datele care se grupează. În acest caz datele care se grupează se spune că formează o *structură*. Ele nu neapărat au un același tip.

Prin *structură* înțelegem o mulțime ordonată de elemente "înrudite", ordonare definită de utilizator în vederea simplificării utilizării grupei de date respective.

Unei structuri îi se atașează un *nume*. De asemenea, se atașează cîte un nume fiecarei componente ale unei structuri, nume care se utilizează la referirea componentelor respective.

Un exemplu simplu de structură este data calendaristică. Aceasta este o grupă care se compune din trei date înrudite:

- zi;
- lună;
- an.

Aceste trei date nu neapărat au toate același tip. De exemplu, ziua și anul pot fi de tip *întreg* (*int*), iar luna poate fi de tip *nemonic* dacă ea se reprezintă prin denumire.

Un alt exemplu simplu de structură îl reprezintă numerele complexe. Un număr complex este o mulțime ordonată de două numere, fiecare de tip *double*. Primul număr reprezintă partea reală a numărului complex, iar cel de al doilea, partea lui imaginată.

Structura, ca și variabilele simple și tablourile, corespunde și ea unui tip de dată. Tipul unei structuri nu are nimic comun cu tipurile componentelor sale. Ea este un tip *nou*, diferit de cele predefinite în limbaj. Un astfel de tip se spune că este un *tip definit de utilizator* sau mai scurt *tip utilizator*. De exemplu, data calendaristică definită mai sus reprezintă un tip nou de date, diferit de cele predefinite. De asemenea, numerele complexe reprezentate prin structuri de perechi de numere flotante în dublă precizie, definesc tipul complex.

Tipurile utilizator se definesc printr-o *declarație* care, așa cum vom vedea ulterior, a suferit mai multe modificări. Tot printr-o declarație se stabilește legătura dintre numele unei date structurate și tipul ei.

În general, prin *tip* înțelegem o mulțime de date împreună cu operațiile care pot fi efectuate cu datele respective.

De exemplu, tipul *int* se definește prin mulțimea numerelor intregi din intervalul [-32768,32767] reprezentate prin complement față de doi pe 16 biți. Asupra acestor date sunt definite o serie de operații, cum ar fi cele 4 operații aritmetice, operații de comparație etc.

În cazul tipurilor predefinite sunt predefinite atât mulțimea și reprezentarea datelor tipului respectiv, cât și operațiile cu aceste date.

În cazul tipurilor utilizator se definesc de către utilizator, mulțimea și reprezentarea datelor tipului respectiv prin intermediul construcției *struct*. Utilizatorul definesc operațiile cu aceste date prin intermediul unor funcții. De exemplu, în cazul tipului *complex* se definește reprezentarea numerelor complexe printr-o construcție *struct*. Operațiile cu numerele complexe se pot realiza apelând funcții corespunzătoare.

Tipurile utilizator pot fi definite ca mai sus prin intermediul facilităților existente în limbajul C. Un neajuns al tipurilor utilizator definite în limbajul C este faptul că nu se stabilește nici o legătură între reprezentarea tipului și funcțiile care definesc operațiile cu datele respective. Acești neajunsuri au fost înălțat în C++ prin introducerea noțiunii de *clasa*. Tipurile definite prin intermediul claselor se numesc *tipuri abstracte* de date.

În cazul unui tip abstract de date se asigură legătura dintre reprezentările datelor și funcțiile care definesc operațiile asupra lor. Ca rezultat al acestei legături se obține protecția datelor, utilizatorul neavând acces direct la ele decit numai prin intermediul funcțiilor definite în acest scop.

Construcția *struct*, precum și definirea claselor se vor prezenta mai tîrziu.

1.9. Declarația de variabilă simplă

În limbajul C nu există declarații implicate. Aceasta înseamnă că orice variabilă înainte de a fi utilizată trebuie declarată.

Declarația unei variabile simple stabilește legătura dintre numele variabilei și tipul valorilor pe care le poate avea variabila respectivă.

În cea mai simplă formă, o declarație de variabilă simplă are formatul:

tip lista_de_nume;

În calitate de *tip* putem folosi cuvintele cheie ale tipurilor predefinite.

Lista_de_nume se compune dintr-un nume de variabilă simplă sau mai multe separate prin virgule.

Exemple:

1. `int i,j;`

Variabilele *i* și *j* sunt variabile simple de tip *int*. Compilatorul aloca pentru fiecare o zonă de memorie de 16 biți.

Prin *i* ne referim la valoarea conținută în zona de memorie alocată variabilei *i*. De asemenea, tot prin intermediul lui *i* putem atribui sau modifica valoarea din zona de memorie alocată variabilei *i*.

2. `char c;`

Variabila *c* este o variabilă simplă de tip *char*. El îi se aloca o zonă de memorie de un octet (8 biți).

În această zonă putem păstra un caracter al codului ASCII extins, prin codul lui. Prin *c* ne putem referi la caracterul respectiv.

3. `long double x;`

Variabila *x* este o variabilă simplă de tip *long double*. El îi se aloca o zonă de memorie de 10 octeți în care se păstrează o valoare flotantă.

1.10. Declarația de tablou

Un tablou, ca orice variabilă simplă, trebuie declarat înainte de a fi utilizat. Declarația de tablou, în forma cea mai simplă, conține *tipul comun* elementelor sale, *numele* tabloului și *limitele superioare* pentru fiecare indice, incluse între paranteze pătrate:

tip nume[lim1][lim2]...[limn];

unde:

tip

- Este un cuvînt cheie pentru tipurile predefinite.

limi

- Este limita superioară a indicelui al *i*-lea; aceasta înseamnă că indicele al *i*-lea poate avea valorile:

0,1,2,...,limi-1

Limitele *limi* (*i*=1,2,...,n) sunt *expresii constante*. Prin expresie constantă înțelegem o expresie care poate fi evaluată la compilare în momentul întlnirii ei de către compilator.

În paragraful precedent am văzut că la elementele unui tablou ne putem referi folosind variabile cu indici.

În limbajul C, numele unui tablou este un simbol care are ca valoare adresa primului său element.

La întlnirea unei declarații de tablou, compilatorul aloca o zonă de memorie necesară pentru a păstra valorile elementelor sale. Numele tabloului respectiv poate fi utilizat în diferite expresii și valoarea lui este chiar adresa de început a zonei de memorie care i-a fost alocată.

Exemple:

1. `int vect[10];`

Declarația de față definește tabloul *vect* de 10 elemente și el are tipul *int*.

Pentru acest tablou se alocă $10 \times 2 = 20$ octeți.

vect este un simbol a cărui valoare este adresa primului său element, adică adresa lui *vect[0]*. Deci *vect[0]* are ca valoare valoarea primului element al tabloului, iar *vect* are ca valoare adresa acestui element.

2. `char tab[100];`

Tabloul *tab* este un tablou unidimensional de tip *char*, care are 100 de elemente. I se alocă 100 de octeți și *tab* are ca valoare adresa elementului *tab[0]*.

3. `double dmat[10][50];`

Tabloul *dmat* este un tablou bidimensional de tip *double*. El reprezintă o matrice de 10 linii și 50 de coloane fiecare. Compilatorul rezervă pentru acest tablou

$$10 \times 50 \times 8 = 4000 \text{ octeți.}$$

La elementele acestui tablou ne referim prin:

dmat[0][0] *dmat[0][1]*...*dmat[0][49]*

dmat[1][0] *dmat[1][1]*...*dmat[1][49]*

...

dmat[9][0] *dmat[9][1]*...*dmat[9][49]*

dmat are ca valoare adresa elementului *dmat[0][0]*.

1.11. Apelul și prototipul funcțiilor

Într-un program o funcție poate avea o definiție și unul sau mai multe apeluri.

Am văzut mai sus că o funcție se definește prin antet urmat de corpul ei. Antetul, de obicei, are formatul:

tip nume(lista declaratiilor parametrilor formali)

Lista declaratiilor parametrilor formali este fie vidă, fie conține o declarație de parametru formal sau mai multe separate prin virgulă. Menționăm că declaratiile parametrilor formali aflate într-o listă de felul celei de sus, nu se termină prin punct și virgulă ca cele pentru variabile simple și tablouri.

Exemple:

1. `int f(int x, double y);`

Funcția *f* are doi parametri:

x de tip *int*

și

y de tip *double*.

Funcția returnează o valoare de tip *int*.

2. `double df(long a, int b, unsigned c);`

Funcția *df* are trei parametri:

a - de tip *long*;

b - de tip *int*;

c - de tip *unsigned*.

Ea returnează o valoare de tip *double*.

O funcție poate fi apelată folosind o construcție de forma:

nume(lista_parametrilor_efectivi)

unde:

nume

- Este numele funcției care se apelează.

lista

- Este fie vida dacă funcția nu are parametri, fie se compune din unul sau mai mulți *parametri efectivi* separați prin virgule.

parametrilor

- Un parametru efectiv este o expresie.

efectivi

- Parametrii efectivi se corespund cu cei formalii prin *ordine* și

tip.

La apel se atribuie parametrilor formalii valorile parametrilor efectivi și apoi execuția se continuă cu prima instrucție din corpul funcției apelate. La revenirea din funcție se ajunge în funcția din care s-a facut apelul și execuția continuă cu construcția următoare apelului.

Pentru a apela o funcție putem utiliza construcția de mai sus urmata de caracterul punct și virgula.

O altă posibilitate este aceea de a folosi construcția de mai sus drept operand al unei expresii. Un astfel de apel este posibil numai pentru funcțiile care returnează o valoare la revenirea din ele. În acest caz valoarea returnată de funcție se folosește la evaluarea expresiei din care s-a făcut apelul.

Un parametru efectiv de la apelul unei funcții poate fi numele unui tablou. În acest caz, în antetul funcției respective parametrul corespunzător îl vom declara ca fiind tablou.

De exemplu, dacă *tab* este un tablou unidimensional declarat ca mai jos:

`int tab[100];`

și *tab* se folosește la apelul funcției *f*:

f(tab);

atunci funcția *f* are antetul:

`void f(int x[100])`

Mentionăm că, limita superioară 100 poate fi omisă la declararea parametrului formal *x*, dar nu și parantezele patrate. Deci aceeași funcție poate avea urmatorul antet:

`void f(int x[])`

În cazul parametrilor formalii care sunt tablouri cu mai multe dimensiuni,

numai limita primului indice poate fi omisă.

Exemplu:

Fie declarația

```
double mat[4][10];
```

și apelul

```
fct(mat);
```

Antetul funcției *fct* poate fi urmatorul:

```
void fct(double mat[4][10]);
```

O funcție poate fi apelată într-un punct al unui fișier sursă dacă în prealabil a fost definită în același fișier sursă.

Exemplu:

```
void f1(void) //definiția funcției f1
{
    ...
}

void f2(void) //definiția funcției f2
{
    ...
    /* se apelaza funcția f1; ea este în prealabil definită */
    f1();
    ...
}
```

Apelurile funcției nu pot fi precedate totdeauna de definiția ei. În astfel de cazuri definiția funcției apelate este înlocuită printr-un aşa numit *prototip* al ei.

Prototipul unei funcții are un format asemănător cu antetul ei. Acesta reprezintă o informație pentru compilator cu privire la:

- tipul valorii returnate de funcție;
- existența și tipurile parametrilor funcției.

Acste informații sunt prezente în antetele funcțiilor. De aceea, un prototip al unei funcții poate fi scris ca și antetul funcției respective, după care se pune punct și virgulă. Există și o formă precurtată pentru prototip și anume aceea în care se omită numele parametrilor:

tip nume(lista tipurilor parametrilor formalii):

Exemple:

1. void f(void);

Prototipul de față indică faptul că *f* este o funcție fără parametri și care nu returnează nici o valoare.

2. double a(void);
Funcția *a* nu are parametri. Ea returnează o valoare flotantă în dublă precizie.
3. void c(int x,long y[],double z);
Funcția *c* nu returnează nici o valoare. Are trei parametri:
 - primul este de tip *int*;
 - al doilea este un tablou unidimensional de tip *long*;
 - al treilea este de tip *double*.
4. void c(int ,long[],double);
Acest prototip exprimă același lucru cu cel precedent.

Compilatorul utilizează datele din prototip pentru a verifica tipurile parametrilor de la apel (parametri efectivi). În cazul în care un parametru efectiv are un tip diferit de tipul corespunzător din prototip, compilatorul C convertește automat valoarea parametrului efectiv spre tipul indicat în prototip.

Utilizatorii limbajelor C și C++ pot folosi o serie de funcții aflate în bibliotecile standard ale acestor limbaje. Apelul unei funcții de bibliotecă implică și el prezența prealabilă a prototipurilor funcțiilor respective în textul sursă. Pentru a simplifica inserarea în textul sursă a prototipurilor funcțiilor de bibliotecă, s-au construit fișiere cu astfel de prototipuri. Ele au extensia .h (header). Un astfel de fișier conține prototipuri pentru funcții de bibliotecă "înrudite". De exemplu, fișierul *stdio.h* conține prototipuri pentru funcțiile de bibliotecă utilizate frecvent în operații de intrare/ieșire, fișierul *string.h* conține prototipurile pentru funcțiile utilizate la prelucrarea sirurilor de caractere etc.

Mentionăm că mediile integrate de dezvoltare Turbo C și C++ permit utilizatorului să găsească prototipurile funcțiilor de bibliotecă. În acest scop se procedează astfel:

- se tastează numele funcției pentru care se dorește să se găsească informații;
- se fixează cursorul pe o literă arbitrară a numelui funcției;
- se tastează <CTRL>-F1.

Se obține prototipul funcției, precum și fișierul cu extensia .h care-l conține. De asemenea, se afișează și alte informații în legătură cu funcția respectivă, utile pentru apelurile ei.

1.12. Preprocesare

Un program sursă C sau C++ poate fi prelucrat înainte de a fi supus compilării. O astfel de prelucrare se numește *preprocesare*. Ea este realizată automat înaintea compilării. Preprocesarea constă, în principiu, în substituții. Preprocesarea asigură:

- incluzări de fișiere cu texte sursă;
- definiții și apeluri de macrouri;
- compilare condiționată.

Preprocesarea se realizează prin prelucrarea unor informații specifice care au ca prim caracter caracterul diez (#).

În paragraful de față vom aborda construcția `#include` și parțial construcția `#define` utilizată la substituiri de succesiuni de caractere. Ulterior se vor descrie și alte facilități oferite de preprocesare.

1.12.1. Incluzări de fișiere cu texte sursă

Un fișier cu text sursă poate fi inclus cu ajutorul construcției `#include`. Această construcție are unul din următoarele formate:

`#include "specificator_de_fisier"`

sau

`#include <specificator_de_fisier>`

unde:

specificator_de_fisier

- Depinde de sistemul de operare. El definește un fișier cu text sursă păstrat pe disc. În faza de preprocesare, textul fișierului respectiv se substituie construcției `#include`. În felul acesta textul fișierului respectiv ia parte la compilare împreună cu textul în care a fost inclus.

În cazul sistemului de operare DOS, specificatorul de fișier trebuie să fie un nume de fișier împreună cu extensia lui (.C pentru compilatorul C, .CPP pentru compilatorul C++, .H pentru fișiere de tip header (fișiere cu prototipuri etc.). De asemenea, în afară de numele și extensia fișierului, specificatorul de fișier poate conține și o "cale", dacă este necesar, pentru localizarea fișierului.

Diferența dintre cele două formate constă în modul de căutare al fișierului de inclus.

Formatul cu parantezele unghiulare `<...>` se utilizează la incluzarea *fișierelor standard*, cum sunt cele care conțin prototipuri pentru funcțiile de bibliotecă. Directoarele în care se caută aceste fișiere se definesc în prealabil cu ajutorul submeniuului *Directories* al meniului *Options* din mediul integrat de dezvoltare Turbo C sau Turbo C++. Ordinea de căutare în aceste directoare corespunde cu ordinea în care au fost definite aceste directoare. În cazul în care fișierul căutat nu este găsit în aceste directoare, se dă un mesaj de eroare.

În cazul în care se utilizează caracterele *ghilimele*, fișierul se caută în directorul curent sau conform "căii" dacă aceasta este prezentă.

Un fișier standard care se include frecvent este fișierul *stdio.h* care conține

prototipurile pentru o serie de funcții ce realizează operații de intrare/ieșire. El se include folosind construcția:

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

Exemplu:

1. `#include "fis1.cpp"`

Se include textul fișierului *fis1.cpp* aflat în directorul curent.

2. `#include "c:\tc\sursă\fis2.c"`

În acest exemplu este prezentă calea fișierului *fis2.c*. Se observă că pentru a descrie calea, caracterul backslash se dublează conform convenției de reprezentare a caracterului backslash într-un sir de caractere.

Un text inserat cu ajutorul construcției `#include` la rindul său poate conține construcții `#include` pentru alte fișiere.

Construcțiile `#include` se seriu, de obicei, la începutul fișierelor sursă, pentru ca textele inserate să fie valabile în tot fișierul sursă care se compilează.

1.12.2. Substituiri de succesiuni de caractere la preprocesare

Construcția `#define` se poate folosi la substituții de succesiuni de caractere. În acest scop se utilizează formatul:

`#define nume succesiune_de_caractere`

unde:

nume Este precedat și urmat de cel puțin un spațiu.

Folosind aceasta construcție, preprocesarea *substituie nume* cu *succesiune_de_caractere* poște tot în textul sursă care urmează construcției `#define` respective, exceptând cazul cind *nume* apare într-un sir de caractere sau într-un comentariu.

Numele definit printr-o construcție `#define` substituindu-se prin secvență de caractere corespunzătoare nu mai este prezent la compilare.

De obicei, un astfel de nume se scrie cu litere mari pentru a se poate evidenția faptul că el este definit printr-o construcție `#define`.

Succesiune_de_caractere începe cu primul caracter care nu este alb. Ea poate fi continuată pe mai multe linii terminând rindul care dorim să se continue cu backslash.

`#define` este folosită frecvent pentru a defini constante. De aceea, uneori un nume definit printr-o construcție `#define` se spune că este o *constantă simbolică*.

O construcție `#define` autorizează substituția pe care o definește din punctul în care ea este serială și pîna la sfîrșitul fișierului în care ea este serială sau pîna la întîlnirea unei construcții `#undef` care o anulează. Aceasta are formatul:

#undef nume

La intlnirea ei, se dezactivează substituirea lui nume cu succesiunea de caractere care i-a fost atașată în prealabil printr-o construcție `#define`.

Succesiunea de caractere dintr-o construcție `#define` poate conține nume care au fost definite în prealabil prin alte construcții `#define`.

Exemplu:

```
1. ...
#define A 100
/* A se substitue prin 100 incepind din acest punct al fisierului sursa */
...
#define FACT 20
#define DIMMAX (A*FACT)
...
char tab[DIMMAX];
...
double mat[A][FACT];
...
```

După procesare se obțin declarațiile:

```
...
char tab[(100*20)];
...
double mat[100][20];
...
```

`DIMMAX` s-a substituit prin $(100*20)$ și nu prin valoarea 2000 a produsului deoarece procesorul nu face calcule ei numai substituții. Din această cauza se recomanda ca expresiile utilizate în construcțiile `#define` să fie incluse în paranteze rotunde.

2.

```
#define A 123
#define B A+120
...
x=3*B; // se substitue prin x=3*123+120;
...
```
3.

```
#define A 123
#define B (A+120)
...
x=3*B; // se substitue prin x=3*(123+120);
...
```
4.

```
...
#define A 100
int x{A+1}; // declaratia devine int x{100+1};
...
#undef A
#define A 3.5
...
```

```
double y;
...
y=A; // instructiunca devine y=3.5;
...
```

În exemplele de mai sus `A` este o constantă simbolică. Ea este un nume atribuit unei constante, nume care însă nu mai este prezent în faza de compilare.

`B` definește o expresie constantă, care este evaluată de compilator la intlnirea ei.

Constantele simbolice se utilizează frecvent în locul constantelor obișnuite deoarece ele prezintă următoarele avantaje:

- Permite să se atruije nume sugestive unor constante. Este mult mai sugestiv să folosim constanta simbolică `PI` definită prin:
`#define PI 3.14159`
decit valoarea ei.
- Permite realizarea de prescurtări. De exemplu, dacă valoarea 3.14159 se folosește de mai multe ori în program, atunci este mai simplu să folosim numele `PI` atribuit ei.
- Permite înlocuirea simplă a unei constante printr-o altă. Dacă o constantă se folosește de mai multe ori într-un program și ulterior se constată că valoarea ei trebuie schimbată (de exemplu este o constantă flotantă căreia trebuie să-i schimbăm precizia), atunci este mult mai simplu să-i schimbăm valoarea o singură dată în construcția `#define`, decit să o căutăm peste tot în program și să-i schimbăm valoarea în mod corespunzător.

2. INTRĂRI/IEȘIRI STANDARD

Prin *intrări/ieșiri* înțelegem un set de operații care permit schimbul de date între un program și un periferic.

În general, operația de introducere a datelor de la un periferic se numește *citire*, iar cea de ieșire pe un periferic *scriere*.

Numim *terminal standard* terminalul de la care s-a lansat programul. De obicei, terminalurile standard sunt de tip *display*. În acest caz operația de scriere se mai numește și *afișare*.

Limbajul C nu dispune de instrucțiuni specifice pentru operațiile de intrare/ieșire. Ele pot fi realizate apelind *functii* construite special pentru acest scop. Astfel de funcții există în biblioteca limbajului C. Operațiile de intrare/ieșire realizate prin intermediul funcțiilor de bibliotecă le numim *operații de intrare/ieșire standard*.

Utilizatorul poate el însuși să construiască astfel de funcții, în cazul în care nu sunt utilizabile cele standard existente.

Functiile de bibliotecă, utilizate mai frecvent pentru realizarea operațiilor de intrare/ieșire folosind terminalul standard sunt:

- pentru intrări: *getch*, *getche*, *gets* și *scanf*;
- pentru ieșiri: *putch*, *puts* și *printf*.

La acestea mai adăugăm macrourile:

getchar - pentru intrări;

și

putchar - pentru ieșiri.

Aceste macroururi sunt definite în fișierul *stdio.h* și din această cauză utilizarea lor implică includerea acestui fișier.

2.1. Funcțiile *getch* și *getche*

Functiile *getch* și *getche* sunt dependente de implementare. Mai jos indicăm utilizările lor sub mediile de programare Turbo C și Turbo C++.

Ambele funcții permit citirea direct de la tastatură a unui caracter.

Functia *getch* citește de la tastatură *fără ecou*, deci caracterul tastat la display nu se afișează pe ecranul acestuia. Ea permite citirea caracterelor de la tastatură atât a celor corespunzătoare codului ASCII, cit și a celor corespunzătoare unor funcții speciale cum ar fi tastele F1, F2 etc. sau combinații de taste speciale.

La citirea unui caracter al codului ASCII, funcția returnează codul ASCII al caracterului respectiv.

În cazul în care se acționează o tastă care nu corespunde unui caracter ASCII, funcția *getch* se apelează de *două ori*: la primul apel funcția returnează valoarea *zero*, iar la cel de al doilea apel se returnează o valoare specifică tastei acționate.

Functia *getche* este analogă cu funcția *getch*, cu singura diferență că ea realizează citirea *cu ecou* a caracterului tastat. Aceasta înseamnă că se afișează automat pe ecranul terminalului caracterul tastat.

Ambele funcții *nu au parametri* și se pot apela ca operanzi în expresii.

La apelarea lor se vizualizează *fereastra utilizator* și se așteaptă tastarea unui caracter. Programul continuă după tastarea caracterului.

Un apel de formă:

getch();

se utilizează în cazul în care dorim să vizualizăm fereastra utilizator și să blocăm programul pentru a analiza conținutul curent al ecranului. Pentru a debloca programul se acționează o tastă corespunzătoare unui caracter al codului ASCII. Caracterul respectiv se citește fără ecou și apoi se continuă execuția programului.

Functiile *getch* și *getche* au prototipurile în fișierul *conio.h*, deci utilizarea lor implică includerea acestui fișier.

2.2. Funcția *putch*

Functia *putch* afișează un caracter pe ecranul terminalului standard. Ea are un parametru care determină imaginea afișată la terminal.

Functia *putch* poate fi apelată astfel:

putch(expresie);

Prin acest apel se afișează imaginea definită de valoarea parametrului *expresie*. Valoarea parametrului se interpretează ca fiind codul ASCII al caracterului care se afișează.

Dacă valoarea expresiei se află în intervalul [32,126], atunci se afișează un caracter imprimabil al codului ASCII. Dacă valoarea respectivă este în afara acestui interval, atunci se afișează diferite imagini care pot fi folosite în diverse scopuri, cum poate fi de exemplu trasarea de chenare.

Functia *putch* afișează caractere colorate în conformitate cu culoarea curentă setată în modul *text* de funcționare al ecranului.

La revenirea din funcția *putch* se returnează codul imaginii afișate (valoarea parametrului de la apel).

Prototipul funcției *putch* se află în fișierul *conio.h*.

Exerciții:

- 2.1 Să se scrie un program care citește un caracter imprimabil și-l afișează apoi pe ecran.

Caracterul respectiv se citește folosind apelul:

getch()

La revenirea din *getch()* se returnează codul ASCII al caracterului tastat.

Acum urmează să îl folosim la apelul funcției *putch* pentru a afișa caracterul tastat. Deci pentru a rezolva problema de față putem folosi construcția:

```
putch(getch());
```

În acest caz parametrul efectiv este o expresie formată dintr-un singur operand, operand care constă din apelul funcției *getch*. Aceasta este posibil deoarece orice funcție care returnează o valoare poate fi apelată ca operand al unei expresii.

Mentionăm că funcția *getch* returnează valoarea 13 (CR) la citirea caracterului rezultat din acționarea tastei *Enter*.

PROGRAMUL BII1

```
#include <conio.h>
main() /* citeste fara ecou un caracter imprimabil ASCII si-l afiseaza pe ecran */
{
    putch(getch());
}
```

Observații:

1. Programul este inefectiv dacă se acționează o tastă ce nu corespunde caracterelor ASCII.
2. Caracterul afișat prin *putch* se poate vedea în fereastra utilizator. Aceasta se vizualizează tastind:
<Alt>-F5
Se revine în fereastra utilizator acționând o tastă oarecare.
- 2.2 Să se scrie un program care citește fară ecou un caracter imprimabil ASCII, îl afișează la terminal și apoi trece cursorul pe linia următoare.

Programul de față este asemănător cu cel precedent. În plus se cere trecerea cursorului pe linia următoare. În acest scop se apelează funcția *putch* cu codul caracterului *newline*:

```
putch(10);
```

Mentionăm că putem înlocui valoarea 10 prin constanta caracter '\n':

```
putch ('\n');
```

PROGRAMUL BII2

```
#include <conio.h>
main() /* citeste fara ecou un caracter imprimabil ASCII, il afiseaza la terminal si apoi se trece cursorul pe linia urmatoare */
{
    putch(getch());
    putch('\n');
}
```

- 2.3 Programul BII3 realizează același lucru ca și programul BII2, cu deosebirea că înainte de a se termina execuția lui, se afișează fereastra utilizator și se așteaptă acționarea unei taste.

PROGRAMUL BII3

```
#include <conio.h>
main() /* citeste fara ecou un caracter imprimabil ASCII, il afiseaza, trece cursorul pe linia urmatoare, afiseaza fereastra utilizator si se asteapta actionarea unei taste; programul se termina dupa actionarea unei taste */
{
    putch(getch());
    putch('\n');
    getch();
}
```

2.3. Macrourile *getchar* și *putchar*

ACESTE MACROURI SINT DEFINITE IN FIȘIERUL *stdio.h*. ELE SE APELEAZĂ LA FEL CA FUNCȚIILE.

Macroul *getchar* permite citirea cu ecou a caracterelor de la terminalul standard. Se pot citi numai caractere ale codului ASCII, nu și caractere corespunzătoare tastelor speciale. Prin intermediu macroului *getchar* caracterele nu se citesc direct de la tastatură. Caracterele tastate la terminal se introduc într-o zonă tampon pînă la acționarea tastei *Enter*. În acest moment, în zona tampon, se introduce caracterul de rînd nou (*newline*) și se continuă execuția lui *getchar*. Se revine din *getchar* returnindu-se codul ASCII al caracterului curent din zona tampon. La un nou apel al lui *getchar* se revine cu codul ASCII al caracterului următor din zona tampon. La epuizarea tuturor caracterelor din zona tampon, apelul lui *getchar* implică tastarea la terminal a unui nou set de caractere care se reincarcă în zona tampon.

Un astfel de mod de desfășurare a operației de citire implică o anumită organizare a memoriei și accesului la caractere, organizare care conduce la noțiunea de *fișier*.

În general, prin fișier se înțelege o mulțime ordonată de elemente păstrate pe suporturi. Elementele unui fișier se numesc *inregistrări*. În mod frecvent fișierele sunt păstrate pe discuri. Cu toate acestea, este util să se considere organizate în fișiere chiar și datele care se tastează sau se afișează la terminal. În acest caz înregistrarea este un rînd afișat la terminal sau succesiunea de caractere tastată la terminal și terminată prin acționarea tastei *Enter*.

Fișierele conțin o înregistrare specială care marchează *sfîrșitul de fișier*. Această înregistrare se realizează la tastatură prin secvențe speciale. În cazul limbajelor Turbo C și Turbo C++, sfîrșitul de fișier se obține tastind:

<CTRL>-Z

al cărui ecou este: ^Z

Macroul `getchar` returnează valoarea constantei simbolice EOF (End Of File) la întâlnirea sfîrșitului de fișier. Această constantă este definită în fișierul `stdio.h`. Valoarea ei depinde de implementare. În cazul mediilor de programare Turbo C și Turbo C++, ea are valoarea -1.

Macroul `getchar` se apelează fără parametri și de obicei este un operand al unei expresii: `getchar()`.

Macroul `putchar` afișează un caracter al codului ASCII. El returnează codul caracterului afișat sau -1 la eroare.

Se poate apela prin formatul de mai jos:

`putchar(expresie);`

Valoarea expresiei reprezintă codul ASCII al caracterului care se afișează.

Observații:

1. Citirea caracterelor prin intermediul zonelor tampon are avantajul că permite corectarea erorilor la tastare. Astfel, operatorul de la terminal poate să modifice o secvență de caractere tastate, înainte de a acționa tastă Enter.
2. Macroul `getchar` returnează valoarea 10 (newline) la citirea caracterului corespunzător acționării tastei Enter.
3. Apelul
`putchar(10);`
sau echivalentul său
`putchar('\n');`
are ca efect trecerea cursorului în coloana unu de pe linia următoare.

Exerciții:

- 2.4 Să se scrie un program care citește un caracter folosind macroul `getchar`, îl afișează folosind macroul `putchar`, trece cursorul în coloana unu din linia următoare și apoi afișează fereastra utilizator pînă la acționarea unei taste.

Acest exercițiu este analog cu 2.3. În acest caz se cere schimbarea funcțiilor `getch` și `putch` cu macrourile `getchar` și respectiv `putchar`.

PROGRAMUL BII4

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

main() /* - citește un caracter folosind getchar;
           - afișaza caracterul folosind putchar;
           - trece cursorul la inceputul rîndului urmator;
           - blochează programul pînă la tastarea unui caracter */
{
    putchar(getchar());
    putchar('\n');
    getch();
}
```

Observație:

Deoarece se folosește macroul `getchar`, operatorul va tasta un caracter ASCII care trebuie să fie urmat de acționarea tastei Enter. Fără acționarea tastei Enter, programul ramîne în așteptare. După acționarea tastei Enter se va reveni din `getchar` cu codul ASCII al primului caracter tastat.

Dacă prima tastă acționată este chiar tastă Enter, atunci programul continuă execuția în mod automat și se revine din `getchar` cu valoarea 10 (valoarea corespunzătoare tastei Enter).

2.4. Funcțiile `gets` și `puts`

Funcția `gets` poate fi folosită pentru a introduce de la terminalul standard o succesiune de caractere terminată prin acționarea tastei Enter.

Citirea se face cu *ecou*. Se pot citi numai caractere ale codului ASCII.

Funcția are ca parametru *adresa de inceput* a zonei de memorie în care se păstrează caracterele citite. De obicei, această zonă de memorie este alocată unui tablou unidimensional de tip `char`.

Deoarece numele unui tablou are ca valoare adresa de inceput a zonei de memorie alocate, rezulta că numele unui tablou poate fi utilizat ca parametru în apelul funcției `gets`. În felul acesta, caracterele citite se vor păstra în tabloul respectiv.

Funcția `gets` returnează adresa de inceput a zonei de memorie în care s-au păstrat caracterele.

La întâlnirea sfîrșitului de fișier (<CTRL>-Z) se returnează valoarea zero. Zero nu reprezintă o adresa posibilă pentru `gets` și de aceea, ea poate fi folosită pentru a semnala sfîrșitul de fișier. De obicei, valoarea returnată de `gets` nu se testează față de zero, ci față de constanta simbolica `NUL`, definită în fișierul `stdio.h`.

Caracterul `newline`, care termină rîndul, nu se păstrează în memorie. În locul lui se păstrează caracterul `NUL` (\0). În felul acesta, caracterele citite prin `gets` formează un șir de caractere în conformitate cu cerințele limbajelor C și C++ (secvența de caractere se termină prin caracterul `NUL`).

Din cele de mai sus rezulta că daca `tab` este declarat prin:

```
char tab[255];
```

atunci apelul:

```
gets(tab);
```

păstrează în `tab` succesiunea de caractere tastată la terminalul standard în linia curentă. Totodată, `newline` se înlocuiește cu `NUL`.

Funcția `puts` afișează la terminalul standard un șir de caractere ale codului ASCII. Cursorul, după afișarea șirului respectiv, se trece automat în coloana intii

de pe linia următoare (deci caracterul *NUL* se înlocuiește cu newline).

Funcția are ca parametru adresa de început a zonei de memorie care conține caracterele de afișat.

În cazul în care sirul de caractere care se afișează se păstrează într-un tablou unidimensional de tip *char*, drept parametru se poate folosi numele acestui tablou.

Funcția *puts* returnează codul ultimului caracter al sirului de caractere afișat (caracterul care precede pe *NUL*) sau -1 la eroare.

Dacă *tab* are declarația de mai sus și el păstrează un sir de caractere, atunci apelul

```
puts(tab);
```

afișează la terminalul standard sirul respectiv de caractere și apoi trece cursorul în coloana intii de pe rindul următor.

Funcțiile *gets* și *puts* au prototipurile în fișierul stdio.h.

Exerciții:

- 2.5 Sa se scrie un program care citește de la intrarea standard numele și prenumele unei persoane, afișează inițialele persoanei respective pe un rind, fiecare inițială este urmată de un punct.

Numele și prenumele se tastează pe două rânduri diferite și apoi se vor afișa inițialele respective. În final se trece cursorul în coloana intii a liniei următoare și se afișează ferestra utilizator.

PROGRAMUL BII5

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

main() /*- citește numele și prenumele unei persoane tastate pe două linii separate;
         - afișează pe un rind inițialele urmate de punct;
         - afișează ferestra utilizator.*/
{
    char nume[255];
    char prenume[255];

    gets(nume); // citește prima linie tastată la terminal
    gets(prenume); // citește linia a două tastată la terminal
    putchar(nume[0]); // afișează prima inițială
    putchar('.'); // afișează punct după prima inițială
    putchar(prenume[0]); // afișează a două inițială
    putchar('.'); // punct după a două inițială
    putchar('\n'); // trece în coloana intii din linia următoare
    getch(); // se afișează ferestra utilizator; pentru a termina programul
             // se actionează o tasta oarecare
}
```

- 2.6 Să se modifice programul precedent astfel încât după afișarea inițialelor să se afișeze textul:

Pentru a termina actionati o tasta

Acest text se poate afișa cu ajutorul funcției *puts*, pe care o apelăm înainte de apelul lui *getch*. Se poate utiliza următorul apel al funcției *puts*:

```
puts("Pentru a termina actionati o tasta");
```

În acest apel parametrul efectiv este o expresie redusă la un operand care este o constantă sir.

La acest apel compilatorul păstrează caracterele constantei sir într-o zonă de memorie și apoi realizează apelul funcției *puts* folosind adresa de început a zonei respective.

PROGRAMUL BII6

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

main() /* - citește numele și prenumele unei persoane tastate pe două linii separate;
         - afișează inițialele persoanei urmate de puncte pe un același rind;
         - pe rindul următor se afișează textul:
               Pentru a termina programul actionati o tasta;
               se afișează ferestra utilizator. */
{
    char nume[255];
    char prenume[255];

    gets(nume);
    gets(prenume);
    putchar(nume[0]);
    putchar('.');
    putchar(prenume[0]);
    putchar('.');
    puts("\nPentru a termina programul actionati\
          o tasta");
    getch();
}
```

2.5. Funcția printf

Funcția *printf* poate fi folosită pentru a afișa date pe ecranul terminalului standard sub controlul unor formate. Ea poate fi apelată printr-o construcție de forma:

```
printf(control,par1,par2,...,parN);
```

unde:

control

- Este un sir de caractere care definește textele și formatele datelor care se scriu;

par1,par2,...,parn - Sint expresii. Valorile lor se scriu conform specificatorilor de format prezenți în parametrul de control.

Datele gestionate prin *printf* sint supuse unor transformări. Aceasta din cauză că datele au un format extern și unul intern.

Parametrul *control* al funcției *printf* definește aceste transformări ale datelor care se afișează pe ecran, transformări care se mai numesc *conversii*. El conține așa numiții *specificatori de format* care definesc conversiile datelor din format intern în format extern. În afara acestor specificatori de format, parametrul *control* al funcției *printf* poate conține succesiuni de caractere care se afișează ca atare în poziții corespunzătoare.

În concluzie, parametrul *control* are în compunerca sa:

- succesiuni de caractere care se afișează ca atare;
- specificatori de format care definesc conversiile valorilor parametrilor *par1, par2, ..., parn* din format intern în format extern.

Menționăm că aceste construcții nu sunt totdeauna ambele prezente. De exemplu, dacă dorim să se afișeze un text, atunci parametrul *control* nu conține specificatori de format, decât numai textul respectiv, iar parametrii *par1, par2, ..., parn* sunt absenți. În mod analog, parametrul *control* poate să conțină numai specificatori de format.

În mod frecvent, parametrul *control* conține atât specificatori de format, cit și alte caractere.

Exemplu:

Apelul funcției *puts* folosit în exercițiul 2.6.:

```
puts("\nPentru a termina programul actionati o tasta");
poate fi înlocuit printr-un apel al funcției printf care conține numai parametrul de control fără specificatori de format:
printf("\nPentru a termina programul actionati o tasta\n");
```

În cazul apelului funcției *printf* s-a folosit caracterul *newline* de la sfârșitul șirului de caractere pentru ca după afișarea textului, cursorul să treacă în coloana unu din linia următoare. Acest fapt se realizează automat în cazul funcției *puts* și deci la apelul ei șirul de caractere nu se mai termină cu *newline*.

Specificatorii de format, dacă există, se corespund cu parametrii *par1, par2, ..., parn*. Astfel, al *k*-lea specificator de format controlează afișarea valorii parametrului *park*.

Formatul extern al unei date constă dintr-o succesiune de caractere imprimabile. Ele se afișează într-o zonă numită *cimp*.

Un specificator de format începe cu un caracter procent (%). În continuare, mai pot exista caracterele indicate mai jos:

un caracter minus optional - Implicit datele se cadrează în dreapta cîmpului în care ele se scriu. Dacă este prezent caracterul minus, atunci data corespunzătoare lui este cadrată în stînga.

un sir de cifre zecimale optional

- Care definește dimensiunea minima a cîmpului în care se afișează caracterele care intră în compunerea formatului extern al datei. În cazul în care data necesită un cîmp de dimensiune mai mare decât cel precizat, ea se va scrie pe atitea caractere cîte și sunt necesare. În cazul în care data necesită un cîmp mai mic decât cel definit în specificatorul de format, ea se va scrie în cîmpul respectiv în dreapta sau în stînga, dacă este prezent caracterul minus în specificatorul de format corespunzător ei. De asemenea, în acest caz cîmpul se completează cu caractere nesemnificative; implicit, caracterele nesemnificative sunt spații. Caracterele nesemnificative vor fi zeroi dacă numărul ce indică dimensiunea minimă a cîmpului începe cu un zero nesemnificativ (aici zeroi nesemnificativ nu înseamnă că numărul este în octal, deoarece acest sir de cifre nu reprezintă o constantă întreagă, ci are semnificația amintită mai sus).

un punct optional, urmat de un sir de cifre zecimale

- Sirul de cifre zecimale aflate după punct definește precizia datei care se scrie sub controlul specificatorului respectiv. Dacă punctul este prezent, atunci și precizia este prezentă. În cazul în care data care se scrie este flotantă, precizia definește numărul de zecimale care se scriu. În cazul în care data este un sir de caractere, precizia indică numărul de caractere care se scriu.

una sau două litere

- Care definește tipul de conversie aplicat datei care se scrie.

Din cele de mai sus, rezultă că un specificator de format începe cu un caracter procent și se termină cu o literă.

Între caracterul procent și litera care sfîrșește un specificator de format mai pot apărea și alte caractere a căror interpretare s-a dat mai sus.

În cazul în care după caracterul procent urmează un caracter diferit de cele folosite la definirea specificatorilor de format, caracterul procent respectiv se ignoră și se afișează caracterul care urmează după el. Această regulă permite afișarea insuși a caracterului procent. Astfel, succesiunea:

%%

nu reprezintă un specificator de format, primul caracter procent se ignora, iar cel de al doilea caracter procent se va afișa la terminal.

Funcția *printf* returnează numărul de octeți (caractere) care se afișează la terminal sau -1 la eroare.

Prototipul funcției *printf* se află în fișierul *stdio.h*.

În cele ce urmează vom indica principalele conversii realizate prin specificatorii de format. Ele sunt definite de literele de la sfârșitul specificatorului de format.

2.5.1. Litera c

Permite afişarea unui caracter. Valoarea parametrului corespunzător este interpretată ca fiind codul ASCII al unui caracter și caracterul respectiv se afișează în cimpul definit de specifiant.

2.5.2. Litera s

Permite afişarea unui sir de caractere.

Parametrul corespunzător are ca valoare adresa de început a zonei de memorie care conține caracterele sirului de afișat. Se afișează toate caracterele sirului pînă la întîlnirea caracterului NUL.

Exemple:

1. Secvența de apeluri:

```
putchar(getchar());
```

se poate realiza folosind funcția *printf* astfel:

```
printf("%c",getchar());
```

2. Secvența de apeluri:

```
putchar(getchar());  
putchar('\n');
```

se poate înlocui prin:

```
printf("%c\n",getchar());
```

3. Apelul lui *printf* pentru a afișa un sir de caractere:

```
printf("abc");
```

se poate scrie folosind specifiantul de format pentru siruri de caractere:

```
printf("%s", "abc");
```

În ambele cazuri, compilatorul pastrează sirul de caractere "abc" într-o zonă specială prevăzută pentru acest fapt și apelează funcția *printf* folosind ca parametru adresa de început a acestei zone de memorie.

4. Fie apelul lui *printf*:

```
printf("*%4c*",getchar());
```

Caracterul citit prin *getchar* se afișează într-un cimp de 4 caractere, cadrat în dreapta. De exemplu, dacă s-a citit caracterul A, atunci apelul de mai sus va afișa:

```
* A*
```

5. Apelul:

```
printf("*%-4c*",getchar());
```

va afișa caracterul citit într-un cimp de 4 caractere, cadrat în stînga. De exemplu, dacă se citește caracterul A, atunci apelul de mai sus va afișa:

```
*A *
```

6. Fie apelul:

```
printf("*%10s*", "abc");
```

se afișează:

```
* abc*
```

7. Apelul:

```
printf("*%-10s*", "abc");
```

afișează:

```
*abc *
```

8. Apelul:

```
printf("*%15.10*", "limbajul C++");
```

afișează:

```
* limbajul C*
```

2.5.3. Litera d

Datele de tip *int* pot fi convertite și afișate în zecimal folosind un specifiant de format terminat cu litera *d*. Valorile negative se afișează precedate de semnul minus.

Exemple:

1. Specifiantul

```
%d
```

se folosește pentru a afișa, în zecimal, o dată de tip *int*.

2. Apelul:

```
printf("*%10d*",123);
```

afișează:

```
* 123*
```

3. Apelul:

```
printf("*%-10d*",123);
```

afișează:

```
*123 *
```

4. Apelul:

```
printf("*%010d*",123);
```

afișează:

```
*0000000123*
```

2.5.4. Litera o

Datele de tip *int* sau *unsigned* pot fi convertite și afișate în octal folosind un specifiant de format terminat cu litera *o*.

Fiecare grupă de trei cifre binare se înlocuiește printr-o cifră octală. Gruparea

cifrelor se face de la dreapta spre stanga, adică de la ordinea inferioare spre cele superioare.

Exemplu:

Apelul:
`printf("%%10o", 123);`
afișează:
* 173*

2.5.5. Literele x și X

Datele de tip *int* sau *unsigned* pot fi convertite și afișate în hexazecimal folosind un specificator de format terminat cu litera *x* sau *X*. În acest scop, patru cifre binare se înlocuiesc printr-o cifră hexazecimală.

Exemplu:

Apelul:
`printf("%%10x", 123);`
afișează:
* 7b*

În cazul în care se folosește litera mare *X*, cifrele hexa peste 9 se afișează cu litere mari.

2.5.6. Litera u

Litera *u* este asemănătoare cu litera *d*. În acest caz se realizează conversia unei date binare de tip *unsigned* în decimal.

2.5.7. Litera l

Litera *l* poate precede pe una din literele *d*, *o*, *x*, *X* sau *u*.

În acest caz se fac conversii din tipul *long* sau *long unsigned*. Astfel, specificatorul terminat în *ld* realizează conversia din *long* în decimal. Specificatorul *lu* realizează conversia din *long unsigned* în decimal. Specificatorul *lo* face conversie din *long* sau *long unsigned* în octal. Specificatorul *lx* sau *lX* face conversia din *long* sau *long unsigned* în hexazecimal.

2.5.8. Litera f

Litera *f* permite conversia datelor de tip *float* sau *double* spre formatele:

parte_intreagă.*parte_fracționară*
sau
parte_intreagă

Partea întreagă este un sir de cifre zecimale care este precedat de semnul minus dacă numărul este negativ.

Numărul zecimalelor se definește de precizia indicată în specificatorul de format. Dacă ea este absentă, atunci se afișează 6 zecimale. Ultima cifră afișată este rotunjită. Rotunjirea se face prin adăus dacă prima cifră neglijată este cel puțin egală cu 5 și prin lipsă în caz contrar.

Exemplu:

Valoarea datei	Specificator	Afișare
3.14159265	%5f	3.141593
123.672	%7f	123.672000
3.14159265	%7.2f	3.14
123.672	%10.1f	123.7
-123.672	%10.1f	-123.7
3.14159265	%10.0f	3
123.672	%10.0f	124

2.5.9. Literele e și E

Acstea permit conversia datelor flotante de tip *float* sau *double* spre formatele:

parte_intreagă.*parte_fracționară exponent*

sau

parte_intreagă exponent

Partea întreagă este o cifră zecimală care este precedată de semnul minus dacă numărul este negativ.

Numărul zecimalelor este cel definit de precizia dacă aceasta este prezentă în specificatorul de format. În cazul în care precizia este absentă, numărul zecimalelor afișate este egal cu 6. Ultima cifră afișată este rotunjită conform regulei indicate mai sus la descrierea literei *f*.

Exponentul începe cu litera *e* dacă specificatorul de format se termină cu *e* și cu *E* dacă el se termină cu *E*. Urmează un semn plus sau minus dacă acesta este negativ. După semn se află un întreg zecimal de cel puțin 2 cifre.

Exponentul definește o putere a lui zece care inmulțește restul reprezentării numărului pentru a obține valoarea reală a acestuia.

Exemplu:

Valoarea datei	Specificator	Afișare
3.14159265	%e	3.141593e+00
123.672	%c	1.236720e+02
123.672	%.1E	1.2E+02
0.673	%.0f	6.730000E-01
123.672	%.0f	1E+02

2.5.10. Literele g și G

Specificatorul de format terminat cu litera *g* sau *G* afișează data respectivă fie ca în cazul specificatorului terminat cu *f*, fie ca în cazul specificatorului terminat cu *e* sau *E*. Formatul de afișare cu exponent sau fără, se alege automat în așa fel încit afișarea să ocupe un număr minim de caractere. De asemenea, specificatorii %*g* și %*G* afișează 6 zecimale numai dacă acestea sunt toate semnificative.

Se folosește litera *e* la scrierea exponentului dacă specificatorul se termină cu *g* și *E* pentru specificatorul terminat cu *G*.

Observație:

Datele de tip *float* au o precizie de 6-7 zecimale. De aceea, pentru datele de acest tip nu se recomandă să se afișeze un număr mai mare de zecimale.

Datele de tip *double* au o precizie de pînă la 15 zecimale și deci, în acest caz, putem afișa pînă la 15-16 zecimale. Afișarea unui număr mai mare de zecimale nu are sens.

2.5.11. Litera L

Litera *L* poate precede una din literele *f*, *e*, *E*, *g* și *G*. În acest caz data care se afișază este de tip *long double*. Convențiile de afișare sunt aceleași, adică datele afișate cu un specificator de format terminat cu *Lf* sunt fără exponent, cele cu un specificator de format terminat cu *Le* sau *LE* se afișează cu exponent, iar cele afișate cu un specificator de format cu *Lg* sau *LG* se afișează cu unul din formatele precedente care asigură o afișare cu un număr minim de caractere.

Exerciții:

- 2.7 Să se scrie un program care afișează pe o linie, între două caractere asterisc, caracterul citit de la terminal prin *getchar*, apoi trece cursorul în coloana intîi de pe linia următoare și afișează ecranul utilizator.

PROGRAMUL BI17

```
#include <stdio.h>
#include <conio.h>
main() /*- citeste un caracter cu getchar;
         - afiseaza pe o linie caracterul citit, intre doua caractere asterisc;
         - trece cursorul in coloana intii din linia urmatoare;
         - afiseaza fereastra utilizator. */
{
    printf("*%c*\n", getchar());
    puts(" Actionati o tasta pentru a continua");
    getch();
}
```

- 2.8 Să se modifice programul din exemplul precedent în așa fel încit caracterul

citit prin *getchar* să se afișeze între două caractere procent.

PROGRAMUL BI18

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

main() /* - citeste un caracter cu getchar;
         - afiseaza pe o linie caracterul citit, intre doua caractere procent;
         - trece cursorul in coloana intii din linia urmatoare;
         - afiseaza fereastra utilizator. */
{
    printf("%%c%%\n", getchar());
    puts(" Actionati o tasta pentru a continua");
    getch();
}
```

- 2.9 Programul de mai jos afișează textul "Limbajul C++", folosind următorii specificatori de format:

%s, %15s, %-15s, %10s, %15.10s și %-15.10s

Pentru a pune în evidență cimpul afișat prin specificatorii respectivi, vom include de fiecare dată cimpul între caractere asterisc.

PROGRAMUL BI19

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

#define V "Limbajul C++"

main() /* afiseaza, intre caractere asterisc, textul Limbajul C++
         folosind diferiti specificatori de format */

{
    printf("**%s*\n", V);
    printf("**%15s*\n", V);
    printf("**%-15s*\n", V);
    printf("**%10s*\n", V);
    printf("**%15.10s*\n", V);
    printf("**%-15.10s*\n", V);
    puts (" Actionati o tasta pentru a continua");
    getch();
}
```

Rezultatele execuției programului BI19 sunt:

```
*Limbajul C++*
* Limbajul C+++
*Limbajul C++  *
*Limbajul C+++
*   Limbajul C*
*Limbajul C   *
Actionati o tasta pentru a continua
```

- 2.10 Să se scrie un program care citește un caracter cu *getchar* și afișează codul său ASCII.

PROGRAMUL BII10

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

main() /* citește un caracter cu getchar si afiseaza codul sau ASCII */
{
    printf("%d\n", getchar());
    puts("Actionati o tasta pentru a continua");
    getch();
}
```

- 2.11 Să se scrie un program care citește un caracter care nu corespunde codului ASCII și afișează codul său returnat prin funcția *getch*.

Amintim că pentru a citi caractere care nu corespund codului ASCII, se apelează *getch* de două ori. La primul apel, *getch* returnează valoarea zero. La cel de al doilea apel se returnează un cod specific caracterului citit.

PROGRAMUL BII11

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

main()/* citește un caracter care nu aparține codului ASCII
       și afiseaza codul caracterului citit */
{
    printf("%d\n", getch()); //afiseaza 0
    printf("%d\n", getch()); //afiseaza codul caracterului
    puts("Actionati o tasta pentru a continua");
    getch();
}
```

- 2.12 Să se scrie un program care afișează constanta 12345 în zecimal, octal și hexazecimal.

PROGRAMUL BII12

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

#define V 12345

main() /* afiseaza intregul 12345 in zecimal, octal si hexazecimal */
{
    printf("zecimal %d\n", V);
    printf("octal %o\n", V);
    printf("hexazecimal %x\n", V);
    puts("Actionati o tasta pentru a continua");
    getch();
```

}

- 2.13 Să se scrie un program care afișează constanta 123456789 în zecimal, octal și hexazecimal.

PROGRAMUL BII13

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

#define V 123456789

main() /* afiseaza constanta 123456789 in zecimal, octal si hexazecimal */
{
    printf("zecimal %ld\n", V);
    printf("octal %lo\n", V);
    printf("hexazecimal %lx\n", V);
    puts("Actionati o tasta pentru a continua");
    getch();
}
```

- 2.14 Programul BII14 afișează constanta 123 cu diferiți specifiicatori de format.

PROGRAMUL BII14

```
#include <stdio.h>
#include <conio.h>
#define V 123

main() /* afiseaza constanta 123 cu diferiti specifiicatori de format */
{
    printf(">%d*\n", V);
    printf("*%7d*\n", V);
    printf("*%-7d*\n", V);
    printf("%07d*\n", V);
    puts("Actionati o tasta pentru a continua");
    getch();
}
```

Rezultatele execuției programului sunt:

```
*123*
*   123*
*123   *
*0000123*
Actionati o tasta pentru a continua
```

- 2.15 Programul BII15 afișează numere neintregi folosind diferiți specifiicatori de format.

PROGRAMUL BII15

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

```

#define A 47.389
#define X -123.5e20

main() /* afiseaza numere neintregi cu diferiti specifatori de format*/
{
    printf("A=%f\n", A);
    printf("A=%.3f\n", A);
    printf("A=%.2f\n", A);
    printf("A=%1f\n", A);
    printf("A=%0f\n", A);
    printf("X=%e\n", X);
    printf("X=%.4e\n", X);
    printf("X=%.3e\n", X);
    printf("X=%.2e\n", X);
    printf("X=%1e\n", X);
    printf("A=%g X=%G\n", A, X);
    printf("%101\n%20e\n%20g\n", 1.25, 1.25, 1.25);
    printf("%20f\n%20e\n%20g\n", .25, .25, .25);
    printf("%20f\n%20e\n%20g\n", 123e-1, 123e-1, 123e-1);
    printf("%20f\n%20e\n%20g\n", 12.3e4, 12.3e4, 12.3e4);
    printf("%30.20Le\n", 356.782143657891213891);
    printf("%40.30Le\n", 31415926535897932384526434e-251);
}

```

Rezultatele execuției programului sunt:

```

A=47.389000
A=47.389
A=47.39
A=47.4
A=47
X=-1.235000e+22
X=-1.2350e+22
X=-1.235e+22
X=-1.24e+22
X=-1.2e+22
A=47.389 X=-1.235E+22
    1.250000
    250000e+00
        1.25
        0.250000
        .500000e-01
            0.25
            12.300000
            1.230000e+01
                12.3
                123000.000000
                1.230000e+05
                    123000
                    3.56782143657891214000e+02
                    3.141592653589793240000000000000e+00

```

2.6. Funcția scanf

Funcția *scanf* poate fi folosită pentru a introduce date tastate la terminalul standard sub controlul unor formate. Ea poate fi apelată printr-o construcție de forma:

scanf(control,par1,par2,...,parn);

unde:

control

- Este un sir de caractere care definește formatele datelor și a eventualelor texte aflate la intrarea de la tastatură.

par1,par2,...,parn - Sunt adresele zonelor în care se păstrează datele citite după ce au fost convertite din formatele lor externe în formate interne corespunzătoare.

Caracterele albe din compunerea parametrului *control* sunt neglijate. Restul caracterelor, care nu fac parte dintr-un specificator de format, trebuie să existe la intrare în poziții corespunzătoare. Acestea se folosesc în vederea efectuării de controale asupra datelor citite.

De exemplu, dacă o dată care se citește se atribuie variabilei x, atunci data respectivă poate fi precedată de textul:

x=

În acest caz parametrul trebuie să conțină în poziția corespunzătoare:

x=specificatorul de format al datei care se citește

Textele de la intrare pot fi precedute de caractere albe care se neglijiază.

Specificatorii de format controlează introducerea datelor care au diferite formate externe reprezentate prin anumite succesiuni de caractere. De asemenea, specificatorii de format definesc conversiile din formate externe (ale datelor) în cele interne.

Specificatorii de format sunt asemănători cu cei întlniți la funcția *printf*. El începe cu un caracter procent și se termină cu 1-2 litere. Literele definesc tipul conversiei.

Între procent și litere, într-un specificator de format mai putem utiliza, în ordine:

- un caracter asterisc optional;
- un sir de cifre optional, care definește lungimea maximă a cimpului din care se citește data sub controlul formatului respectiv.

Cimpul controlat de un specificator de format începe cu *primul* caracter curent care nu este alb și se termină:

- fie la caracterul după care urmează un caracter alb;
- fie la caracterul după care urmează un caracter care nu corespunde specificatorului de format care controlează acel cimp;

- c. fie la caracterul prin care se ajunge la lungimea maximă a cimpului, indicată în specificatorul de format.

Condiția c. este absentă dacă în specificatorul de format nu este indicată lungimea maximă a cimpului.

Menționăm că această regulă nu se aplică la citirea de caractere.

În cazul în care specificatorul de format conține caracterul asterisc, data din cimpul respectiv va fi prezentă la intrare dar ea nu se atribuie nici unei variabile și deci nu-i va corespunde un parametru, spre deosebire de ceilalți specificatori care nu conțin caracterul asterisc.

Funcția `scanf` citește toate cimpurile care corespund specificatorilor de format, inclusiv eventualele texte prezente în parametrul *control*.

În cazul unei erori, citirea se intrerupe în locul în care s-a întlnit eroarea. Eroarea poate proveni:

- din necoresponțea textului curent din parametrul de control cu cel din fișierul de intrare;
- din neconcordanța dintre data din cimp și specificatorul de format sub controlul căruia se face citirea.

Apariția unei erori poate fi pusă ușor în evidență deoarece `scanf` returnează, la revenirea din ea, numărul cimpurilor citite corect. În felul acesta, valoarea returnată de `scanf` poate fi testată și se poate stabili cine cimpuri au fost citite la fiecare apel a lui `scanf`.

Funcția `scanf` citește date din zona tampon atașată tastaturii, la fel ca și `getchar`. De aceea, datele se citesc efectiv după ce s-a acționat tasta *Enter*.

La întlnirea sfîrșitului de fișier (`<CTRL>-Z` la mediile Turbo C și Turbo C++) se returnează valoarea *EOF*. Se recomandă ca sfîrșitul de fișier să fie precedat de un caracter alb pentru a nu pierde datele tastată înainte de sfîrșitul de fișier. De asemenea, după sfîrșitul de fișier se acționează tasta *Enter* pentru a termina înregistrarea curentă.

Funcția `scanf`, ca și funcția `printf` are prototipul în fișierul *stdio.h*.

Așa cum s-a arătat mai sus, parametrii *par1*, *par2*, ..., *parn* sunt adresele zonelor de memorie în care se păstrează datele citite de la tastatură în urma conversiilor în format intern. Adresa unei zone de memorie se exprimă adesea folosind *operatorul unar &*.

Așa de exemplu, dacă *i* este o variabilă simplă oarecare, atunci

`&i`

reprezintă adresa zonei de memorie alocată variabilei *i*.

În general, dacă *nume* este numele unei variabile simple, atunci adresa de început a zonei de memorie alocată variabilei *nume* se exprimă cu ajutorul expresiei:

`&nume`

Amintim că dacă *nume* este numele unui tablou, atunci nu vom mai folosi operatorul *&*, deoarece în acest caz *nume* are ca valoare chiar adresa de început a

zonei de memorie alocate tabloului respectiv.

2.6.1. Litera c

Specificatorul de format

`%c`

se folosește pentru a citi, cu ajutorul lui `scanf`, caracterul curent din zona tampon corespunzătoare datelor introduse de la tastatură.

În acest caz nu se face avans pînă la primul caracter care nu este alb, ci pur și simplu se citește caracterul curent (unul singur). Codul său ASCII se păstrează în zona de memorie definită de parametrul efectiv corespunzător.

Exemplu:

Fie declarația:

`char car;`

Apelul:

`scanf ("%c", &car);`

citește caracterul curent din zona tampon și păstrează codul ASCII al acestuia în zona de memorie alocată variabilei *car*.

Apelind în continuare funcția `printf`:

`printf ("%c", car);`

se va afișa, la terminalul standard, caracterul citit prin apelul de mai sus a lui `scanf`.

2.6.2. Litera s

Specificatorii de format terminați prin *s* se folosesc pentru a citi șiruri de caractere.

La citirea cu un astfel de specificator de format, data se consideră ca facind parte dintr-un cimp care începe cu un caracter care nu este alb și care se termină fie la caracterul după care urmează un caracter alb, fie la caracterul prin care se ajunge la lungimea maximă de caracter indicată în specificatorul de format.

Caracterele citite se pot păstra într-o zonă de memorie organizată ca un tablou de caractere. În acest caz parametrul corespunzător specificatorului de format poate fi chiar numele acestui tablou.

Menționăm că după ultimul caracter citit sub controlul unui astfel de specificator de format se păstrează în mod automat caracterul NUL ('0').

Exemple:

1. Fie textul

Limbajul C++

tastat pe un același rînd la terminalul standard.

Dacă `tab1` și `tab2` sunt declarate ca mai jos:

```
char tab1[10];
char tab2[4];
```

atunci textul de mai sus poate fi citit folosind apelurile:

```
scanf ("%s", tab1); // se citește textul: Limbajul
                     // cimpul incepe cu litera L și se termină cu l
                     // deoarece după l urmează spațiu
```

```
scanf ("%s", tab2); // se citește textul: C++
                     // cimpul incepe cu litera C și se termină cu cel de al doilea
                     // caracter + deoarece după el urmează newline
```

Același lucru se poate realiza folosind un singur apel a lui `scanf` cu doi specifiicatori de format:

```
scanf ("%s %s", tab1, tab2);
```

Menționăm că spațiul dintre specifiicatorii de format nu este esențial, el se neglijiază. Deci același apel poate fi scris și sub forma

```
scanf ("%s%s", tab1, tab2);
```

Datele citite prin aceste apeluri se pastrează în memorie prin codurile lor ASCII astfel:

```
tab1[0] -> 'L', tab1[1] -> 'i', tab1[2] -> 'm',
tab1[3] -> 'b', tab1[4] -> 'a', tab1[5] -> 'j',
tab1[6] -> 'u', tab1[7] -> 'l', tab1[8] -> '\0',
tab2[0] -> 'C', tab2[1] -> '+', tab2[2] -> '+',
tab2[3] -> '\0'.
```

2. Fie declarațiile:

```
char tab1[10];
char tab2[10];
```

și apelul:

```
scanf ("%2s%9s", tab1, tab2);
```

Presupunem că la terminal s-a tastat cuvintul *necunoscut* precedat și urmat de cel puțin un caracter alb.

În acest caz funcția `scanf` realizează următoarele:

- Primul specifiicator de format asigură citirea cuvintului:

ne

Aceasta deoarece cimpul are lungimea maximă de 2 caractere. El incepe cu litera "n" (primul caracter care nu este alb) și se termină cu litera "e" la care s-a ajuns să se citească două caractere, cît s-a specificat în specifiicatorul de format;

- Al doilea specifiicator de format asigură citirea în continuare. În acest caz cimpul incepe cu litera "e" și se termină cu litera "l" deoarece după l urmează un caracter alb, iar numărul total al literelor cuvintului *cunoscut* este 8, care este mai mic decât lungimea maxima admisă: 9.

În concluzie, după apelul de mai sus, vom avea:

```
tab1[0] -> 'n'
tab1[1] -> 'e'
tab1[2] -> '\0'
tab2[0] -> 'c'
tab2[1] -> 'u'
...
tab2[7] -> 'l'
tab2[8] -> '\0'
```

3. Specifiicatorul

`%1s`

permete citirea unui singur caracter. O primă deosebire dintre acesta și specifiicatorul `%c` constă în aceea că, în timp ce `%1s` citește primul caracter care nu este alb, specifiicatorul `%c` citește caracterul curent, indiferent dacă acesta este alb sau nu. Deci cei doi specifiicatori citesc același lucru numai dacă, caracterul curent nu este alb.

O altă deosebire dintre cei doi specifiicatori constă în aceea că, specifiicatorul `%1s` implică memorarea caracterului *NUL* după caracterul citit, ceea ce nu are loc în cazul specifiicatorului `%c`. De aceea, se vede:

```
char car;
...
scanf ("%c", &car);
```

se modifică dacă folosim specifiicatorul `%1s`, astfel:

```
char car[2];
...
scanf ("%1s", car);
```

2.6.3. Litera d

Specifiicatorii terminați prin litera *d* permit citirea intregilor zecimali și conversia lor spre tipul *int*.

Numărul poate fi precedat de un semn. El trebuie să aparțină intervalului [-32768,32767]. În caz contrar, rezultatul va fi imprevizibil.

Cimpul din care se citește întregul se definește conform regulii generale indicate mai sus.

Exemple:

1. Fie declarația

```
int n;
și apelul
scanf ("%d", &n);
```

Dacă în cimpul curent din zona tampon de intrare se află întregul

123

precedat și urmat de cel puțin un caracter alb, atunci după acest apel se va păstra, în zona de memorie alocată lui *n*, valoarea în binar a întregului zecimal 123.

La revenire funcția *scanf* returnează valoarea 1 (un cimp citit corect).

2. Fie declarațiile

int i1,i2,i3;

și apelul:

scanf ("%2d %3d %2d",&i1,&i2,&i3);

Dacă în zona de intrare se află succesiunea de cifre

1234567

precedată și urmată de caractere albe, atunci după apel variabilele *i1*, *i2* și *i3* au respectiv valorile 12, 345 și 67.

Într-adevăr, primul specificator de format asigură citirea cifrelor dintr-un cimp de 2 cifre, al doilea specificator dintr-un cimp de 3 cifre, iar al treilea dintr-un cimp de 2 cifre. La revenire funcția *scanf* returnează valoarea 3 (3 cimpiuri citite corect).

3. Fie declarația

int n;

și apelul

scanf ("%d", &n);

Dacă, la intrare, construcția curentă este

i23

atunci construcția respectivă se consideră eronată, deoarece ea nu este un întreg zecimal. În acest caz *scanf* nu citește nimic și se revine din ea cu valoarea zero. Construcția respectivă va rămâne curentă, deci ea poate fi citită apelind în continuare funcția *scanf* cu un alt specificator de format (de exemplu *%s*) sau macroul *getchar*.

4. Considerăm declarația și apelul lui *scanf* din exemplul precedent.

Dacă la intrare se află construcția

23i

atunci se va citi întregul 23.

Cimpul începe cu cifra 2 și se sfîrșește cu cifra 3, deoarece după ea urmează o literă, adică un caracter care nu intră în compunerea întregilor zecimali.

La un nou apel al funcției *scanf* se va incerca citirea literei *i*.

După citirea întregului 23 se va reveni din *scanf* cu valoarea 1.

2.6.4. Litera o

Litera *o* se utilizează ca și litera *d*, deosebirea constă în aceea că în loc să se citească un întreg zecimal, se va citi un întreg *octal*. În acest caz fiecare cifră octală se înlocuiește cu trei cifre binare care definesc aceeași valoare ca și cifra respectivă.

2.6.5. Literele x sau X

Ambele litere se utilizează la fel ca și litera *d*, deosebirea constă în aceea că în loc să se citească un întreg zecimal, se va citi un întreg *hexazecimal*. În acest caz fiecare cifră hexazecimală se înlocuiește cu 4 cifre binare care definesc aceeași valoare ca și cifra respectivă.

Cifrele hexazecimale mai mari decât 9 pot fi reprezentate prin litere mici sau mari.

2.6.6. Litera u

Se utilizează pentru a citi întregi zecimali care se păstrează convertiți în binar și au tipul *unsigned*.

2.6.7. Litera f

Litera *f* permite citirea numerelor zecimale care pot fi reprezentate atât cu exponent, cât și fără exponent și păstrarea lor în format *flotant simplu precizie* deci date de tip *float*.

2.6.8. Litera l

Litera *l* poate precede pe oricare din literele *d*, *o*, *x*, *X*, *u* sau *f*.

Dacă *l* precede pe una din literele *d*, *o*, *x* sau *X*, atunci data citită se va converti spre tipul *long* în loc de *int*.

Specificatorul terminat cu *lu* realizează conversia spre *unsigned long*.

Specificatorul *lf* realizează conversia numărului citit în *flotantă dublă precizie*, deci spre tipul *double*.

2.6.9. Litera L

Litera *L* se folosește la scrierea specificatorilor terminați prin *Lf*. Ea permite conversia numerelor zecimale atât cu exponent, cât și fără exponent, în format *flotant de tip long double*.

Exerciții:

- 2.16 Sa se scrie un program care citește un caracter de la terminalul standard cu ajutorul funcției `scanf` și afișează codul ASCII al caracterului respectiv.

PROGRAMUL BII16

```
#include <stdio.h>
#include <conio.h>
main() /* - citește un caracter folosind funcția scanf;
           - afișează codul ASCII al caracterului respectiv. */
{
    char car;

    scanf("%c", &car);
    printf("%d\n", car);
    puts("Actionati o tasta pentru a continua");
    getch();
}
```

- 2.17 Sa se scrie un program care citește numele și prenumele unei persoane separate prin spații albe, apoi afișează prenumele pe un rind și numele pe rindul următor.

PROGRAMUL BII17

```
#include <stdio.h>
#include <conio.h>
main() /* - citește numele și prenumele unei persoane separate prin spații albe;
           - afișeaza prenumele pe un rind și numele pe rindul urmator */
{
    char nume[255];
    char prenume[255];

    scanf("%s %s", nume, prenume);
    puts(prenume);
    puts(nume);
    puts("Actionati o tasta pentru a continua");
    getch();
}
```

- 2.18 Sa se scrie un program care citește un întreg zecimal de cel mult 4 cifre și-l afișează în zecimal, octal și hexazecimal.

PROGRAMUL BII18

```
#include <stdio.h>
main() /* - citește un întreg zecimal de cel mult 4 cifre și-l afișează în zecimal,
           - octal și hexazecimal */
{
    int i;
    scanf("%d", &i);
    printf("%d\n", i);
```

```
    printf("%o\n", i);
    printf("%x\n", i);
}
```

Observație:

La execuția acestui program nu se mai afișeză fereastra utilizator cu rezultatele. Pentru vizualizarea ei se tastează

<Alt>-F5

Se revine în fereastra de editare acționind o tastă oarecare.

- 2.19 Să se scrie un program care citește un întreg zecimal de cel mult 9 cifre și-l afișează în zecimal, octal și hexazecimal.

PROGRAMUL BII19

```
#include <stdio.h>
main() /* citește un întreg zecimal de cel mult 9 cifre
           si-l afișează în zecimal, octal și hexazecimal */
{
    long i;

    scanf("%ld", &i);
    printf("%ld\n%lo\n%lx\n", i, i, i);
}
```

- 2.20 Să se scrie un program care:

– citește o dată calendaristică tastată sub forma:

zzllaa;

– afișează data respectivă sub forma:

19aa/ll/zz

unde:

zz

– Două cifre ce reprezintă ziua din lună: 01, 02,

ll

– Două cifre ce reprezintă numărul lunii: 01, 02, ..., 12.

aa

– Două cifre ce reprezintă ultimele două cifre ale unui an calendaristic care începe cu 19.

Programul are la intrare o succesiune de 6 cifre care pot fi precedate și urmate de spații albe. Ele pot fi citite cîte două folosind de trei ori specifikatorul de format %2d.

PROGRAMUL BII20

```
#include <stdio.h>
main() /* - citește o data calendaristica sub forma: zzllaa;
           - o afișează sub forma: 19aa/ll/zz. */
{
    int zi;
    int luna;
```

```

int an;

scanf("%2d %2d %2d", &zi,&luna,&an);
printf("19%02d/%02d/%02d\n",an,luna,zi);
}

```

Observație:

La afișare s-a folosit specificatorul de format:

%02d

Acesta afișează cifrele cu un zero nesemnificativ în față. Deci cifrele 1,2,...,9 se afișează ca 01,02,...,09.

- 2.21 Să se scrie un program care citește datele de mai jos și apoi afișează tipul operației citite pe un rind, iar celelalte date pe rîndul următor:

Tip operație	denumire	UM	cod	preț	cantitate
I	REZISTENTA	010KO	123456789	15	1000

PROGRAMUL BII21

```

#include <stdio.h>
#define MAXDEN 30
main() /* citește un rînd de date de la intrarea standard și le afiseaza astfel:
           - tip operație pe un rînd;
           - celelalte date pe rîndul urmator. */
{
    char den[MAXDEN+1];
    char tip[2];
    int val;
    char unit[3];
    long cod;
    float pret,cantitate;

    scanf("%1s%30s %3d %2s %ld %f %f",tip,den,&val,unit,&cod,
          &pret,&cantitate);
    printf("%s\n",tip);
    printf("%s %03d%s %ld %f %f\n",den,val,unit,cod,pret,
           cantitate);
}

```

Observație:

Tipul este un caracter:

- I (intrare);
- E (ieșire);
- T (transfer) etc.

Citirea lui se face folosind specificatorul %1s. Acesta permite eliminarea eventualelor caractere albe care ar putea precede litera ce definește tipul.

3. EXPRESII

O expresie, în limbajul C, este formată dintr-un *operand* sau mai mulți legați prin *operatori*.

3.1. Operand

Un operand poate fi:

- o constantă;
- o constantă simbolică;
- numele unei variabile simple;
- numele unui tablou;
- numele unei structuri;
- numele unui tip;
- numele unei funcții;
- referirea la elementul unui tablou (variabilă cu indici);
- referirea la elementul unei structuri;
- apelul unei funcții;
- expresie inclusă în paranteze rotunde.

Unui operand îi corespunde un *tip* și o *valoare*. Dacă tipul operandului este bine precizat la compilare, valoarea operandului se determină fie la compilare, fie la execuție.

Exemplu:

1. 1234
Este o constantă întreagă zecimală de tip *int*. Reprezintă un operand constant de tip *int*.
2. 0xa1b2
Este o constantă întreagă hexazecimală de tip *unsigned*. Reprezintă un operand constant de tip *unsigned*.
3. Fie declarația

$$\text{float alb2;} \\ \text{Numele} \\ \quad \text{alb2}$$

este numele unei variabile simple, deci reprezintă un operand. El are tipul *float*.
4. Fie declarația

$$\text{int tab[10];} \\ \text{tab[2]}$$

Este o referire la elementul al treilea al tabloului *tab*. Reprezintă un operand

de tip *int*.

Numele

tab

este numele unui tablou și deci el reprezintă un operand. El este de tip "adresa de intregi de tip *int*" (*pointer to int*), adică are ca valoare adresa unei zone de memorie care conține intregi de tip *int*. Se utilizează în mod frecvent ca parametru în apelul funcțiilor.

5. *sum(n, x)*

Este un apel al funcției *sum*. Reprezintă un operand al cărui tip coincide cu tipul valorii returnate de funcția *sum*. Funcția *sum* poate fi apelată ca operand dacă returnează o valoare.

6. *(expresie)*

Este un operand al cărui tip coincide cu tipul expresiei inclusă între paranteze.

3.2. Operatori

Operatorii pot fi *unari* sau *binari*.

Un operator unar se aplică unui singur operand.

Un operator binar se aplică la doi operanzi. Operatorul binar se aplică la operandul care îl precede imediat și la care îl urmează imediat.

Operatorii limbajului C nu pot avea ca operanzi constante și (șiruri de caractere). De asemenea, există limite în aplicarea operatorilor la anumiți operanzi.

Mai jos, se descriu operatorii limbajului C. Menționăm că limbajul C++ conține și alți operatori. Ei vor fi descriși ulterior, pe măsură ce se vor defini construcțiile specifice limbajului C++.

Operatorii limbajului C pot fi grupați în mai multe clase: operatori aritmetici, operatori de relație, operatori logici etc.

La scrierea unei expresii se pot folosi operatori din aceeași clasă sau din clase diferite. În principiu, se pot defini expresii complexe în care să se utilizeze operatori din toate clasele. La evaluarea unei astfel de expresii este necesar să se țină seama de *prioritățile* operatorilor care aparțin diferitelor clase de operatori, de *asociativitatea* operatorilor de aceeași prioritate și de *regula conversiilor implicite*.

3.2.1. Operatorii aritmetici

Operatorii aritmetici se utilizează la efectuarea calculelor cu date de diferite tipuri predefinite. Aceștia sunt:

- operatorii unari + și -;
- operatorii binari multiplicativi * și %;

c. operatorii binari aditivi + și -.

Operatorii unari sunt mai prioritari decât cei binari, iar operatorii multiplicativi sunt mai prioritari decât cei aditivi.

Operatorii unari au aceeași prioritate. Ei se asociază de la dreapta spre stînga.

Operatorii multiplicativi au aceeași prioritate. De asemenea, operatorii aditivi au aceeași prioritate.

Operatorii binari se asociază de la stînga la dreapta.

Operatorul unar + nu are nici un efect.

Operatorul unar - are ca efect negativarea valorii operandului pe care-l precede.

Operatorul * reprezintă operatorul de înmulțire al operanzilor la care se aplică.

Operatorul / reprezintă operatorul de impărțire a valorii operandului care îl precede la valoarea operandului care îl urmează.

În cazul în care ambele operanzi sunt întregi (*int*, *unsigned* sau *long*), se realizează o impărțire întreagă, adică rezultatul impărțirii este partea întreagă a cîntului.

Operatorul % se aplică numai la operanzi întregi și are ca rezultat restul impărțirii întregi a valorilor operanzilor săi.

Operatorii binari + și - reprezintă operațiile obișnuite de adunare și scădere definite în matematică.

Exemplu:

1. 1234 și +1234

sunt expresii care au aceeași valoare și tip.

Prima se reduce la operandul 1234 care este o constantă de tip *int*.

A doua se compune din operandul 1234 la care se aplică operatorul unar +. Cum acesta nu are nici un efect, cele două expresii au aceeași valoare și tip.

2. -1234

este o expresie care se compune din operandul 1234 la care se aplică operatorul unar de negativare.

3. 7/3

Deoarece operanzii sunt întregi, se realizează impărțirea întreagă a lui 7 la 3. Se obține valoarea 2 de tip *int*.

4. 7%3

Expresia are ca valoare restul impărțirii lui 7 la 3, deci $7 \% 3 = 1$.

5. -7%3

Expresia are ca valoare restul impărțirii lui -7 la 3, deci $-7 \% 3 = -1$. Într-adevăr,

$-7 / 3 = -2$ și $-7 = (-2) * 3 + (-1)$,
decă restul impărțirii este -1.

Exerciții:

- 3.1 Să se scrie un program care citește valoarea lui x , calculează valoarea polinomului $p(x)=3x^{**}2-8x+7$ și afișează rezultatul. Variabila x este de tip intreg.

Menționăm că aici și în continuare prin $a^{**}b$ notăm a la puterea b .

PROGRAMUL BIII1

```
#include <stdio.h>
main() /* - citeste valoarea lui x;
           - calculeaza valoarea polinomului 3x**2 - 8x + 7;
           - afiseaza rezultatul. */
{
    int x;

    printf("tastati valoarea lui x=");
    scanf("%d", &x);
    printf("x=%d\tp(x)=%d\n", x, 3*x*x - 8*x + 7 );
}
```

Observație:

Rezultatul calculului este incorrect dacă cel puțin un rezultat parțial depășește intervalul valorilor de tip int ($[-32768,32767]$).

În general, dacă operanții unui operator binar au același tip t , atunci rezultatul aplicării operatorului are tipul t . În cazul în care rezultatul este în afara intervalului de valori corespunzător tipului t , rezultatul aplicării operatorului este eronat.

Dacă un operator binar se aplică la operanți de tipuri diferite, se aplică regula conversiilor implicate (vezi mai jos).

- 3.2 Să se scrie un program care citește valoarea lui x , calculează valoarea polinomului $f(x)=3,5x^{**}3-9,8x+3,7$ și afișează rezultatul. Calculele se fac în flotantă dublă precizie.

PROGRAMUL BIII2

```
#include <stdio.h>
main() /* - citeste valoarea lui x;
           - calculeaza si afiseaza valoarea polinomului 3,5x**3 - 9,8x + 3,7.
           */
{
    double x;

    printf("tastati valoarea lui x = ");
    scanf("%lf", &x);
    printf("x=%g\ta(x)=%g\n", x, 3.5*x*x*x - 9.8*x + 3.7);
}
```

Observație:

Constantele flotante 3.5, 9.8 și 3.7 se consideră de tip *double*.

- 3.3 Să se scrie un program care citește valoarea lui x , calculează valoarea polinomului $a(x)=3x^{**}20-6x^{**}16+8x^{**}9-7x^{**}5+1$ și afișează rezultatul. Calculele se fac în flotantă dublă precizie.

În acest exemplu intervin puteri mari ale lui x . De aceea, pentru a evita expresii de forma:

$x*x*...*x$

unde x să se repete de 20 de ori, vom apela funcția *pow* care realizează ridicarea la putere. Ea are prototipul definit în fișierul *math.h* și acesta este:

`double pow(double x, double y);`

La revenire se returnează $x^{**}y$, rezultatul fiind de tip *double*.

PROGRAMUL BIII3

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

main() /* - citeste valoarea lui x;
           - calculeaza si afiseaza valoarea polinomului:
           a(x)=3x**20-6x**16+8x**9-7x**5+1. */
{
    double x;

    printf("tastati valoarea lui x = ");
    scanf("%lf", &x);
    printf("x=%g\ta(x)=%g\n", x,
           3.0*pow(x, 20.0) - 6.0*pow(x, 16.0) +
           8.0*pow(x, 9.0) - 7.0*pow(x, 5.0)
           + 1.0);
}
```

- 3.4 Să se scrie un program care citește valoarea variabilei x și a coeficienților polinomului:

$$q(x)=c_4*x^{**}4+c_3*x^{**}3+c_2*x^{**}2+c_1*x+c_0,$$

calculează valoarea polinomului $q(x)$ și afișează rezultatul. Calculele se fac în flotantă dublă precizie.

Calculul valorii unui polinom se poate face folosind funcția de biblioteca *poly*. Ea are prototipul definit în fișierul *math.h* și este:

`double poly(double x, int n, double c[]);`

Returnează valoarea polinomului de grad n , pentru valoarea lui x ,

coeficienții polinomului fiind $c[0], c[1], \dots, c[n]$; $c[0]$ este termenul liber; $c[1]$ este coeficientul lui x ; în general, $c[i]$ este coeficientul lui x la puterea i .

PROGRAMUL BIII4

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

main() /* - citește valoarea lui x și coeficienții c[0],c[1],...,c[4] ai polinomului
         q(x)=c[4]*x**4+c[3]*x**3+c[2]*x**2+c[1]*x+c[0]
         - calculează valoarea polinomului q(x);
         - afisează rezultatul. */
{
    double x,c[5];

    printf("valoarea lui x = ");
    scanf("%lf",&x);
    printf("coeficienții polinomului\n");
    printf("c0 = ");
    scanf("%lf",&c[0]);
    printf("c1 = ");
    scanf("%lf",&c[1]);
    printf("c2 = ");
    scanf("%lf",&c[2]);
    printf("c3 = ");
    scanf("%lf",&c[3]);
    printf("c4 = ");
    scanf("%lf",&c[4]);
    printf("x=%g\ntq(x)=%g\n",x,poly(x,4,c));
}
```

- 3.5 Să se scrie un program care citește cu ecou de la tastatură o literă mică și o afișează ca literă mare.

Conversiile literelor mici în litere mari se realizează simplu în cazul utilizării codului ASCII. Într-adevar, codurile ASCII ale literelor mici se află în intervalul [97,122], iar ale literelor mari în intervalul [65,90]. Aceste coduri corespund literelor în ordine alfabetică:

Literă mică	Codul ASCII	Literă mare	Codul ASCII
a	97	A	65
b	98	B	66
c	99	C	67
...			
z	122	Z	90

Codul ASCII al unei litere mari se obține din codul ASCII al aceleiași litere mici, scăzind valoarea 32. Deci, dacă variabila *lit* are ca valoare codul ASCII al unei litere mici, atunci expresia:

lit-32

are ca valoare codul ASCII al aceleiași litere, dar mari.

De obicei, diferența 32 dintre codurile ASCII ale aceleiași litere, se exprimă mai sugestiv prin expresia:

'a'-'A'

Această expresie are doi operanzi care sunt constante caracter, deci fiecare are tipul *int*. Înseamnă că tipul expresiei de mai sus este *int*. În acest fel, codul ASCII al literei mari se obține cu ajutorul expresiei:

lit-(‘a’-‘A’)

care se scrie mai simplu:

lit-'a'+'A'

PROGRAMUL BIII5

```
#include <conio.h>
main() /* citește cu ecou de la tastatura o literă mică și o afișează ca literă mare */
{
    putch(getche() - 'a' + 'A');
}
```

Observație:

În acest program se presupune că se tastează la terminal o literă mică.

- 3.6 Să se scrie un program care citește cu ecou de la tastatură o literă mare și o afișează ca literă mică.

În acest caz codul literelor mari se adună cu valoarea 32, adică cu diferența 'a'-'A'.

PROGRAMUL BIII6

```
#include <conio.h>
main() /* citește cu ecou de la tastatura o literă mare și o afișează ca literă mică */
{
    putch(getche() + 'a' - 'A');
}
```

- 3.7 Să se scrie un program care citește un număr ce reprezintă aria unei suprafețe exprimată în jugăre.

Se cere să se exprime aria respectivă în hectare, prăjini pătrate și stinjeni pătrați.

Aveam:

1 jugăr = 576 prăjini pătrate = 1600 stinjeni pătrați = 5754,6415 metri pătrați.

PROGRAMUL BIII7

```
#include <stdio.h>
main() /* - citeste un numar ce exprima aria unei suprafete in jugare;
         - afiseaza aria respectiva in:
           - prajini patrate;
           - stinjeni patrati;
           - hectare. */
{
    double aria;
    scanf("%lf", &aria);
    printf("aria=%g jugare\n", aria);
    printf("aria=%g prajini patrate\n", aria*576);
    printf("aria=%g stinjeni patrati\n", aria*1600);
    printf("aria=%g hectare\n", aria*0.57546415);
}
```

3.2.2. Regula conversiilor implicite

Această regulă se aplică la evaluarea expresiilor. Ea acționează atunci cind un operator binar se aplică la doi operanzi de tipuri diferite.

În cazul în care un operator se aplică la doi operanzi de tipuri diferite, operandul de tip "inferior" se convertește spre tipul "superior" al celuilalt operand și rezultatul este de tipul "superior".

Regula conversiilor implicite are în vedere și niște conversii generale independente de operatori. Ea se aplică în mai mulți pași, în ordinea indicată mai jos.

Înainte de toate se convertesc operanzele de tip *char* și *enum* (acest tip va fi definit mai tîrziu) în tipul *int*.

Dacă operatorul curent se aplică la operanze de același tip, atunci se execută operatorul respectiv, iar tipul rezultatului coincide cu tipul comun al operanzei. Dacă rezultatul aplicării operatorului reprezintă o valoare în afara limitelor tipului respectiv, atunci rezultatul este eronat (are loc o "depășire").

Dacă operatorul binar curent se aplică la operanze diferenți, atunci se face o conversie înainte de execuția operatorului, conform pașilor de mai jos:

1. Dacă unul din operanze este de tip *long double*, atunci celălalt operand se convertește spre tipul *long double* și rezultatul aplicării operatorului este de tip *long double*.
2. Altfel, dacă unul din operanze este de tip *double*, atunci celălalt operand se convertește spre tipul *double* și rezultatul aplicării operatorului este de tip *double*.
3. Altfel, dacă unul din operanze este de tip *float*, atunci celălalt operand se convertește spre tipul *float* și rezultatul aplicării operatorului este de tip

float.

4. Altfel, dacă unul din operanze este de tip *unsigned long*, atunci celălalt operand se convertește spre *unsigned long* și rezultatul aplicării operatorului este de tip *unsigned long*.
5. Altfel, dacă unul din operanze este de tip *long*, atunci celălalt operand se convertește spre tipul *long* și rezultatul aplicării operatorului este de tip *long*.
6. Altfel, unul din operanze trebuie să fie de tip *unsigned*, celălalt de tip *int* și acesta se convertește spre *unsigned*, iar rezultatul aplicării operatorului este de tip *unsigned*.

Aplicind regula de mai sus, la fiecare operator curent, în procesul de evaluare a unei expresii, se determină în final tipul expresiei respective.

Exemple:

1. int c;
...
c='a'+'A'
Constantele caracter 'a' și 'A' sunt de tip *int*, deci toți operanzele sunt de tip *int*. Nu este necesară nici o conversie pentru evaluarea expresiei.
2. int i;
...
12345*i
Operanzele sunt de tip *int*, expresia este de tip *int*; rezultatul înmulțirii este eronat dacă este în afara intervalului [-32768,32767].
3. int i;
long double a;
...
a*3+i
Ținând seama de prioritățea operatorilor, întîi se execută înmulțirea, apoi adunarea; expresia se evaluatează astfel:
 - 3 se convertește spre *long double* (regula 1), se realizează înmulțirea și rezultatul este de tip *long double*;
 - la rezultatul înmulțirii se adună *i* după ce, în prealabil, *i* se convertește spre *long double*, deci expresia este de tip *long double*.

Exerciții:

- 3.8 Să se scrie un program care citește doi întregi de la intrarea standard și afișează media lor cu o zecimală.

PROGRAMUL BIII8

```
#include <stdio.h>
main() /* citește doi întregi și afisează media lor aritmetică cu o zecimă */
{
    int a,b;
    scanf ("%d %d", &a,&b);
    printf ("a=%d\nb=%d\nmedia=%1.1f\n",a,b,(a+b)/2.0f);
}
```

Observații:

1. Pentru calculul mediei se folosește expresia:

$$(a+b)/2.0$$

Ea se evaluateaza astfel:

- Se calculează suma $a+b$; rezultatul este de tip *int*, deoarece ambii operanzi sint de tip *int*. Dacă suma nu aparține intervalului $[-32768,32767]$, rezultatul este eronat.
- Rezultatul adunării se convertește spre *float*, deoarece celălalt operand este de tip *float*, apoi se face împărțirea și rezultatul este de tip *float*.

2. Dacă s-ar fi utilizat expresia

$$(a+b)/2.0$$

suma $a+b$ s-ar fi convertit spre *double* deoarece împărțitorul este de tip *double*; în acest caz rezultatul este de tip *double*.

3. Expresia

$$(a+b)/2$$

realizează împărțirea întreagă dintre operanții $a+b$ și 2 care sint de tip *int*; de aceea, pentru a și b de paritate diferite, expresia de mai sus nu dă rezultatul corect.

- 3.9 Sa se scrie un program care citește măsura unui unghi în grade sexagesimale și afisează valoarea funcției sinus pentru unghiul respectiv.

Măsura unghiului în grade sexagesimale se tastează sub forma:

g m s

unde:

- | | |
|----------|-----------------------------------|
| <i>g</i> | - Întreg ce reprezinta gradele. |
| <i>m</i> | - Întreg ce reprezinta minutele. |
| <i>s</i> | - Întreg ce reprezinta secundele. |

Pentru calculul funcției sinus se apelează funcția *sin* aflată în biblioteca

limbajelor C și C++.

Prototipul ei este:

double sin(double);

și se află în fișierul *math.h*.

Parametrul funcției este măsura unghiului în radiani pentru care se calculează valoarea funcției *sin*. Conversia măsurii unghiului din grade sexagesimale în radiani se realizează folosind expresia:

$$(g+m/60.0+s/3600.0)*PI/180.0$$

unde

$$PI = 3.14159265358979$$

PROGRAMUL BIII9

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

#define PI 3.14159265358979

main() /* citește masura unui unghi în grade sexagesimale și afisează valoarea funcției sin pentru unghiul respectiv.*/
{
    int g,m,s;
    scanf ("%d %d %d", &g,&m,&s);
    printf ("grade:%d\nminute:%d\ntsecunde:%d\n",g,m,s);
    printf ("sin=%1.13f\n",sin((g+m/60.0 + s/3600.0)*PI/180.0));
}
```

3.2.3. Operatorii de relație

Operatorii de relație sint 4:

- | | |
|----|----------------------|
| < | - mai mic; |
| <= | - mai mic sau egal; |
| > | - mai mare; |
| >= | - mai mare sau egal. |

Ei au aceeași prioritate, care este mai mică decit a operatorilor aditivi.

O expresie de forma

E1 operator_de_relație E2

unde:

E1 și E2 - Sunt expresii, care sunt unele din valorile 0 sau 1.

Astfel, expresia are valoarea 1, dacă valorile celor două expresii satisfac relația prin care acestea sunt legate și 0 în caz contrar.

De obicei, se spune despre o expresie relațională că ea are valoarea *adevărat* dacă subexpresiile ei satisfac operatorul relațional prin care sunt legate; în caz

contrar se spune că expresia are valoarea *fals*.

În limbajele C și C++ nu există valori logice speciale. Valoarea *fals* se reprezintă prin valoarea 0 (zero), iar valoarea *adevărat* printr-o valoare diferită de zero.

Exerciții:

3.10 Să se scrie un program care citește două numere, primul este de tip *int*, iar al doilea este de tip *double*. Programul afișează valoarea 2 dacă numărul flotant este mai mare decât cel întreg și 1 în caz contrar.

Presupunem că primul număr se atribuie variabilei *i*, iar cel de al doilea variabilei *f*.

Așunci expresia

$f > i$

are valoarea 1 dacă numărul flotant este mai mare decât cel întreg și 0 în caz contrar. Pentru evaluarea relației de mai sus, valoarea variabilei *i* se convertește spre *double* și abea apoi se execută operatorul de relație. Tipul expresiei este *int*.

Programul de față trebuie să afișeze valoarea acestei expresii mărită cu 1, ceea ce se obține folosind expresia

$(f > i) + 1$

Parantezele sunt necesare deoarece operatorul *+* este mai prioritar decât *>*. În lipsa lor, se compară valoarea lui *f* cu valoarea sumei *i+1*.

PROGRAMUL BII110

```
#include <stdio.h>
main() /* - citeste pe i si f;
           - afiseaza 2 daca f este mai mare decat i si 1 in caz contrar */
{
    int i;
    double f;

    scanf("%d %lf", &i, &f);
    printf("i=%d\ tf=%f\ t%d\n", i, f, (f > i) + 1 );
}
```

3.2.4. Operatorii de egalitate

Operatorii de egalitate sunt doi:

$==$ - egal;
 $!=$ - diferit.

Ei au aceeași prioritate care este imediat mai mică decât cea a operatorilor de relație.

O expresie de forma:

E1 operator_de_egalitate E2

unde:

E1 și E2 - Sunt expresii, are una din valorile 0 sau 1.

Astfel, expresia de mai sus are valoarea 1, dacă valorile expresiilor *E1* și *E2* satisfac operatorul de egalitate prin care sunt legate și zero în caz contrar. În primul caz, se obișnuiește să se spună că expresia de mai sus este *adevărată*, iar în cel de al doilea caz, că ea este *falsă*.

Exemple:

1.

```
int a;
long b;
```

Expresia
 $b == a * a$ are valoarea adevărat (1) dacă *b* este egal cu patratul lui *a*; altfel ea are valoarea fals (0).
2.

```
int x;
```

Expresia
 $x \% 4 == 0$ are valoarea adevărat dacă *x* este multiplu de 4 și fals în caz contrar.
3.

```
int x;
```

Expresia
 $x \% 100 != 0$ are valoarea adevărat dacă *x* nu este multiplu de 100 și zero în caz contrar.
4. Expresia

```
getchar() != EOF
```

are valoarea adevărat dacă macroul *getchar* a citit un caracter diferit de sfîrșitul de fișier.
5.

```
int x;
scanf("%d", &x) == 1
```

Această expresie are valoarea adevărat dacă *scanf* a citit de la terminalul standard un întreg (*scanf* returnează numărul cîmpurilor citite de la terminalul standard). În caz contrar, adică dacă la intrare nu se află un întreg, expresia de mai sus are valoarea fals.

3.2.5. Operatorii logici

Aceștia sunt:

$!$ - negația logică (operator unar);
 $\&&$ - și logic;
 $\|$ - sau logic.

Negația logică are aceeași prioritate ca și ceilalți operatori unari ai limbajului C. De altfel, toți operatorii unari au aceeași prioritate care este imediat mai mare decit cea a operatorilor multiplicativi.

Expresia

$\text{!} \text{operand}$

are valoarea zero (fals) daca operand are o valoare diferita de zero și valoarea unu (adevarat) daca operand are valoarea zero.

Operatorul $\&\&$ are prioritatea mai mica decit cei de egalitate. O expresie de forma:

$E1\&\&E2$

are valoarea 1, daca expresiile E1 și E2 sunt ambele diferite de zero. În rest, expresia de mai sus are valoarea zero.

Operatorul $\|$ are prioritatea imediat mai mică decit operatorul $\&\&$. Expresia

$E1\|E2$

are valoarea zero, daca ambele expresii E1 și E2 la care se aplică operatorul $\|$ au valoarea zero. În rest, expresia de mai sus are valoarea unu.

Operatorii logici binari ($\&\&$ și $\|$) se evaluatează de la stanga la dreapta. Dacă la evaluarea unei expresii se ajunge într-un punct în care se cunoaște valoarea de adevăr (fals sau adevărat) a intregii expresii, atunci restul expresiei, aflată în dreapta punctului respectiv, nu se mai evaluatează.

Exemple:

1. $\text{!}a\&\&b$

Dacă a are valoarea zero și b este diferit de zero, atunci expresia de mai sus are valoarea unu (adevărat).

Într-adevăr, dacă a are valoarea zero, atunci $\text{!}a$ are valoarea unu.

Dacă b este diferit de zero, atunci ambii operanzi ai operatorului $\&\&$ sunt diferenți de zero și deci întreaga expresie are valoarea unu.

2. $\text{!}a\|b$

Aceasta expresie are valoarea zero dacă a are valoarea diferită de zero, iar b are valoarea zero. În acest caz $\text{!}a$ are valoarea zero și deci ambii operanzi ai operatorului $\|$ au valoarea zero. Conform definiției operatorului $\|$, rezultă că întreaga expresie are valoarea zero.

3. $\text{!}a\&\&b\|\text{!}b\&\&a$

Această expresie are valoarea zero daca a și b au ambi valoarea zero sau ambi sint diferenți de zero. În caz contrar, expresia are valoarea unu. Acest lucru rezulta din tabela de mai jos:

a	b	$\text{!}a$	$\text{!}b$	$\text{!}a\&\&b$	$\text{!}b\&\&a$	$\text{!}a\&\&b\ \text{!}b\&\&a$
0	0	1	1	0	0	0
diferit de zero	diferit de zero	0	0	0	0	0
diferit de zero	0	0	1	0	1	1
0	diferit de zero	1	0	1	0	1

Din cele de mai sus rezultă că expresia

$\text{!}a\&\&b\|\text{!}b\&\&a$

realizează sau logic exclusiv între a și b .

Pentru $a=0$ și b diferit de zero, expresia de mai sus se evaluatează astfel:

- Se evaluatează $\text{!}a$ și se obține valoarea unu.
- Se aplică operatorul și logic $\text{!}b$ și se obține unu.

Deoarece $\text{!}a\&\&b$ este legat de restul expresiei prin operatorul sau logic, rezultă că în acest moment se poate afirma că întreaga expresie este adevărată (are valoarea unu). De aceea, în acest caz nu se mai evaluatează restul expresiei, adică termenul $\text{!}b\&\&a$.

Exerciții:

- 3.11 Să se scrie un program care citește un număr și afișează unu dacă numărul respectiv aparține intervalului $[-1000,1000]$ și zero în caz contrar.

Dacă notăm cu a numărul citit, atunci el trebuie să satisfacă simultan relațiile: $a \geq -1000$

și

$a \leq 1000$

Acest fapt se poate exprima cu ajutorul expresiei:

$a \geq -1000 \&\& a \leq 1000$.

PROGRAMUL BIIII1

```
#include <stdio.h>
main() /* citește pe a și afișează unu dacă numarul citit aparține intervalului [-1000,1000] și zero în caz contrar */
{
    double a;

    scanf("%lf", &a);
    printf("a=%g\t%d\n", a, a >= -1000 && a <= 1000);
}
```

- 3.12 Să se scrie un program care citește un întreg din intervalul [1600,4900], ce reprezintă un an calendaristic, afișează unu dacă anul este bisect și zero în caz contrar sau dacă anul nu aparține intervalului indicat mai sus.

Un an, din calendarul gregorian, este bisect dacă este multiplu de patru și nu este multiplu de o sută sau dacă este multiplu de patru sute. Această regulă este valabilă pentru anii care nu sunt anteriori anului 1600.

Calendarul gregorian a fost introdus datorită faptului că anul iulian a fost prea lung.

Conform calendarului iulian, un an era bisect, dacă era divizibil cu patru. Din această cauză la 133 de ani se înregistrează o diferență de o zi față de anul tropic. În felul acesta, în anul 1582 s-a constatat o diferență de 10 zile întârziere și s-a hotărât corectarea diferenței respective convenind că după ziua de joi 4 octombrie 1582 să urmeze ziua de vineri 15 octombrie 1582. De asemenea, s-a stabilit că durata unui an calendaristic să fie:

$$365 + \frac{1}{4} - \frac{1}{100} + \frac{1}{400} = 365,2425 \text{ zile}$$

În această expresie termenii au semnificația:

365	- Numărul de zile ale anului iulian.
$\frac{1}{4}$	- Adăosul de un sfert de zi, adică de o zi la patru ani, conform calendarului iulian.
$\frac{1}{100}$	- Eliminarea unei zile la fiecare o sută de ani. Deci un astfel de an nu mai este bisect, deși el este multiplu de patru.
$\frac{1}{400}$	- Adăugarea unei zile la fiecare 400 de ani. În felul acesta, un an multiplu de 400 are o zi în plus, deci este bisect.

Calendarul gregorian, având durată anului egală cu 365, 2425, are o eroare de aproximativ 26 de secunde. Aceasta înseamnă că regula indicată mai sus pentru a stabili anii bisecți (care au 366 de zile) este valabilă pentru anii următori anului 1582 și care nu sunt anteriori anului 4900.

Într-adevăr, eroarea de 26 de secunde dintr-un an gregorian devine o zi abea la aproximativ 3323 de ani, deci după anul 4900 ($1582+3323=4905$). De aceea, regula de mai sus se poate aplica pentru anii din intervalul [1600,4900].

Dacă notăm cu an anul calendaristic, atunci el este bisect dacă de exemplu:
 $an \% 4 == 0$ (an este multiplu de 4)

și
 $an \% 100 != 0$ (an nu este multiplu de o sută).

Deci anul este bisect dacă expresia

$$(1) \ an \% 4 == 0 \ \&\& \ an \% 100 != 0$$

este adevărată.

De asemenea, anul este bisect și în cazul în care expresia

$$(2) \ an \% 400 == 0 \text{ (an este multiplu de 400)}$$

este adevărată. Deci anul este bisect dacă este adevarata expresia (1) sau (2), adică dacă este adevarată expresia:

$$an \% 4 == 0 \ \&\& \ an \% 100 != 0 \ || \ an \% 400 == 0$$

PROGRAMUL BIII12

```
#include <stdio.h>
main() /* citeste un intreg care reprezinta un an calendaristic din intervalul [1600,4900]
         și afiseaza dacă anul este bisect și zero dacă nu este sau nu aparține
         intervalului indicat */
{
    int an;

    scanf("%d", &an);
    printf("an=%d\n", an,
           'an >= 1600 && an <= 4900 && (an%4 == 0 &&
           an%100 != 0 || an%400 == 0));
}
```

- 3.13 Să se scrie un program care citește un întreg din intervalul [1600,4900] ce reprezintă un an calendaristic și afișează numărul de zile din anul respectiv.

PROGRAMUL BIII13

```
#include <stdio.h>
main() /* citeste un intreg care reprezinta un an calendaristic din intervalul [1600,4900] și
         afiseaza numarul de zile din anul respectiv */
{
    int an;

    scanf("%d", &an);
    printf("anul %d are %d zile\n", an,
           365 + (an%4 == 0 &&
           an%100 != 0 || an%400 == 0));
}
```

Observație:

În acest program se consideră că anul tastat la terminalul standard aparține intervalului [1600,4900].

3.2.6. Operatorii logici pe biți

Operatorii logici pe biți sunt:

- | | |
|--------|---|
| \sim | - complement față de unu (operator unar); |
| $<<$ | - deplasare stanga; |
| $>>$ | - deplasare dreapta; |
| $\&$ | - și logic pe biți; |
| $^$ | - sau exclusiv logic pe biți; |
| $ $ | - sau logic pe biți. |

Acești operatori se aplică la operanzi de tip întreg. Ei se execută bit cu bit și operanții se extind la 16 biți dacă este necesar.

Complementul față de unu are aceeași prioritate ca celelalte operatori unari. El schimbă fiecare bit 1 al operandului cu zero și fiecare bit zero al acestuia cu 1.

Exemple:

1. ~ 1234

1234 se reprezintă în binar astfel:

10011010010

Se extinde la 16 biți:

0000010011010010

apoi se aplică operatorul de complementare față de 1:

1111101100101101

sau în octal:

0175455

2. ~ 1234

Operatorii unari au aceeași prioritate și se asociază de la dreapta la stînga, deci întîi se realizează negativarea și apoi complementarea față de 1.

Numărul 1234 se reprezintă în binar:

10011010010

se negativează:

1111101100101110

apoi se complementează față de 1:

10011010001

sau în octal:

02321

3. ~ 1234

În acest caz, întîi se determină complementul față de 1, apoi se negativează numărul.

1234 în binar este:

10011010010

se complementează față de 1:

1111101100101101

se negativează:

10011010011

sau în octal:

02323

Operatorii de deplasare sunt operatori binari. Ei au aceeași prioritate, care este imediat mai mică decât a operatorilor aditivi și imediat mai mare decât prioritatea operatorilor relaționali.

Operatorul $<<$ realizează o deplasare la stînga a valorii primului său operand

cu un număr de poziții binare egal cu valoarea celui de-al doilea operand al său. Această operație este echivalentă cu inmulțirea cu puteri ale lui 2.

În mod analog, operatorul $>>$ realizează o deplasare la dreapta a valorii primului său operand cu un număr de poziții binare egal cu valoarea celui de-al doilea operand al său. Această operație este echivalentă cu o împărțire cu puteri ale lui 2.

Exemple:

1. int a;

a<<3

are aceeași valoare ca și

a*8

adică se inmulțește a cu 2 la puterea 3.

2. int a;

a>>4

are aceeași valoare ca și

a/16

adică a se imparte la 2 la puterea 4.

3. $\sim 0<<3$

Această expresie se evaluatează astfel:

a. Se extinde zero pe 16 biți:

0000000000000000

b. Se aplică operatorul de complementare față de unu:

1111111111111111

c. Se fac 3 deplasări spre stînga:

1111111111110000

sau în octal:

177770

4. $\sim(\sim 0<<3)$

Rezultatul din exemplul precedent se complementează față de unu:

0000000000001111

Operatorul și *logic pe biți* se execută bit cu bit, conform tabelei de mai jos:

1&1=1

1&0=0

0&1=0

0&0=0

Operatorul $\&$ are prioritatea imediat mai mică decât operatorii de egalitate. Se utilizează la anulări de biți.

Exemple:

1. int a;
a&0377

are ca valoare, valoarea octetului mai puțin semnificativ al valorii variabilei *a*.

Într-adevăr, se extinde constanta octală 0377 la 16 biți:

00000001111111

apoi se face și logic pe biți între această constantă și valoarea variabilei *a*. Se păstrează ultimii 8 biți ai lui *a*, primii 8 biți anulindu-se.

2. int a;
a&0177400

are ca valoare, valoarea lui *a* după ce s-au anulat ultimii 8 biți ai săi. Aceasta rezultă din faptul că, constanta octală 0177400 are în binar valoarea:

1111111000000000

3. int a;
a&0177776

are ca valoare cel mai mare număr par care nu-l depășește pe *a*.

Diferența dintre operatorul și logic (**&&**) și operatorul și logic pe biți (**&**) constă în aceea că primul se realizează global, față de al doilea care se realizează pe biți.

Așa de exemplu, dacă $x=2$ și $y=1$, atunci $x\&\&y$ are valoarea 1, deoarece ambele operanzi sunt diferenți de zero. În schimb, $x\&y$ are valoarea zero. Într-adevăr, în acest caz se realizează un și pe biți cu valorile:

$x=000000000000000010$
 $y=0000000000000001$

 $x\&y=0000000000000000$

Operatorul sau exclusiv pe biți se execută bit cu bit, conform tabelei de mai jos:

$1^1=0$
 $1^0=1$
 $0^1=1$
 $0^0=0$

Operatorul **^** are prioritatea imediat mai mică decit operatorul **&**. Se utilizează pentru a anula sau poziționa diferenți biți.

Exemple:

1. int a;
a^a

are valoarea 0. Într-adevăr, cei doi operanzi fiind identici, se executa operatorul **^** pentru biți identici, ori în acest caz el are ca rezultat pe zero.

2. int a;
a^1

are o valoare care depinde de paritatea lui *a*. Într-adevăr, dacă *a* este par, atunci ultimul său bit este zero.

Expresia

a^1

setează la valoarea unu ultimul bit a lui *a*. În felul acesta, rezultatul este egal cu cel mai mic număr impar care-l depășește pe *a*.

Dacă *a* este impar, atunci ultimul bit are valoarea unu.

Expresia

a^1

anulează ultimul bit a lui *a*. Deci, rezultatul este egal cu cel mai mare număr par care nu-l depășește pe *a*.

Operatorul sau logic pe biți se execută bit cu bit, conform tabelei de mai jos:

$1|1=1$
 $1|0=1$
 $0|1=1$
 $0|0=0$

Acest operator are prioritatea imediat mai mică decit operatorul sau exclusiv pe biți (**^**) și imediat mai mare decit operatorul și logic (**&&**).

Operatorul **|** se folosește la setări de biți.

Exemple:

1. int a;
all

are o valoare a cărui ultim bit este unu, indiferent de valoarea variabilei *a*. Ceilalți biți coincid cu cei ai lui *a*. De aceea, valoarea acestei expresii este cel mai mic număr impar care nu este mai mic decit *a*.

2. int a;
al0100000

are o valoare cu bitul cel mai semnificativ setat, indiferent de valoarea variabilei *a*. Ceilalți biți coincid cu cei ai lui *a*.

3.2.7. Operatorii de atribuire

Operatorul de atribuire, în forma cea mai simplă, se notează prin caracterul " $=$ ". El se utilizează în expresii de forma:

$v = (\text{expresie})$

unde:

- v - Este o variabilă simplă, referențiază un element din tablou (variabilă cu indicii) sau de structură.

Operatorul de atribuire are o prioritate mai mică decât toți operatorii pe care i-am întîlnit pînă în prezent. De aceea, parantezele din construcția indicată mai sus, de obicei nu sunt necesare.

Operatorul de atribuire are ca efect atribuirea valorii expresiei aflată în dreapta semnului de atribuire variabilei v . Ulterior o să vedem că în stînga semnului de atribuire se poate afla chiar o expresie care definește o adresă.

La o atribuire se face și o conversie dacă este necesar. Astfel, valoarea expresiei din dreapta semnului de atribuire se va converti spre tipul variabilei din stînga lui înainte de a se face atribuirea; dacă cele două tipuri sunt diferite.

Expresia

$v = \text{expresie}$

este o *expresie de atribuire*. Ea are ca *valoare* valoarea care se atribuie, iar ca *tip*, tipul variabilei v .

Rezultă că o construcție de forma:

$v_1 = (v = \text{expresie})$

este corectă și reprezintă tot o expresie de atribuire: lui v_1 îi se atribuie valoarea atribuită în prealabil lui v , facîndu-se și conversie dacă este necesar.

Deoarece operatorii de atribuire se asociază de la *dreapta spre stînga*, expresia de mai sus se poate scrie fără paranteze:

$v_1 = v = \text{expresie}$

În general, o expresie de atribuire are forma:

$v_n = \dots = v_1 = v = \text{expresie}$

Valoarea expresiei aflate în partea dreaptă se atribuie intîi lui v , apoi lui v_1 și așa mai departe și în final se atribuie lui v_n . Atribuirile sunt precedate de conversii în cazul în care valoarea care se atribuie este de un tip diferit decît tipul variabilei la care se face atribuirea.

Pentru operatorii de atribuire, în afara semnului $=$ se mai poate folosi și succesiunea de caractere:

$op =$

unde op este un operator binar aritmetic sau logic pe biți. Deci op poate fi unul

din operatorii:

$/ \% * - + << >> \& ^ |$

Această construcție se folosește pentru a face prescurtări.
Expresia

$v op = \text{expresie}$

este echivalentă cu expresia de atribuire:

$v = v op (\text{expresie})$

Expresiile de atribuire pot fi folosite peste tot în program unde este legal să apară o expresie. Acest fapt permite adesea să se facă compactări în programul sursă.

Exemple:

1. $\text{int } a;$
 $a=10$
variabila a îi se atribuie valoarea 10.

2. $\text{int } i;$
 $i=i+3$
valoarea variabilei i se mărește cu 3.

3. $\text{int } i;$
 $i += 3$
are același efect ca și expresia de la exemplul precedent.

4. Expresia de atribuire:
 $a[i*3+10][j*2-3]=a[i*3+10][j*2-3]*x$
se scrie prescurtat astfel:
 $a[i*3+10][j*2-3]*=x$

Exerciții:

3.14 Să se scrie un program care citește valoarea lui x , calculează valorile expresiilor:

$4x*x+3x$

$4x*x+3x+1$

și

$(4x*x+3x-1)/(4x*x+3x+1)$

și afișează valorile respective.

PROGRAMUL BIII14

```
#include <stdio.h>
main() /* citește pe x și afisează valorile expresiilor:
        4x*x + 3x
        4x*x + 3x + 1
```

```

(4x*x + 3x - 1)/(4x*x + 3x + 1) */

{
    double x,y;

    scanf("%lf", &x);
    printf("x=%g\ny=4x*x + 3x=%g\n", x, y=4*x*x + 3*x );
    printf("4x*x+3x+1 = %g\n", (4*x*x+3*x+1)/
          = %g\n", y+1, (y-1)/(y+1));
}

```

Observație:

Parametrul:

$$y=4*x*x+3*x$$

din primul apel al funcției *printf* este o expresie de atribuire. Prin intermediu ei se atribuie lui *y* valoarea expresiei

$$4*x*x+3*x$$

Valoarea expresiei de atribuire coincide cu valoarea atribuită lui *y*. *Tipul expresiei* de atribuire coincide cu tipul lui *y*, deci tipul ei este *double*.

- 3.15 Să se scrie un program care citește valoarea lui *x*, calculează și afișează valorile expresiilor:

$$x^{*}10, x^{*}20, x^{*}30 \text{ și } x^{*}40.$$

PROGRAMUL BIII15

```

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

main() /* citește pe x și afișează valorile expresiilor:
           x**10
           x**20
           x**30
           x**40 */
{
    double x,y,z;

    scanf("%lf", &x);
    printf("x=%g\nx**10=%g\n", x, y = pow(x,10.0));
    printf("\nx**20=%g\n", z= y*y);
    printf("\nx**30=%g\n", y*z);
    printf("\nx**40=%g\n", z*z);
}

```

- 3.16 Să se scrie un program care citește valoarea lui *x*, afișează valorile lui *x*, $\{x\}$, calculează și afișează valorile expresiilor:

$$7[x]*[x]-3[x]+10$$

și

$$7\{x\}*\{x\}-3\{x\}+10$$

unde:

- $\{x\}$ - Notează partea întreagă a lui *x* ($\{x\} \leq x$).
- $\{x\}$ - Notează partea fracționara a lui *x* ($\{x\} = x - \lfloor x \rfloor$).

PROGRAMUL BIII16

```

#include <stdio.h>
main() /* citește pe x;
           afișează pe x, [x], {x};
           calculează și afișează valorile expresiilor:
           7[x]*[x]-3[x]+10
           și
           7\{x\}*\{x\}-3\{x\}+10 */
{
    int i;
    double x,y;

    scanf("%lf", &x);
    printf("x=%d\n", x);
    printf("[x]=%d\n", i = x);
    printf("{x}=%f\n", y = x - i);
    printf("7[x]*[x] - 3[x] + 10=%d\n", 7*i*i - 3*i + 10);
    printf("7\{x\}*\{x\} - 3\{x\} + 10=%g\n", 7*y*y - 3*y + 10);
}

```

3.2.8. Operatorii de incrementare și decrementare

Acești operatori sunt unari și au aceeași prioritate ca și ceilalți operatori unari ai limbajului C.

Operatorul de *incrementare* se notează cu **++**, iar cel de *decrementare* cu **--**.

Operatorul de incrementare mărește valoarea operandului său cu 1, iar cel de decrementare micșorează valoarea operandului sau cu 1.

Acești operatori pot fi folosiți *prefixați*:

++operand

--operand

sau *postfixați*:

operand++

operand--

În cazul în care un operator de incrementare sau decrementare este prefixat, se folosește valoarea operandului la care s-a aplicat operatorul respectiv.

În cazul în care un operator de incrementare sau decrementare este postfixat, se folosește valoarea operandului dinaintea aplicării operatorului respectiv.

Exemplu:

Presupunem că *x* are valoarea 3. Dacă considerăm expresia de atribuire

`y=++x`

atunci lui `y` îi se atribuie valoarea 4 (la atribuire se folosește valoarea incrementată).

Dacă utilizăm expresia de atribuire

`y=x++`

pentru `x=3`, atunci lui `y` îi se atribuie valoarea 3 (la atribuire se folosește valoarea dinaintea incrementării).

În ambele cazuri valoarea lui `x` s-a mărit cu 1.

3.2.9. Operatorul de forțare a tipului sau de conversie explicită (expresie cast)

Adesea dorim să specificăm conversia valorii unui operand spre un *tip dat*. Acest lucru este posibil folosind o construcție de forma:

(tip) operand

Prinț-o astfel de construcție valoarea operandului se convertește spre tipul indicat în paranteze.

În construcția de mai sus (*tip*) se consideră că este un operator unar. Îl vom numi *operator de forțare a tipului sau de conversie explicită*.

Construcția de mai sus o vom numi *expresie cast*.

Exemplu:

Presupunem că funcția `f` are un parametru de tip *double*. Fie declarația:

`int n;`

Atunci, pentru a apela `f` cu parametrul `n`, este necesar ca, în prealabil, `n` să se convertească spre *double*. Acest lucru se poate realiza prinț-o atribuire:

```
double x;  
...  
f((double)n);
```

Un alt mod mai simplu de conversie a parametrului intreg spre tipul *double* este utilizarea unei *expresii cast*:

`f((double)n);`

Operatorul *(tip)* fiind unar, are aceeași prioritate ca și ceilalți operatori unari ai limbajului C.

Exerciții:

3.17 Să se scrie un program care citește un întreg și afișează rădăcina pătrată din numărul respectiv.

PROGRAMUL BIII17

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

main() /* - citeste pe n;
           - calculeaza si afiseaza radacina patrata din n */
{
    long n;

    scanf ("%ld", &n);
    printf ("n=%ld\nsqrt(n)=%g\n", n, sqrt ((double)n));
}
```

Observație:

În acest program s-a utilizat funcția *sqrt* pentru extragerea rădăcinii pătrate. Ea are prototipul:

`double sqrt(double);`

Acest prototip este definit în fișierul *math.h*.

2.18 Să se scrie un program care citește pe `n` de tip intreg și afișează valoarea expresiei $n/(n+1)$ cu 15 zecimale.

PROGRAMUL BIII18

```
#include <stdio.h>
main() /* - citeste pe n;
           - calculeaza si afiseaza pe n/(n+1) cu 15 zecimale */
{
    long n;

    scanf ("%ld", &n);
    printf ("n=%ld\nn/(n+1)=% .15g\n", n,
           (double)n/(n+1));
}
```

Observație:

Expresia $n/(n+1)$ realizează împărțirea întreagă a lui `n` la `n+1`. Pentru a obține cîtul împărțirii cu 15 zecimale este necesar să se efectueze împărțirea neîntreagă. În acest scop s-a convertit operandul `n` spre tipul *double*. În felul acesta, conform regulei conversiilor implicate, se convertește spre *double* și cel de al doilea operand și apoi se face împărțirea celor doi operanzi flotați.

3.2.10. Operatorul dimensiune

Dimensiunea în octeți a unei date sau al unui tip se poate determina folosind operatorul *sizeof*. El poate fi folosit sub forma:

sizeof *data*
sau
sizeof(*tip*)
unde:
data

- Poate fi:
 - un nume de variabilă simplă;
 - un nume de tablou;
 - un nume de structură;
 - referirea la un element de tablou (variabila cu indice);
 - referirea la elementul unei structuri.
- Poate fi:
 - un cuvânt sau cuvintele cheie ale unui tip predefinit;
 - o construcție care definește un tip.

tip

Expresia

sizeof *data*

poate fi scrisă și sub forma **sizeof(data)**. Ea are ca valoare dimensiunea în octeți a operandului "data". Astfel, dacă data este:

numele unei variabile simple

- Atunci rezultatul va fi numărul de octeți alocați variabilei respective.

numele unui tablou

- Atunci rezultatul va fi numărul de octeți al zonei de memorie alocate tabloului respectiv.

numele unei structuri

- Atunci rezultatul va fi numărul de octeți al zonei de memorie alocate structurii respective.

etc.

Expresia

sizeof(*tip*)

are ca valoare numărul de octeți alocați pentru reprezentarea unei date de tip *tip*.

Operatorul dimensiune (**sizeof**) este unar și are aceeași prioritate ca restul operatorilor unari ai limbajului C.

Exemple:

1.

```
int x;
```


sizeof *x*
are valoarea 2, deoarece pentru variabila *x* se alocă 2 octeți.
2.

```
long double y[7];
```


sizeof *y[3]*
are valoarea 10, deoarece pentru o dată de tip *long double* se alocă 10 octeți.

sizeof *y*
are valoarea 70, deoarece tabloul are 7 elemente în total.

3. **sizeof** (*char*)
are valoarea 1, deoarece pentru o dată de tip *char* se alocă un octet.

3.2.11. Operatorul adresa

Operatorul adresa este unar și se notează prin caracterul &. El se aplică pentru a determina adresa de început a zonei de memorie alocată unei date. În forma cea mai simplă, acest operator se utilizează în construcții de forma:

&nume

unde:

nume

- Este numele unei variabile simple sau a unei structuri.

Menționăm că în cazul în care *nume* este numele unui tablou, atunci acesta are ca valoare chiar adresa de început a zonei de memorie alocate tabloului respectiv. Deci, în acest caz nu se mai utilizează operatorul adresa &.

Acest operator a fost utilizat deja freeevent la apelul funcției *scanf*.

Mai tîrziu o să vedem și alte utilizări ale acestui operator.

Fieind un operator unar, el are aceeași prioritate ca și ceilalți operatori unari ai limbajului C.

Operatorul unar & are în limbajul C++ și o altă utilizare așa cum se vă vede mai tîrziu.

3.2.12. Operatorii paranteză

Parantezele rotunde se utilizează și pentru a include o expresie, și la apelul funcțiilor.

Amintim că o expresie inclusă în paranteze rotunde formează un operand. În felul acesta se poate impune o altă ordine în efectuarea operațiilor, deoarece ceea ce rezulta din prioritatea și asociativitatea operatorilor.

Operanții obținuți prin incluzarea unei expresii între paranteze impun anumite limite asupra operatorilor. De exemplu, la un astfel de operand nu se pot aplica operanții de incrementare și decrementare sau operatorul adresa. Astfel construcțiile:

(i+10)++ *--(x+y)* *&(x+y)*

sunt eronate.

La apelul unei funcții, lista parametrilor efectivi se include între paranteze rotunde. În acest caz se obișnuiește să se spună că parantezele rotunde sunt *operatori de apel de funcție*.

Parantezele patrate includ expresii care reprezintă *indice*. Ele se numesc *operatori de indexare*.

Parantezele sunt operatori de prioritate maxima. Operatorii unari au prioritatea imediat mai mică decât parantezele.

3.2.13. Operatorii condiționali

Operatorii condiționali permit construirea de expresii a căror valoare să depindă de valoarea unei condiții.

Prin *condiție* înțelegem o expresie care poate avea două valori: *adevărat* sau *fals*. În limbajul C, o condiție se reprezintă printr-o expresie oricare. Ea are valoarea *adevărat* dacă este diferită de zero și *fals* în caz contrar.

Numim *expresie condițională*, o expresie și carei valoare și tip este dependenta de valoarea unei condiții.

Un exemplu simplu de expresie condițională este aceea care are ca valoare maximul dintre două numere. Astfel, dacă a și b sunt două numere, atunci valoarea expresiei care calculează maximul dintre a și b depinde de valoarea condiției $a > b$.

Intr-adevar, dacă $a > b$ are valoarea adevarat, atunci expresia respectivă are valoarea a . În caz contrar (adică $a \geq b$ are valoarea fals) expresia de calcul a maximului va avea valoarea b .

O expresie condițională are formatul:

$E1?E2:E3$

unde:

$E1, E2$ și $E3$ - Sunt expresii.

Aceasta expresie se evaluatează astfel:

- Se determină valoarea expresiei $E1$.
- Dacă $E1$ are o valoare diferită de zero (are valoarea adevarat), atunci valoarea și tipul expresiei condiționale coincide cu valoarea și tipul expresiei $E2$. Altfel (adică $E1$ are valoarea zero) valoarea și tipul expresiei condiționale coincide cu valoarea și tipul expresiei $E3$.

De aici rezultă că valoarea și tipul expresiei condiționale depinde de valoarea expresiei $E1$.

Folosind formatul de mai sus, calculul maximului dintre a și b se exprimă cu ajutorul expresiei condiționale:

$(a > b) ? a : b$

unde:

- | | |
|-----------|----------------------------------|
| $(a > b)$ | - Este expresia $E1$ (condiția). |
| a | - Este expresia $E2$. |
| b | - Este expresia $E3$. |

Operatorii condiționali sunt:

? și :

rdinea indicată în formatul expresiei

Ei se folosesc totdeauna împreună și decât prioritatea operatorului sau logic (II) condiționale.

Ei au prioritatea imediat mai mare decât prioritățile celorlalți operatorilor de atribuire. Înțind seama de și imediat mai mare decât prioritățile celorlalți operatori, se pot omite parantezele din expresia pentru prioritatea operatorilor, rezultând:

$a > b ? a : b$

Este un caz particular de expresie și deci ea poate fi utilizată oriunde este legal să apară, în program, o expresie.

De exemplu:

$v = (E1 ? E2 : E3)$

este o expresie de atribuire: în partea dreaptă a semnului de atribuire se află o expresie condițională. Ea poate fi scrisă mai simplu fără paranteze, deoarece prioritățile de atribuire este mai puțin prioritățile decât operatorii condiționali.

Operatorii condiționali se asociază de la dreapta la stînga la fel ca și operatorii unari de atribuire.

Exerciții:

2.9 Să se scrie un program care citește două numere și afișează maximul dintre ele.

PROGRAMUL BIII19

```
#include <stdio.h>
main() /* citește două numere și afișează maximul dintre ele */
{
    double a,b;
    scanf("%lf %lf", &a,&b);
    printf("a=%g\tb=%g\tmax(a,b)=%g\n", a,b,a > b ? a : b );
}
```

3.20 Să se scrie un program care citește un număr și afișează valoarea lui absolută.

PROGRAMUL BIII20

```
#include <stdio.h>
main() /* citește un număr și afișează valoarea lui absolută */
{
    double a;
    scanf("%lf", &a);
    printf("a=%g\tabs(a)=%g\n", a, a < 0 ? -a : a );
}
```

3.21 Să se scrie un program care citește doi numere întregi și afișează maximul dintre valorile lor absolute.

PROGRAMUL BIII21

```
#include <stdio.h>
main() /* citeste doua numere intregi si afiseaza maximul dintre valorile lor absolute */
{
    int a,b,c,d;

    scanf("%d %d", &a,&b);
    printf("a=%d\b=b=%d\tabs(a)=%d\tabs(b)=%d\n", a,b,
          c = a<0 ? -a: a, d=b < 0 ? -b : b);
    printf("max(abs(a),abs(b))=%d\n",c > d ? c : d );
}
```

3.22 Să se scrie un program care citește valoarea variabilei x și afișează valoarea funcției $f(x)$ definită ca mai jos:

$$f(x) = \begin{cases} 3x^2 + 7x - 10 & \text{pentru } x < 0; \\ 2 & \text{pentru } x = 0; \\ 4x^2 - 8 & \text{pentru } x > 0. \end{cases}$$

PROGRAMUL BIII22

```
#include <stdio.h>
main() /* citeste pe x si afiseaza valoarea functiei f(x) definita astfel:
          3x^2 + 7x - 10 pentru x < 0;
          2           pentru x = 0;
          4x^2 - 8   pentru x > 0 */
{
    double x;

    scanf("%lf", &x);
    printf("x=%g\tf(x)=%g\n",x,
          x<0 ? 3*x*x+7*x-10 : x>0 ? 4*x*x-8 : 2);
}
```

3.2.14. Operatorul virgulă

Operatorul virgulă leagă două expresii în una singură conform formatului de mai jos:

$exp1,exp2$

Operatorul virgulă are cea mai mică prioritate dintre toți operatorii limbajului C. Prioritatea lui este imediat mai mică decit a operatorilor de atribuire.

Construcția de mai sus este o expresie. Valoarea și tipul acestei expresii coincide cu valoarea și tipul ultimei expresii, deci în cazul de față cu a lui $exp2$.

O construcție de formă:

$(exp1,exp2),exp3$

este corectă și ea este o nouă expresie. Într-adevar, $exp1, exp2$ fiind expresii, $(exp1,exp2)$ este un operand, deci totuși expresie, așa că după ea poate fi serisa o virgulă urmată de o alta expresie. Deoarece operatorul virgulă are prioritatea cea mai mică, rezultă că parantezele pot fi omise, deci expresia de mai sus se poate scrie astfel:

$exp1,exp2,exp3$

În general, o construcție de formă:

$exp1,exp2,...,expn$

unde $exp1,exp2,...,expn$ sunt expresii care o expresie a cărei valoare și tip coincide cu valoarea și tipul lui $expn$.

Într-o astfel de construcție, expresiile se evaluatează pe rînd, de la stînga la dreapta.

Operatorul virgulă se utilizează în situații în care într-un anumit punct al unui program în care este legal să folosim o expresie, este necesar să se realizeze un calcul complex exprimatării mai multe expresii.

Exerciții:

3.23 Să se scrie un program care citește doi intregi și afișează maximul dintre valorile lor absolute.

Acest exercițiu a fost rezolvat și în paragraful precedent (exercițiul 3.21.)

PROGRAMUL BIII23

```
#include <stdio.h>
main() /* citeste doi intregi si afiseaza maximul dintre valorile lor absolute */
{
    int a,b,c,d;

    scanf("%d%d", &a,&b);
    printf("a=%d\b=b=%d\tmax(abs(a),abs(b))=%d\n",
          a,b, ((c=a<0?-a:a),
                 (d=b<0?-b:b), (c > d) ? c : d));
}
```

Observație:

Expresia:

$(c=a<0?-a:a),(d=b<0?-b:b),(c>d)$

se compune din trei expresii care se evaluatează de la stînga la dreapta.

Prima atribuie lui c valoarea absolută a lui a , a doua atribuie lui d valoarea absolută a lui b , iar a treia testează relația $c>d$. Valoarea intregii expresii coincide cu 1 dacă $c>d$ și cu zero în caz contrar.

3.2.15. Alți operatori ai limbajului C

În limbajul C se utilizează și alți operatori și anume:

- operatorul * unar;
- și
- operatorii . și ~.

Operatorul * unar se utilizează pentru a face acces la conținutul unei zone de memorie definită prin adresa ei de început. Se obisnuiește să se spună că operatorul adresă (& unar) este *operator de referințiere*, iar operatorul * unar este *operator de dereferințiere*. Acest operator va fi studiat ulterior împreună cu datele de tip adresă (*pointer*). El are aceeași prioritate ca și ceilalți operatori unari ai limbajului C.

Operatorii . și ~ (acest simbol este compus din semnul "minus" urmat de semnul "mai mare") se utilizează pentru a se face acces la componentele unei structuri. Ei sunt de prioritate maximă, având aceeași prioritate cu parantezele. Ei vor fi studiați în capitolul cu privire la structuri.

3.2.16. Tabela cu prioritățile operatorilor limbajului C

În tabela de mai jos se indică operatorii limbajului C în ordinea descrescătoare a priorităților lor.

Operatorii din aceeași linie au aceeași prioritate. Ei se asociază de la stînga spre dreapta, exceptând operatorii unari, condiționali și de atribuire, care se asociază de la dreapta spre stînga.

()		->	(tip)	sizeof	!	~
+ (unar)	- (unar)	& (unar)	* (unar)	++ --		
* (binar)	/	%				
+ (binar)	- (binar)					
<<	>>					
<	<=	>=	>			
==	!=					
& (binar)						
^						
&&						
	:					
?						
=	<=	>=	+=	-*=	/=	%=
,						

Ulterior aceasta tabela va fi completată și cu alți operatori specifici limbajului C++.

4. INSTRUCȚIUNI

Am văzut că o funcție are structura:

an tet

corp

Corpul unei funcții conține, între acolade, o succesiune de declarații urmărite de o succesiune de instrucții:

```
{  
    declarații  
    instrucții  
}
```

Corpul poate conține numai instrucții sau numai declarații, sau se poate reduce la cele două acolade (acoladele vor fi totdeauna prezente).

Declarațiile permit utilizatorului să definiască date de diferite tipuri (predefinite sau definite de utilizator). De asemenea, datele pot fi inițializate cu ajutorul declarațiilor.

Prelucrarea datelor se realizează cu ajutorul *instrucțiunilor*.

Ordinea în care se execută instrucțiunile unui program definește astfel numita *structură de control a programului*.

Cea mai simplă structură de control este structura *secvențială*. O astfel de structură de control se compune dintr-o succesiune de instrucții care se execută una după alta, în ordinea în care sunt serise în program.

De obicei, la descrierea unui proces de calcul este nevoie să se utilizeze și alte tipuri de structuri. Astfel, C. Bohm și G. Jacopini au arătat în lucrarea [1], că pentru exprimarea proceselor de calcul sunt suficiente trei structuri de control și anume:

- structura secvențială;
- structura alternativă;

și

- structura repetitiva (ciclică) condiționată anterior.

Acest rezultat s-a aflat la baza ideii care a condus în anii '70 la conceptul de *programare structurată*. Acest concept a fost dezvoltat de E.W. Dijkstra în lucrările sale, iar ulterior și de alți specialiști, ca de exemplu, N. Wirth și C.A.R. Hoare. El reprezintă un *stil* în programare care se impune și în prezent. Un efect imediat al programării structurate este ridicarea productivității în programare și creșterea fiabilității programelor.

Prin *program structurat* înțelegem un program care are o structură de control realizată numai cu ajutorul celor trei structuri amintite mai sus.

Ulterior s-au admis încă două structuri și anume:

- structura selectivă;
- și
- structura repetitivă (ciclică) condiționată posterior.

Introducerea acestor structuri permite o flexibilitate mai mare în programare. În acest fel, programarea structurată reprezintă un stil în programare care contribuie la realizarea de programe care au o structură clară și care pot fi ușor depanate și întreținute.

Limbajul C a fost prevăzut cu instrucțiuni menite să permită realizarea simplă a structurilor proprii programării structurate. Structura secvențială se realizează cu ajutorul instrucțiunii *composte*, structura alternativă cu ajutorul instrucțiunii *if*, structura repetitivă condiționată anterior, prin intermediul instrucțiunilor *while* și *for*, structura selectivă se realizează cu ajutorul instrucțiunii *switch*, iar structura repetitivă condiționată posterior cu ajutorul instrucțiunii *do-while*.

Limbajul C are și alte instrucțiuni care reprezintă elemente de bază în construirea structurilor amintite mai sus. Astfel de instrucțiuni sunt: instrucțiunea *expresie* și instrucțiunea *vidă*.

Alte instrucțiuni prezente în limbaj asigură o flexibilitate mare în programare. Acestea sunt instrucțiunile:

return, break, continue și goto.

Mai jos se descriu aceste instrucțiuni și se dau exemple simple de utilizare.

4.1. Instrucțiunea vidă

Instrucțiunea vidă se reduce la caracterul punct și virgulă (;). Ea nu are nici un efect.

Instrucțiunea vidă se utilizează în construcții în care se cere prezența unei instrucțiuni, dar nu trebuie să se execute nimic în punctul respectiv. Astfel de situații apar frecvent în cadrul structurii alternative și repetitive, aşa cum se va vedea în continuare.

4.2. Instrucțiunea expresie

Instrucțiunea expresie se obține scriind punct și virgulă după o expresie. Deci, instrucțiunea expresie are formatul:

expresie;

În cazul în care expresia din compunerea unei instrucțiuni expresie este o expresie de atribuire, se spune că instrucțiunea respectivă este o *instrucțiune de atribuire*.

Un alt caz utilizat frecvent este acela cind expresia este un operand ce reprezintă apelul unei funcții. În acest caz, instrucțiunea expresie este o

instrucțiune de apel a funcției respective.

Exemple:

1. `int x;`
...
`x=10;`
Este o instrucțiune de atribuire. Variabilei *x* i se atribuie valoarea 10.
2. `double y;`
...
`y=y+4; sau y+=4;`
Este o instrucțiune de atribuire, care mărește cu 4 valoarea lui *y*.
3. `putch(c-'a'+'A');`
Este o instrucțiune de apel a funcției *putch*.
4. `double a;`
...
`a++;`
Este o instrucțiune expresie, care mărește valoarea lui *a* cu unu.
5. `double a;`
...
`++a;`
Are același efect ca și instrucțiunea expresie din exemplul precedent.

Observație:

Nu orice expresie urmată de punct și virgulă formează o instrucțiune expresie efectivă. De exemplu, construcția:

a;

deși este o instrucțiune expresie, ea nu are nici un efect.

Exerciții:

- 4.1 Să se scrie un program care citește trei întregi și afișează maximul dintre ei.

PROGRAMUL BIV1

```
#include <stdio.h>
main() /* citește trei intregi și afișează maximul dintre ei */
{
    int a,b,c,d;
    scanf("%d %d %d", &a,&b,&c);
    d = a > b ? a : b; // d=max(a,b)
    printf("a=%d\tb=%d\tc=%d\tmax(a,b,c)=%d\n", a,b,c,d>c ? d: c);
}
```

- 4.2 Sa se scrie un program care citește valorile variabilelor a, b, c, d, x de tip *double* și afișează valoarea expresiei:

$$(a^*x^*x+b^*x+c)/(a^*x^*x^*x+b^*x+d)$$

dacă numitorul este diferit de zero și zero în caz contrar.

PROGRAMUL BIV2

```
#include <stdio.h>
main() /* citește pe a,b,c,d și x, calculează și afișează valoarea expresiei:
          (a*x*x+b*x+c)/(a*x*x*x+b*x+d)
          dacă numitorul este diferit de zero și zero în caz contrar */
{
    double a,b,c,d,x;
    double y,z,u;

    printf("a=");
    scanf("%lf",&a);
    printf("b=");
    scanf("%lf",&b);
    printf("c=");
    scanf("%lf",&c);
    printf("d=");
    scanf("%lf",&d);
    printf("x=");
    scanf("%lf",&x);
    y = a*x*x;
    z=b*x;
    u=y*x+z+d;
    printf("(a*x*x+b*x+c)/(a*x*x*x+b*x+d)=%g\n",
           u ? (y+z+c)/u : u);
}
```

4.3. Instrucțiunea compusă

Instrucțiunea compusă este o succesiune de instrucțiuni incluse între acolade, succesiune care poate fi precedată și de declarații:

```
{
    declarații
    instrucțiuni
}
```

Dacă declarațiile sunt prezente, atunci ele definesc variabile care sunt definite atât timp cît controlul programului se află la o instrucțiune din compunerea instrucțiunii compuse.

Exemplu:

Presupunem că într-un anumit punct al programului este necesar să se permute valorile variabilelor intregi a și b . Aceasta se poate realiza astfel:

```
{
    int t;
```

```
t=a;
a=b;
b=t;
}
```

Variabila t este definită de îndată ce controlul programului ajunge la prima instrucțiune din instrucțiunea compusă ($t=a$). După execuția ultimei instrucțiuni a instrucțiunii compuse, variabila t nu mai este definită.

Instrucțiunea compusă se utilizează unde este necesară prezența unei instrucțiuni dar procesul de calcul din punctul respectiv este mai complex și se exprimă prin mai multe instrucțiuni. În acest caz instrucțiunile respective se includ între acolade pentru a forma o instrucțiune compusă.

Acest procedeu de a forma o instrucțiune compusă din mai multe instrucțiuni, se utilizează frecvent în construirea structurilor alternative și ciclice.

4.4. Instrucțiunea if

Instrucțiunea *if* are următoarele formate:

```
format1
    if(expresie)
        instrucțiune
format2
    if(expresie)
        instrucțiune1
    else
        instrucțiune2
```

La întîlnirea instrucțiunii *if* întîi se evaluatează expresia din paranteze. Apoi, în cazul formatului 1, dacă expresia are valoarea diferită de zero (adică are valoarea adevărată), atunci se execută *instrucțiune*; altfel se trece în secvență la instrucțiunea următoare instrucțiunii *if*. În cazul formatului 2, dacă expresia are o valoare diferită de zero, atunci se execută *instrucțiune 1* și apoi se trece în secvență (adică la instrucțiunea aflată după *instrucțiune 2*); altfel se execută *instrucțiune 2*.

În mod normal, în ambele formate după execuția instrucțiunii *if* se ajunge la instrucțiunea următoare ei. Cu toate acestea, este posibilă și o altă situație cînd instrucțiunile din compunerea lui *if*, definesc ele însele un alt mod de continuare a execuției programului.

Deoarece o instrucțiune compusă este considerată ca fiind un caz particular de instrucțiune, rezultă că instrucțiunile din compunerea lui *if* pot fi instrucțiuni

compuse. De asemenea, instrucțiunile respective pot fi chiar instrucțiuni *if*. În acest caz se spune că instrucțiunile *if* sunt *imbricate*.

Exerciții:

4.3 Se dă funcția:

$$y=3*x*x+2*x-10 \text{ pentru } x>0$$

și

$$y=5*x+10 \text{ pentru } x\leq 0$$

Să se scrie un program care citește valoarea lui x , calculează și afișează valoarea lui y . Acest proces de calcul implică două alternative, în funcție de valoarea lui x :

$$\begin{aligned} &\text{dacă } x>0, \text{ atunci } y=3*x*x+2*x-10 \\ &\text{altfel (adică } x\leq 0), \text{ atunci } y=5*x+10. \end{aligned}$$

Aceasta se transcrie imediat, în limbajul C, folosind formatul 2 al instrucțiunii *if*:

```
if (x>0) /* x>0 este expresie din formatul 2 */
    y=3*x*x+2*x-10; /* instrucțiune 1 */
else
    y=5*x+10; /* instrucțiune 2 */
```

Același proces de calcul se poate realiza cu ajutorul expresiei de atribuire:

$$y=x>0?3*x*x+2*x-10:5*x+10;$$

care utilizează în partea dreaptă o expresie condițională. Aceasta este o scriere mai compactă, dar nu atât de evidentă, ca și instrucțiunea *if* de mai sus.

PROGRAMUL BIV3

```
#include <stdio.h>
main() /* citește pe x, calculeaza și afișeaza valoarea lui y definită astfel:
           y = 3*x*x + 2*x - 10   dacă x > 0
           y = 5*x + 10           dacă x ≤ 0 */
{
    float x,y;
    scanf ("%f", &x);
    if (x > 0)
        y= 3*x*x + 2*x -10;
    else
        y= 5*x + 10;
    printf ("x= %f\ny= %f\n", x,y);
}
```

4.4 Să se scrie un program care citește valoarea lui x , calculează și afișează valoarea lui y definită ca mai jos:

$y=4*x**3+5*x*x-2*x+1$	pentru $x<0$;
$y=100$	pentru $x=0$;
$y=2*x*x+8*x-1$	pentru $x>0$.

Acest proces de calcul implică următoarele alternative:

```
dacă x < 0 atunci y=4*x**3+5*x*x-2*x+1
altfel
    dacă x=0
        atunci y=100
    altfel y=2*x*x+8*x-1
```

Ei se transcrie imediat printr-o instrucțiune *if imbricată*:

```
if (x<0) /* x<0 este expresie din formatul 2 */
    y=4*x**3+5*x*x-2*x+1; // instrucțiune 1
else
// instrucțiune 2 este instrucțiune if
    if (x==0)
        y=100;
    else
        y=2*x*x+8*x-1;
```

Același proces de calcul se poate descrie prin instrucțiunea expresie de mai jos:

$$y=x<0?4*x**3+5*x*x-2*x+1:x==0?100:2*x*x+8*x-1;$$

PROGRAMUL BIV4

```
#include <stdio.h>
main() /* citește pe x, calculeaza și afișeaza pe y definit astfel:
           y = 4*x*x*x + 5*x*x - 2*x + 1   dacă x<0;
           y = 100                           dacă x=0;
           y = 2*x*x + 8*x - 1             altfel */
{
    float x,y,a;
    scanf ("%f", &x);
    a = x*x;
    if ( x < 0 )
        y = 4*x*a + 5*a - 2*x + 1;
    else
        if ( x == 0 )
            y = 100;
        else
            y = 2*a + 8*x - 1;
    printf ("x=%f\ny=%f\n", x,y);
}
```

4.5 Să se scrie un program care citește valorile variabilelor neintregi a și b , calculează rădăcina ecuației:

$$ax+b=0$$

și afișează rezultatul.

Procesul de calcul al rădăcinii ecuației de mai sus trebuie să verifice existența ei:

dacă a este diferit de zero

atunci $x = -b/a$

altfel

dacă $b=0$

atunci ecuația este nedeterminată;

altfel ecuația nu are soluție.

PROGRAMUL BIV5

```
#include <stdio.h>
main() /* citeste pe a și b, calculeaza și afișeaza rădăcina ecuației
          ax + b = 0 */
{
    double a,b;
    if (scanf ("%lf %lf", &a,&b) != 2)
        printf ("coeficienti eronati\n");
    else
        if ( a != 0 )
            printf ("a=%g\b=b=%g\tx=%g\n",a,b, -b/a );
        else
            if ( b== 0 )
                printf ("ecuație nedeterminata\n");
            else
                printf ("ecuația nu are soluție\n");
}
```

Observații:

1. Programul de față conține o instrucțiune *if* imbricată.
2. Pentru a mări claritatea programelor se obișnuiește să se decaleze spre dreapta (cu un tabulator) instrucțiunile din compunerea unei instrucțiuni *if*.
3. În acest program s-a realizat test relativ la valorile tastate pentru a și b . Dacă funcția *scanf* nu returnează valoarea 2, inseamnă că nu s-au tastat două numere la terminal. De aceea, într-o astfel de situație se afișează mesajul: "coeficienti eronati".
- 4.6 Să se scrie un program care citește coeficienții a, b, c, d, e, f , ai unui sistem de două ecuații liniare cu două necunoscute, determină și afișează soluția acestuia cind are o soluție unică.

Fie sistemul de ecuații liniare:

$$ax+by=c$$

$$dx+ey=f$$

Notăm cu \det determinantul coeficienților necunoscutelor. Acesta se

calculează cu relația:

$$\det = ae - bd$$

Notăm cu \det_1 determinantul obținut din \det prin înlocuirea primei coloane cu coloana termenului liber și cu \det_2 determinantul obținut din \det prin înlocuirea coloanei a două cu coloana termenului liber. Atunci:

$$\det_1 = ce - bf \text{ și } \det_2 = af - cd.$$

Procesul de calcul al valorilor lui x și y se desfășoară conform pașilor de mai jos:

1. Se citesc valorile coeficienților a, b, c, d, e, f .
2. $\det = ae - bd$.
3. Dacă $\det = 0$, atunci sistemul nu are soluție unică (este nedeterminat sau incompatibil). Se afișează un mesaj corespunzător și se întrerupe execuția programului. Altfel se continuă cu punctul 4.
4. $\det_1 = ce - bf$.
5. $\det_2 = af - cd$.
6. $x = \det_1 / \det$.
7. $y = \det_2 / \det$.
8. Se afișează valorile lui x și y .

PROGRAMUL BIV6

```
#include <stdio.h>
main() /* citeste pe a,b,c,d,e,f, determina și afișeaza solutia sistemului:
          ax + by = c
          dx + ey = f
          in cazul in care are solutie unica */
{
    double a,b,c,d,e,f,x,y,det,det1,det2;
    if (scanf ("%lf %lf %lf %lf %lf %lf", &a,&b,&c,&d,&e,&f) != 6)
        printf ("coeficienti eronati\n");
    else
        if ((det = a*e - b*d) == 0)
            printf ("sistemul are determinantul nul\n");
        else
        {
            det1 = c*e - b*f;
            det2 = a*f - c*d;
            x = det1/det;
            y = det2/det;
            printf ("x=%g\ty=%g\n",x,y);
        }
}
```

Observații:

1. Expresia
 $(1) (det=a^*e-b^*d)==0$

se evaluatează astfel:

- a. Se calculează
 a^*e-b^*d
 și valoarea respectivă se atribuie lui *det*.

- b. Se compară valoarea atribuită lui *det* cu zero.

Dacă valoarea lui *det* este zero, atunci expresia (1) are valoarea *adevărat*; în caz contrar, expresia (1) are valoarea zero, adică *fals*.

2. Parantezele din expresia (1) sunt obligatorii.

În lipsa lor, expresia (1) devine:

$$(2) \det = a^*e - b^*d == 0.$$

Aceasta se evaluatează astfel:

- a. Se calculează valoarea expresiei:
 $a^*e-b^*d;$
- b. Deoarece operatorul $==$ este mai prioritar decit $=$, se compară valoarea expresiei respective cu zero.
 Dacă expresia respectivă are valoarea zero, atunci expresia
 $(3) a^*e-b^*d==0$
 are valoarea 1 (*adevărat*), altfel are valoarea zero (*fals*).
- c. Se atribuie lui *det* valoarea expresiei 3, adică 0 sau 1.

- 4.7 Să se scrie un program care citește valorile variabilelor *a*, *b*, *c*, calculează și afișează rădăcinile ecuației de gradul 2:

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

Pasii procesului de calcul sunt:

1. Se citesc valorile variabilelor *a*, *b*, *c*.
2. Dacă $a=b=c=0$, ecuația este nedeterminată.
 Se afișează un mesaj corespunzător și se intrerupe execuția programului.
3. Dacă $a=b=0$ și *c* este diferit de zero, ecuația nu are soluție.
 Se afișează un mesaj corespunzător și se intrerupe execuția programului.
4. Dacă $a=0$ și *b* este diferit de zero, ecuația se reduce la o ecuație de gradul întii a cărei soluție este:
 $x=-c/b;$
 se afișează un mesaj corespunzător, soluția *x* și se intrerupe execuția programului.
5. Dacă *a* este diferit de zero, se calculează:
 $\text{delta}=b*b-4*a*c$
 și
 $d=2*a.$
6. Dacă *delta* > 0, atunci *delta=sqrt(delta)*; (*sqrt* este funcția de bibliotecă prin

care se calculează rădăcina pătrată).

Se determină și se afișează rădăcinile:

$$x1=(-b+\text{delta})/d;$$

$$x2=(-b-\text{delta})/d;$$

apoi se intrerupe execuția programului;

7. Dacă *delta*=0, atunci se determină și se afișează rădăcina dublă:
 $x=-b/d;$

se intrerupe execuția programului.

8. Altfel, ecuația are rădăcini complexe conjugate.
 Se calculează:
 $\text{delta}=\text{sqrt}(-\text{delta}).$

Se determină și se afișează rădăcinile:

$$x1=-b/d+i*\text{delta}/d;$$

$$x2=-b/d-i*\text{delta}/d.$$

Se intrerupe execuția programului.

PROGRAMUL BIV7

```
#include <stdio.h>
#include <math.h> // fisierul contine prototip pentru sqrt

main() /* cîtește pe a,b,c, calculeaza si afiseaza radacinile ecuatiei:
          a*x*x + b*x + c = 0 */
{
    double a,b,c,d,delta;

    printf("coefficientul lui x patrat:");
    if(scanf("%lf",&a) != 1)
        printf("coefficientul lui x patrat eronat\n");
    else { /* 1 */
        printf("coefficientul lui x:");
        if(scanf("%lf",&b) != 1)
            printf("coefficientul lui x eronat\n");
        else { /* 2 */
            printf("termenul liber:");
            if(scanf("%lf", &c) != 1)
                printf("termenul liber eronat\n");
            else { /* 3 */
                /* afiseaza coefficientii cititi */
                printf("a=%g\\tb=%g\\tc=%g\\n",a,b,c);
                if( a== 0 && b == 0 && c == 0 )
                    printf("ecuatie nedeterminata\\n");
                else
                    if( a == 0 )
                        printf("ecuatie nu are solutie\\n");
                    else
                        if ( a == 0 ) {
                            printf("ecuatie de gradul 1\\n");
                            printf("x=%g\\n",-c/b);
                        }
            }
        }
    }
}
```

```

else{ /*4*//*a!=0*/
    delta = b*b - 4*a*c;
    d = 2*a;
    if( delta > 0 ) {
        printf("ecuatie are 2\n");
        radacini_reale_si_distincte\n");
        delta = sqrt(delta);
        printf("x1=%g(x2)=%g\n",
            (-b + delta)/d, (-b - delta)/d);
    }
    else if( delta == 0 ) {
        printf("ecuatie are\n");
        radacina_dubla\n");
        printf("x=%g\n", -b/d);
    }
    else /*5*/
        printf("radacini complexe\n");
        delta = sqrt(-delta)/d;
        d = -b/d;
        printf("x1=%g+i(%g)\n",d,delta);
        printf("x2=%g-i(%g)\n",d,delta);
    }/*sfarsit*5*/
}/*sfarsit*4*/
}/*sfarsit*3*/
}/*sfarsit*2*/
}/*sfarsit*1*/
}/*sfarsit program*/

```

Observație:

Programul de față utilizează instrucțiunea *if* cu nivel de imbricare relativ mare. Programele care folosesc instrucțiuni *if* cu niveluri mari de imbricare sunt greu de urmarit. Adesea se uită închiderea tuturor acoladelor. De aceea se recomandă reducerea pe cît posibil a nivelurilor de imbricare ale instrucțiunilor *if*. În multe cazuri, nivelul de imbricare al unei instrucțiuni *if* se poate reduce folosind funcția *exit* (vezi paragraful urmator).

4.5. Funcția standard *exit*

Funcția *exit* are prototipul:

void exit(int cod)

și el se află în fișierele de tip *h*:

stdlib.h

și

process.h

La apelul acestei funcții au loc următoarele acțiuni:

- se videază zonele tampon (bufferele) ale fișierelor deschise în scriere;
- se închid toate fișierele deschise;
- se întrerupe execuția programului.

Parametrul acestei funcții definește starea programului la momentul apelului.

Valoarea zero definește o terminare normală a execuției programului, iar o valoare diferită de zero semnalizează prezența unei erori (terminarea anormală a execuției programului).

În concluzie, putem apela funcția *exit* pentru a termina execuția unui program, indiferent de faptul că acesta se termină normal sau din cauza unei erori.

Exerciții:

- 4.8 Să se modifice programul din exercițiul 4.7 în aşa fel încât el să nu mai conțină instrucțiuni *if* imbricate.

PROGRAMUL BIV8

```

#include <stdio.h>
#include <math.h> // fisierul contine prototip pentru sqrt
#include <stdlib.h> // fisierul contine prototip pentru exit

main() /* citeste pe a,b,c, calculeaza si afiseaza radacinile ecuatiei:
           a*x*x + b*x + c = 0 */
{
    double a,b,c,d,delta;

    printf("coeficientul lui x patrat:");
    if(scanf("%lf",&a) != 1) {
        printf("coeficientul lui x patrat eronat\n");
        exit(1);/* terminare la eroare */
    }
    printf("coeficientul lui x:");
    if(scanf("%lf",&b) != 1) {
        printf("coeficientul lui x eronat\n");
        exit(1);/* terminare la eroare */
    }
    printf("termenul liber:");
    if(scanf("%lf", &c) != 1) {
        printf("termenul liber eronat\n");
        exit(1);/* terminare la eroare */
    }

    /* afiseaza coeficientii cititi */
    printf("a=%g\tb=%g\etc=%g\n",a,b,c);
    if( a == 0 && b == 0 && c == 0) {
        printf("ecuatie nedeterminata\n");
        exit(0);/* terminare fara eroare */
    }
    if( a == 0 && b == 0){

```

```

printf("ecuatie nu are solutie\n");
exit(0);
}
if ( a == 0 ) {
    printf("ecuatie de gradul 1\n");
    printf("x=%g\n", -c/b);
    exit(0);
}
delta = b*b - 4*a*c;
d = 2*a;
if ( delta > 0 ) {
    printf("ecuatie are 2 radacini reale si distincte\n");
    delta = sqrt(delta);
    printf("x1=%g\tx2=%g\n", (-b + delta)/d, (-b - delta)/d);
    exit(0);
}
if ( delta == 0 ) {
    printf("ecuatie are radacina dubla\n");
    printf("x=%g\n", -b/d);
    exit(0);
}
printf("radacini complexe\n");
delta = sqrt(-delta)/d;
d = -b/d;
printf("x1=%g+i(%g)\n", d, delta);
printf("x2=%g-i(%g)\n", d, delta);
}

```

4.6. Instrucțiunea while

Instrucțiunea *while* are formatul:

```
while(expresie)
instrucțiune
```

Primul rind din acest format constituie antetul instrucțiunii *while*, iar *instrucțiune* este corpul ei.

La intînlirea acestei instrucțiuni intii se evaluatează expresia din paranteze. Dacă ea are valoarea *adevărat* (este diferită de zero), atunci se execută *instrucțiune*. Apoi se revine la punctul în care se evaluatează din nou valoarea expresiei din paranteze. În felul acesta, corpul ciclului se execută atât timp cât expresia din antetul ei este diferită de zero. În momentul în care *expresie* are valoarea zero, se trece la instrucțiunea următoare instrucțiunii *while*.

Corpul instrucțiunii *while* poate să nu se execute niciodată. Într-adevăr dacă *expresie* are valoarea zero de la început, atunci se trece la instrucțiunea următoare instrucțiunii *while* fără a executa niciodată corpul instrucțiunii respective.

Corpul instrucțiunii *while* este o singură instrucțiune, care poate fi compusă. În felul acesta, avem posibilitatea să executăm repetat mai multe instrucțiuni grupate într-o instrucțiune compusă.

Corpul instrucțiunii *while* poate fi o altă instrucțiune *while* sau să fie o instrucțiune compusă care să conțină instrucțiuni *while*. În acest caz se spune că instrucțiunile *while* respective sunt imbricate.

Mentionăm că instrucțiunea din corpul unei instrucțiuni *while* poate să definească un alt mod de execuție a instrucțiunii *while* decit cel indicat mai sus. Astfel, ea poate realiza terminarea instrucțiunii *while* fără a se mai ajunge la evaluarea expresiei din antetul ei. De exemplu, dacă în corpul unei instrucțiuni *while* se apelează funcția *exit*, atunci se va termina execuția ciclului *while*, deoarece se întrerupe chiar execuția programului.

Despre instrucțiunea *while* se spune adesea că este o instrucțiune *ciclică*.

Amintim că ea definește o structură repetitivă condiționată anterior.

Exerciții:

- 4.9 Să se scrie un program care calculează și afișează valoarea polinomului $p(x)=3x^3-7x-10$ pentru $x=1, 2, \dots, 10$

Programul de față realizează un proces de calcul care constă din evaluarea și afișarea valorii expresiei:

$$3x^3-7x-10$$

pentru cele 10 valori ale lui *x*. Pentru o valoare a lui *x*, acest lucru se realizează prin intermediul instrucțiunii de apel:

```
printf("x=%d\t p(x)=%d\n", x, 3*x*x*x-7*x-10);
```

Rezultă că, această instrucțiune trebuie să se execute repetat pentru $x=1, 2, \dots, 10$. În acest scop, inițial vom atribui lui *x* valoarea 1:

```
x=1;
```

Apoi, vom utiliza o instrucțiune *while* care să permită execuția repetată a instrucțiunii de apel a funcției *printf* de mai sus.

Deoarece lui *x* i s-a atribuit deja valoarea 1, rezultă că la prima execuție a acestei instrucțiuni se calculează *p(1)*. Pentru ca la a doua execuție a ei să se calculeze *p(2)*, este necesar să se mărească valoarea lui *x* cu 1. Aceasta se poate realiza folosind instrucțiunea expresie

```
x++;
```

Cu alte cuvinte, este suficient să se execute repetat secvența de instrucțiuni:

```
printf("x=%d\t p(x)=%d\n", x, 3*x*x*x-7*x-10);
x++;
```

x având inițial valoarea 1. Această secvență formează corpul ciclului. De aceea, ea va fi inclusă între acolade pentru a forma o singură instrucțiune.

Deoarece ultima execuție a sevenței de mai sus are loc cind $x=10$, rezultă că ea se va executa repetat atât timp cit $x \leq 10$. De aceea, antetul ciclului *while* va fi:

```
while(x<=10)
```

În concluzie, procesul de calcul ce urmează a fi realizat îl putem descrie astfel:

1. $x=1$
2. Atât timp cit $x \leq 10$ se execută:
 - 2.1. Afisează pe x și $p(x)$.
 - 2.2. Incrementează pe x .

PROGRAMUL BIV9

```
#include <stdio.h>
main() /* afiseaza valorile polinomului
          p(x)=3*x*x - 7*x - 10 pentru x = 1,2,...,10 */
{
    int x;

    x = 1;
    while( x <= 10 ) {
        printf("x=%d\t p(x)=%d\n", x, 3*x*x - 7*x - 10 );
        x++;
    }/* sfîrșit while */
}/* sfîrșit main */
```

Observație:

Limbajul C permite scrierea de programe compacte. De exemplu, sevența de mai sus poate fi compactată înlocuind expresia

$x \leq 10$

cu

$++x \leq 10$

În felul acesta se elimină instrucțiunea expresie

$x++;$

din corpul ciclului.

Tinind seama de ordinea operațiilor, expresia

$++x \leq 10$

se evaluează astfel:

1. Se incrementează x .
2. Se compară valoarea incrementată a lui x cu 10.

Dacă aceasta nu-l depășește pe 10, atunci se execută corpul ciclului, care acum constă numai din apelul funcției *printf*.

În caz contrar se încheie execuția ciclului *while* și prin aceasta se termină și execuția programului.

În acest caz se va schimba atribuirea

$x=1$

cu

$x=0$

deoarece execuția corpului ciclului începe după incrementarea lui x . Cu alte cuvinte, programul BIV9 poate fi rescris astfel:

```
...
x=0;
while(++x<=10)
    printf("x=%d\t p(x)=%d\n", x, 3*x*x - 7*x - 10 );
} /* sfîrșit main */
```

Menționăm că expresia

$++x \leq 10$

utilizată în antetul ciclului nu realizează același lucru.

Intr-adevăr, în cazul expresiei

$++x \leq 10$

atât comparația, cit și corpul ciclului *while* se execută cu valoarea incrementată a lui x .

În cazul expresiei

$x++ \leq 10$

comparația se face cu valoarea neincrementată a lui x , iar corpul ciclului *while* se execută cu valoarea incrementată a lui x . De aceea, dacă $x=10$, atunci

$++x \leq 10$

este falsă, deci corpul ciclului nu se mai execută, în schimb

$x++ \leq 10$

are valoarea adevărată și se va executa corpul ciclului și pentru $x=11$. De aceea, se va obține același lucru dacă în antetul ciclului vom folosi expresia

$x++<10$

4.10 Să se scrie un program care tabelizează valorile funcției $\sin(x)$ cu pasul de un grad sexagesimal, x aparținând intervalului $[0,360]$.

Funcția *sinus*, poate fi apelată ca funcție standard și are prototipul
double sin(double x);

in fisierul *math.h*.

Parametrul *x* este in radiani.

Dacă *a* este un unghi in grade sexagesimale, atunci el se reprezintă in radiani dacă se inmulțește cu factorul PI/180 (PI=3.14159265...):

$$a * \text{PI}/180$$

Tabelarea funcției *sinus* se va realiza conform pașilor de mai jos:

1. $f = \text{PI}/180$
2. $x = 0$
3. Atît timp cît $x \leq 359$, se execută:
 - 3.1. Calculează și afișează $\sin(x * f)$.
 - 3.2. Incrementează pe *x*.

Pasul 3 de mai sus este analog cu pasul 2 din exemplul precedent. De aceea, el se realizează printr-un ciclu *while* asemănător. La realizarea lui vom face compactarea indicată în observația de la exemplul respectiv.

PROGRAMUL BIV10

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

#define PI 3.14159265358979

main() /* calculeaza si afiseaza valorile functiei sin(x) din grad
         in grad sexagesimal, x apartinind intervalului [0,359] */
{
    int x;
    double f;

    f = PI/180.0;
    x = -1;
    while(++x <= 359)
        printf("sin(%d)=% .16f\n", x, sin(x*f));
}
```

Observație:

Valorile funcției *sinus* se afișează cite una pe o linie. De aceea, programul afișează 360 de linii. Utilizatorul are posibilitatea de a vedea pe ecran, în fereastra utilizator (tastind <Alt>-F5), numai ultimele 25 de valori. Aceasta din cauză că un ecran, în mod text, se compune din 25 de linii a 80 de coloane.

În astfel de situații se pune problema blocării execuției unui program după ce acesta a afișat un ecran de rezultate. După analiza datelor de pe ecran, utilizatorul urmează să deblocheze execuția programului acționind o tastă oarecare.

Un mod simplu de a bloca execuția unui program este acela de a apela funcția *getch* în momentul în care se dorește să se analizeze datele afișate în fereastra utilizator.

La apelul funcției *getch* programul se blochează deoarece se așteaptă acționarea unei taste. Totodată se afișează automat fereastra utilizator. La acționarea unei taste oarecare, programul se deblochează.

În exemplul de față vom apela funcția *getch* ori de cîte ori programul a afișat 23 de valori ale funcției sinus. Aceste valori ocupă 23 de linii ale ecranului. Pe linia 24 se va afișa textul:

pentru a continua actionati o tasta

Funcția *getch* se apelează în corpul lui *while* de indată ce sunt afișate 23 de valori. În acest scop utilizăm valorile lui *x* care cresc de la 0 la 359. Ecranul trebuie blocat după apelul funcției *printf* și în momentul în care *x+1* are valorile:

23, 46, 69,...

adică atunci cînd *x+1* este multiplu de 23.

Aceasta se exprimă prin expresia:

$$(x+1) \% 23 == 0$$

Deci funcția *getch* se apelează cînd expresia de mai sus este *adevărată*. Înainte de a apela funcția *getch* va trebui apelată funcția *printf* pentru a afișa, pe linia 24, textul:

pentru a continua actionati o tasta

Tinind seama de aceste observații, modificăm programul de față ca mai jos.

PROGRAMUL BIV10A

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

#define PI 3.14159265358979

main() /* calculeaza si afiseaza valorile functiei sin(x) din grad
         in grad sexagesimal, x apartinind intervalului [0,359] */
{
    int x;
    double f;

    f = PI/180.0;
    x = -1;
    while(++x <= 359) {
        printf("sin(%d)=% .16f\n", x, sin(x*f));
        if((x+1)%23 == 0) {
            printf("pentru a continua actionati o tasta\n");
            getch();
        }
    } /* sfîrșit while */
} /* sfîrșit main*/
```

- 4.11 Să se scrie un program care citește un sir de intregi separați prin caractere albe și afișează suma lor. După ultimul număr se va tăsi un caracter alb urmat de un caracter nenumeric (de exemplu sfîrșitul de fișier: <Ctrl>-z, o literă etc), iar după acesta se va actiona tasta *Enter*.

Programul de față are o parte care se executa repetat:

- Citește construcția curentă de la intrare.
 - Dacă la intrare s-a aflat un intreg, atunci acesta se aduna la valoarea unei variabile *s* și se revine la punctul a.
- Altfel se interupe citirea.

De aici rezultă că inițial s trebuie să aibă valoarea zero.

Un intreg se citește apelind funcția *scanf*. Dacă notăm cu *i* variabila căreia i se atribuie intregul citit, atunci la citirea unui intreg, expresia:

```
scanf ("%d", &i) == 1
```

are valoarea *adevărată*. Deci valoarea lui *i* trebuie adunată la *s* atât timp cât expresia de mai sus este *adevarată*. Deci expresia de mai sus se utilizează în antetul instrucțiunii *while*, iar corpul ciclului se compune din instrucțiunea:

```
s = s + i;
```

PROGRAMUL BIV11

```
#include <stdio.h>
main() /* citește un sir de intregi separati prin caractere albe si afiseaza suma lor */
{
    int i, s;

    i = 0;
    while (scanf ("%d", &i) == 1)
        s += i;
    printf ("suma = %d\n", s);
}
```

- 4.12 Să se scrie un program care citește un intreg *n*, calculează și afișează pe *n!*

Avem:

$n! = 1 \cdot 2 \cdot 3 \cdots \cdot (n-1) \cdot n$, pentru $n \geq 0$ și $0!$ se consideră egal cu 1.

Acest calcul implica un proces ciclic.

Intr-adevăr, fie

$f = 1$ și $i = 2$

Atunci:

$f * i = 1 * 2 = 2!$

Considerăm instrucțiunea de atribuire:

$f = f * i;$

Cum $f * i = 2!$, rezultă că $f = 2!$. Putem folosi aceeași instrucțiune pentru a calcula pe $3!$.

Intr-adevăr

$3! = 2! * 3 = f * 3$

deci expresia: $f * i = 3!$ dacă, în prealabil, *i* se mărește cu o unitate. Rezultă că alături de instrucțiunea de mai sus, este necesar să considerăm și instrucțiunea de incrementare a lui *i*:

- $f = f * i;$
- $i = i + 1;$

Această secvență se execută repetat, întâi *a* apoi *b* și apoi din nou *a* și apoi *b* și aşa mai departe. Lui *f* se atribuie succesiv valorile:

$2!, 3!, 4!, \dots$

Pentru a calcula pe $n!$ ($n \geq 1$), secvența de instrucțiuni *a-b* trebuie să se execute repetat atât timp cât *i* nu-l depășește pe *n*, adică atât timp cât expresia:

$i \leq n$

este *adevărată*.

Acest proces de calcul se realizează simplu prin instrucțiunea *while* de mai jos:

```
while (i <= n) {
    f = f * i;
    i++;
}
```

Instrucțiunea compusă poate fi compactată astfel:

```
f = f * i++;
```

Intr-adevăr, operatorul de incrementare se aplică postfixat, deci *f* se inmulțește cu valoarea neincrementată a lui *i*, ca și în cazul instrucțiunii compuse inițiale.

PROGRAMUL BIV12

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

main() /* citește pe n din intervalul [1,170], calculează și afișează pe n! */
{
    int n, i;
    double f;

    printf ("valoarea lui n:");
    if (scanf ("%d", &n) != 1) {

```

```

    printf("nu s-a tastat un intreg\n");
    exit(1);
}
if( n < 0 || n > 170 ) {
    printf("n nu apartine intervalului [0,170]\n");
    exit(1);
}
f = 1.0;
i = 2;
while( i <= n )
    f = f*i++;
printf("n=%d\tn!=%g\n",n,f);
}

```

- 4.13 Să se scrie un program care citește componentele a doi vectori x și y , calculează și afișează valoarea produsului lor scalar.

Dacă vectorul x are componente:

x_1, x_2, \dots, x_n

iar y :

y_1, y_2, \dots, y_n

atunci produsul lor scalar se definește prin suma de produse:

$$(x,y)=x_1*y_1+x_2*y_2+\dots+x_n*y_n$$

Componentele celor doi vectori se tastează în următoarea ordine:

$x_1 \ y_1$

$x_2 \ y_2$

\dots

$x_n \ y_n$

Pe rindul care urmează după ultima perche se poate tasta o literă, un caracter special care nu intră în compunerea unui număr sau sfîrșitul de fișier.

PROGRAMUL BIV13

```

#include <stdio.h>
main() /* citeste componentele vectorilor x si y, calculeaza si
         afiseaza valoarea produsului lor scalar */
{
    double x,y,prodscal;
    int i;

    prodscal = 0.0;
    printf("componentele lui x si y\n");
    i = 0;
    printf("x[%d]\ty[%d]:", i+1,i+1);
    while(scanf("%lf %lf", &x,&y) == 2) {
        prodscal += x*y;
        i++;
    }
}

```

```

    printf("x[%d]\ty[%d]:", i+1,i+1);
}
printf("numarul componentelor vectorilor=%d\n",i);
printf("produsul scalar=%g\n",prodscal);
}

```

- 4.14 Să se scrie un program care citește fără ecou caractere imprimabile și le afișează ca și caractere minus. Programul se întrerupe la tastarea unui caracter care nu aparține codului ASCII.

La realizarea programului vom folosi funcțiile *getch* și *putch*.

Funcția *getch* returnează codul ASCII al caracterului citit sau zero dacă se citește un caracter care nu aparține codului ASCII. Expresia

```
c=getch()
```

atribuie variabilei *c* codul ASCII al caracterului citit. Se vor recodifica prin minus numai caracterele imprimabile, adică de cod ASCII cel puțin egal cu 32 și cel mult 126. Restul caracterelor se neglijă. Dacă *c* are ca valoare codul ASCII al caracterului citit, atunci instrucția

```
if(c>=32&&c<=126) putch('+'');
```

permete afișarea caracterului respectiv ca și caracter minus, dacă acesta este un caracter imprimabil; altfel el este neglijat. Această instrucție se execută repetat, cît timp se tastează un caracter care aparține codului ASCII. De aceea, ea va forma corpul unui ciclu *while*.

În antetul acestui ciclu vom folosi expresia

```
c=getch()
```

care este diferita de zero (adevarata) pentru caracterele codului ASCII și zero (falsă) pentru restul caracterelor.

PROGRAMUL BIV14

```

#include <conio.h>
main() /* citeste caractere imprimabile fara ecou si le afiseaza ca si caracter minus */
{
    int c;

    while ( c = getch() )
        if( c >= 32 && c <= 126 ) putch('+'');
}

```

- 4.15 Să se scrie un program care apelează funcția *putch* cu valorile codului ASCII extins.

Codul ASCII extins cuprinde valorile intregi din intervalul [0,255].

Pe o linie se va afișa valoarea codului urmată de imaginea corespunzătoare

afişată prin funcţia *putch*. Ecranul se blochează după 23 de linii afişate sub forma indicată mai sus şi după ce pe linia 24 se afişează mesajul:

pentru a continua actionati o tasta

După acŃionarea unei taste ecranul se deblocheaza pentru a se afişa un nou set de imagini şi aşa mai departe.

PROGRAMUL BIV15

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

main() /* apeleaza functia putch cu valorile codului ASCII extins */
{
    int c;

    c = -1;
    while ( ++c <= 255 ) {
        printf("cod:%4d\t",c);
        putch(c);
        putchar('\n');
        if(( c+1 ) % 22 == 0) {
            printf("pentru a continua actionati o tasta\n");
            getch();
        }
    }
}
```

4.16 Să se scrie un program care citeşte rezultatele unor măsurători:

x_1, x_2, \dots, x_n

calculează şi afiŞează:

- media aritmetică a măsurătorilor;
 - media geometrică a lor;
- şi
- abaterea pătratică.

Dacă notăm cu *meda* media aritmetică a măsurătorilor, atunci abaterea pătratică este rădăcina patrată din expresia:

$$[(x_1-\text{meda})^2+(x_2-\text{meda})^2+\dots+(x_n-\text{meda})^2]/N$$

unde:

N - Este n sau $n-1$.

Numitorul se ia egal cu $n-1$ cind n este relativ mic. De obicei:

$$N=n-1 \text{ pentru } n < 30$$

şi

$$N=n \text{ pentru } n \geq 30$$

$$\text{Evident } n \geq 2.$$

Deoarece pentru calculul abaterii patratice este nevoie de măsurătorile x_1, x_2, \dots, x_n , cit şi de media lor aritmetică, programul va trebui să păstreze măsurătorile respective într-un tablou unidimensional, pe care îl notăm cu *x*.

Vom presupune ca există cel mult 500 de măsurători ($n \leq 500$).

Media geometrică se calculează prin extragerea rădăcinii de ordinul n din produsul celor n măsurători pozitive dintre cele n măsurători citite.

Setul de măsurători sunt date flotante în simplă precizie. Setul de măsurători se consideră terminat dacă se tastează un caracter care nu intră în compunerea unei date numerice.

Programul listează măsurătorile citite, cîte 5 pe un rind. De asemenea, programul se blochează după afiŞarea a 23 de linii cu măsurători.

PROGRAMUL BIV16

```
#define MAXMAS 500
#define LIMNUMUNIT 30
#define APEL printf("masuratoarea a %d-a:", n+1)

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

main() /* citeşte un set de măsurători, calculează şi afiŞează
         media lor aritmetică, geometrică şi abaterea patratice */
{
    float x[MAXMAS];
    double meda, medg, ap;
    int n, ng, i;

    n=ng=0;
    meda=0.0;
    medg=1.0;
    APEL;

    /* - se citeşte măsurătorile;
       - se inscriează;
       - se înmulţesc cele pozitive
    */
    while (scanf("%f", &x[n]) == 1) {
        meda += x[n];
        if( x[n] > 0 ) {
            medg *= x[n];
            ng++;
        }
        n++;
    }
    APEL;
}

if( n < 1 ) {
    printf("nu s-a tastat nici o masuratoare\n");
    exit(1);
}
```

```

meda /= n;

/* calculeaza media geometrica */
if( ng < 2 ) {
    printf("nu exista nici 2 masuratori pozitive\n");
    medg = -1.0;
}
else
    medg = pow(medg, 1.0/ng);
if( n > 1 ) {
    /* calculeaza abaterea patratica */
    ap = 0.0;
    i = 0;
    while ( i < n ) {
        ap += (x[i] - meda)*(x[i] - meda);
        i++;
    }
    if( n < LIMNUMIT)
        ap /= n-1;
    else
        ap /= n;
}
else {
    printf("nu exista nici 2 masuratori\n");
    ap = -1;
}

/* listeaza masuratorile, cite 5 pe un rind */
i = 0;
while(i < n) {
    printf("x[%d]=%g%c", i+1, x[i], i%5==4 || i==n-1 ? '\n' : ' ');
    i++;
    if(i%(5*23) == 0 ) {
        printf("pentru a continua actionati o tasta\n");
        getch();
    }
}

/* afisaza mediile si abaterea patratica */
printf("media aritmetica=%g\n", meda);
if(medg != -1.0)
    printf("media geometrica=%g\n", medg);
if( n > 1)
    printf("abaterea patratica=%g\n", sqrt(ap));
}

```

Observații:

- Pentru a afișa măsuratorile cite 5 pe un rind s-a apelat funcția *printf* folosind trei specifiicatori de format în parametrul de *control*:

"x[%d]=%g%c"

Parametrul corespunzător lui %d este i+1. Acesta definește numărul de ordine

al măsurătorii curente.

Parametrul corespunzător lui %g este x[i] și reprezintă chiar valoarea măsurătorii curente.

După specifiicatorul %g urmează specifiicatorul %c, a cărui parametru definește caracterul care se afișează după valoarea măsurătorii curente. Acest caracter poate fi spațiu (' ') sau caracterul de rind nou ('\n').

Pe primul rind se afișează valorile:

x[0] x[1] x[2] x[3] x[4]

deci, după măsuratorile de indice 0, 1, 2 și 3 se afișează un spațiu, iar după x[4] urmează caracterul de rind nou.

Analog, pe rindul al doilea se afișează valorile:

x[5] x[6] x[7] x[8] x[9]

deci, după măsuratorile de indice 5, 6, 7 și 8 se afișează un spațiu, iar după x[9] urmează caracterul de rind nou.

În general, pe un rind se afișează valorile:

x[5k] x[5k+1] x[5k+2] x[5k+3] x[5k+4]

Se observă că după o măsuratoare de indice 5k+4 urmează caracterul de rind nou, iar în rest urmează un spațiu.

Un indice *i* este de forma 5k+4, dacă restul împărțirii lui *i* la 5 este egal cu 4, adică, dacă expresia:

i%5==4

este adevarată.

De asemenea, după ultima măsuratoare se va afișa caracterul de rind nou, adică dacă

i=n-1

are valoarea *adevarată*.

Rezultă că afișarea caracterului de rind nou se va face atunci cind expresia

i%5==4||*i*=n-1

este *adevarată*. În rest se afișează un caracter spațiu. Această situație se reprezintă simplu prin expresia condițională:

i%5==4||*i*=n-1?"\n":" "

expresie ce corespunde specifiicatorului de format %c.

Acest exemplu demonstrează odată în plus facilitățile mari oferite de expresiile condiționale la compactarea programelor.

Această metodă de afișare a fost indicată de autorii limbajului C în lucrarea [2].

- Pentru a calcula radicalul de ordinul *ng* din *medg* se apelează funcția *pow*

cu parametrul *medg* și $1.0/\text{ng}$.

Funcția *pow* are prototipul:

```
double pow (double x,double y);
```

și returnează pe *x* la puterea *y*. În cazul de față $1.0/\text{ng}$ are tipul *double* deoarece 1.0 este o constantă de tip *double*. Menționăm că expresia $1/\text{ng}$ are valoarea zero (împărțire de întregi cu $\text{ng} > 1$) deci este necesară utilizarea constantei 1.0 . Apelul funcției *pow* în programul de mai sus returnează pe *medg* la puterea $1.0/\text{ng}$, adică radical indice *ng* din *medg*.

- 4.17 Sa se scrie un program care citește un cuvint (succesiune de caractere diferite de caracterele albe) și afișează prefixele și sufixele lui diferite de el însuși.

Fie cuvintul:

e t* f

Atunci prefixele acestui cuvint, diferite de el însuși sunt:

e
e t
e t*
e t*f

În mod analog, sufixele aceluiași cuvint sunt:

f
*f
t*f
t*t*f

Se presupune că un cuvint nu depășește 70 de caractere.

Programul avansează peste eventualele caractere albe care preced cuvintul de prelucrat. Apoi citește caracterele din compunerea cuvintului și le pastrează ca elemente ale tabloului *tab*.

Cuvintul se termină la:

- intilnirea unui caracter alb;
- intilnirea sfîrșitului de fișier;
- după ce s-au citit 70 de caractere care nu sunt albe.

Caracterele se citesc folosind macroul *getchar*. Acesta se apelează prin expresia de atribuire:

```
c=getchar()
```

Care atribuie lui *c* codul ASCII al caracterului citit. Pentru a avansa peste caracterele albe este necesar să stabilim dacă codul atribuit lui *c* este codul

spațiului, al tabulatorului sau al caracterului de rind nou. Expresia

```
(c==getchar())==' '
```

compară codul caracterului citit cu cel al spațiului și are valoarea *adevărat*, dacă s-a citit un spațiu și *fals* în caz contrar.

În mod analog, expresia

```
c=='\t'
```

este adevărată dacă s-a citit caracterul tabulator, iar expresia

```
c=='\n'
```

este adevărată dacă s-a citit caracterul *newline*. Rezultă că, dacă se citește un caracter alb, atunci este adevărată una dintre cele trei expresii de mai sus, ceea ce conduce la faptul că expresia:

```
(1) (c==getchar())==' '||c=='\t'||c=='\n'
```

este adevărată dacă și numai dacă se citește un caracter alb.

De aici, rezultă că pentru a avansa peste caracterele albe vom utiliza un ciclu *while* care se va executa atât timp cit expresia (1) este adevărată. Deci antetul ciclului este:

```
while((c==getchar())==' '||c=='\t'||c=='\n')
```

Deoarece acest ciclu nu are de realizat altceva decit să avanzeze peste caracterele albe, corpul lui se compune din instrucțiunea vidă.

Citirea caracterelor cuvintului de la intrare se realizează tot printr-o instrucțiune *while*. De data aceasta, ciclul *while* se execută atât timp cit expresia (1) este falsă, nu s-a intilnit sfîrșitul de fișier și nici nu s-au citit încă 70 de caractere.

Prima condiție se exprimă prin negația expresiei (1), presupunind că *c* conține dejasă caracterul curent.

```
(2) !(c==' '||c=='\t'||c=='\n')
```

Tinând seama de faptul că expresia

```
!(a&&b)
```

este echivalentă cu:

```
!a&&!b
```

expresia (2) se poate pune sub forma:

```
(3) !(c==' ')&&!(c=='\t')&&!(c=='\n')
```

Deoarece expresia:

```
!(a==b)
```

este echivalentă cu:

a!=b

expresia (3) se poate pune sub o formă mai simplă:

(4) $c \neq ' \& \& c \neq '\t' \& \& c \neq '\n'$

A doua condiție pentru continuarea ciclului *while* este ca să nu se citească sfîrșitul de fișier. Aceasta înseamnă că expresia:

(5) $c \neq EOF$

trebuie să fie adevărată.

Dacă notăm cu i numărătorul de caractere citite, diferite de caracterele albe, atunci ciclul *while* continuă dacă pe lângă condițiile (4) și (5) de mai sus, este adevărată și condiția exprimată prin expresia:

(6) $i < 70$ (se presupune că inițial i a fost 0).

De aici rezultă că ciclul *while* pentru citirea caracterelor cuvintului aflat la intrare se continuă atât timp cât expresiile (4), (5) și (6) sunt adevărate, ceea ce se exprimă conectând expresiile respective prin operatorul "și logic".

În corpul ciclului se păstrează codul caracterului citit în tabloul t de tip caracter, se incrementează numărătorul de caractere și apoi se citește caracterul următor. Acestea se realizează prin instrucțiunile de atribuire:

$t[i+1] = c;$

și

$c = getchar();$

Pentru determinarea prefixelor cuvintului urmăram pașii de mai jos unde:

- | | |
|-----|--|
| i | - Are ca valoare lungimea cuvintului. |
| j | - Are ca valoare lungimea prefixului curent. |
| k | - Numără caracterele unui prefix. |

1. $j=1$ - primul prefix are lungimea 1.
2. Cit timp $j < i$ se execută:
 - 2.1. $k=0$.
 - 2.2. Cit timp $k < j$ se execută:
 - 2.2.1. Se afișează $t[k]$.
 - 2.2.2. Se mărește k cu o unitate.
 - 2.3. Se afișează *newline* deoarece s-au afișat toate caracterele prefixului curent.
 - 2.4. Se mărește j cu o unitate pentru a afișa prefixul următor.

Pasul 2 se realizează prin două cicluri *while* imbricate:

```
while(j < i) {
    k=0; /* punctul 2.1. */
    while(k < j) { /* punctul 2.2. */
        putchar(t[k]); /* punctul 2.2.1. */
        k++; /* punctul 2.2.2. */
    }
    putchar('\n'); /* punctul 2.3. */
    j++; /* punctul 2.4. */
}
```

Sufixelete se pot determina printr-o secvență analogă. Dacă în cazul prefixelor acestea încep cu $t[0]$, în cazul sufixelor, caracterul de început este variabil. Astfel, sufixul de lungime 1 conține numai ultimul caracter al cuvintului, deci este $t[i-1]$. Sufixul de lungime 2 se compune din ultimele 2 caractere ale cuvintului:

$t[i-2]$ și $t[i-1]$.

Sufixul de lungime 3 se compune din caracterele:

$t[i-3], t[i-2]$ și $t[i-1]$.

În general, sufixul de lungime j se compune din caracterele:

$t[i-j], t[i-j+1], \dots, t[i-1]$.

De aceea, în acest caz variabila k nu se mai initializează cu 0, ci cu $i-j$.

Deoarece toate sufixele au ca ultim caracter pe $t[i-1]$, rezultă că variabila k variază cu pasul 1 de la $i-j$ pînă la $i-1$. În rest, secvența pentru determinarea sufixelor este aceeași cu cea pentru determinarea prefixelor.

PROGRAMUL BIV17

```
#include <stdio.h>

#define MAX 70

#include <stdlib.h>

main() /* citește un cuvint și afiseaza prefixele și sufixele lui */
{
    int c, i, j, k;
    char t[MAX];

/* avans peste eventualele caractere albe care preced cuvintul de prelucrat */
    while((c = getchar()) == ' ' || c == '\t' || c == '\n')
        ;
    if(c == EOF)
    {
        printf("cuvint vid\n");
        exit(1);
    }

/* se citesc caracterele cuvintului de la intrare */
    i = 0; /* numarator de caractere */
    while(c != ' ' && c != '\t' && c != '\n')
```

```

    && c != EOF && i < 70) {
        t[i++] = c;
        c = getchar();
    }

/* afiseaza prefixele cuvintului de lungime i */
j = 1; /* pare ca valoare lungimea prefixului curent */
while( j < i ) {
    k=0; /* numarator pentru caracterele prefixului: de la 0 la j-1 */
    while( k < j )
        putchar(t[k++]);
    putchar('\n'); /* dupa prefixul afisat */
    j++;
} /* sfarsit afisare prefixe */

/* afiseaza suflele cuvintului citit */
j = 1; /* lungimea sufleului curent */
while( j < i ) {
    k = i-j; /* numarator pentru caracterele sufleului:
                  de la i-j la i-1 */
    while( k < i )
        putchar(t[k++]);
    putchar('\n');
    j++;
}

```

4.7. Instrucțiunea for

Instrucțiunea *for*, ca și instrucțiunea *while*, se utilizează pentru a realiza o structură *repetitivă condiționată anterior*.

Formatul ei este:

```
for(exp1;exp2;exp3)
  instrucțiune
```

unde:

exp1, exp2 și - Sunt expresii.

exp3

Antetul instrucțiunii *for* este:

```
for(exp1;exp2;exp3)
```

iar *instrucțiune* este corpul ei și ea se executa repetat.

Expresia *exp1* se numește partea de *initializare* a ciclului *for*, iar *exp3* este partea de *reinițializare* a lui. Expresia *exp2* este condiția de continuare a ciclului *for* și ea joacă același rol cu expresia din ciclul *while*.

Instrucțiunea *for* se execută astfel:

- Se executa secvența de inițializare definită de *exp1*.

- Se evaluatează expresia *exp2*.

Dacă are o valoare diferită de zero (este *adevărată*), atunci se execută instrucțiunea care formează corpul ciclului.

Așa cum expresia are valoarea zero adică *false* se termină execuția instrucțiunii *for* și se trece la instrucțiunea urmatoare.

- Dupa execuțarea corpului ciclului se executa secvența de reinițializare definită de *exp3*.

Apoi se reia execuția de la pasul 2.

Ca și în cazul instrucțiunii *while*, instrucțiunea din corpul ciclului *for* nu se execută niciodată dacă *exp2* are valoarea zero chiar de la început.

Expresiile din antetul lui *for* pot fi și vide. Caracterele punct și virgula vor fi totdeauna prezente.

Exemplu:

Secvența de insumare a elementelor tabloului *tab*:

```
s=tab[0]+tab[1]+...+tab[n-1]
```

se realizează executând repetat instrucțiunea compusă:

```
{
    s=s+tab[i];
    i++;
}
```

unde inițial *s=0* și *i=0*, atât timp cât *i<n*. Ea se poate realiza cu ajutorul instrucțiunii *for* de mai jos:

```
for(s=0,i=0;i<n;i++)
    s=s+tab[i];
```

În acest caz expresia *exp1* este formată din două atribuiri:

s=0, i=0

exp2 este reprezentată de condiția

i<n

iar *exp3* este expresia

i++

Conform definiției instrucțiunii *for*, la început se evaluatează *exp1*, deci se execută atribuirile

s=0

și

i=0

Se evaluatează *exp2*, adică se determină valoarea condiției

i< n

Dacă ea este adevărată, atunci se execută corpul instrucțiunii *for*, adică suma $s=s+tab[i]$

Apoi se evaluează *exp3*, adică se execută incrementarea
i++;

După incrementarea lui *i* se revine la evaluarea condiției *exp2*. În felul acesta, ciclul continuă pînă cînd condiția *i< n* devine falsă, adică în momentul în care *i* devine egal cu *n*. Deci ciclul se întrerupe după ce elementele:

tab[0], tab[1], ..., tab[n-1]

au fost adăugate la *s*.

Secvența de mai sus poate fi scrisă cu ajutorul instrucțiunii *while* astfel:

```
s=0;
i=0;
while(i<n){
    s=s+tab[i];
    i++;
}
```

În general, instrucțiunea *for*:

```
for(exp1;exp2;exp3)
instructiune
```

poate fi scrisă cu ajutorul unei secvențe în care se utilizează instrucțiunea *while*:

```
exp1;
while(exp2)
{
    instructiune
    exp3;
}
```

Această echivalare nu are loc într-un singur caz și anume atunci cînd, în corpul instrucțiunii se utilizează instrucțiunea *continue* (vezi mai jos).

Reciproc, orice instrucțiune *while* poate fi scrisă cu ajutorul unei instrucțiuni *for* în care *exp1* și *exp3* sunt vide.

Așteptă, instrucțiunea *while* de mai jos:

```
while(exp)
instructiune
```

este echivalentă cu instrucțiunea *for*

```
for(;exp;)
instructiune
```

O instrucțiune *for* de forma

```
for(;;)
instructiune
```

este validă și ca este echivalentă cu instrucțiunea *while* de mai jos:

```
while(1)
instructiune
```

Un astfel de ciclu se poate termina prin alte mijloace decit cel obișnuit, cum ar fi instrucțiunea de revenire dintr-o funcție, un salt la o etichetă etc. (vezi mai jos).

Din cele de mai sus rezultă echivalența celor două cicluri. Autorii recomandă utilizarea instrucțiunii *for* în ciclurile în care sunt prezente părțile de inițializare și reinicializare. De obicei, aceste cicluri sunt aşa numitele cicluri cu *pas*.

Exerciții:

- 4.18 Să se scrie un program care citește întregul *n* din intervalul $[0,170]$, calculează și afișează pe *n!*

Acest exercițiu a fost rezolvat cu ajutorul instrucțiunii *while*. Propunem cititorului să rescrie exercițiile rezolvate cu ajutorul instrucțiunii *while* înlocuind-o pe aceasta cu *for*.

PROGRAMUL BIV18

```
#include <stdio.h>
#include <stdlib.h>
main() /* citește pe n din intervalul [0,170], calculeaza si afiseaza pe n! */
{
    int n,i;
    double f;

    printf("valoarea lui n:");
    if(scanf("%d",&n) != 1) {
        printf("nu s-a tastat un intreg\n");
        exit(1);
    }
    if( n < 0 || n > 170 ) {
        printf("n nu apartine intervalului [0,170]\n");
        exit(1);
    }
    for( f=1.0, i=2; i <= n; i++) f *= i;
    printf("n = %d\n! = %g\n", n,f);
}
```

- 4.19 Să se scrie un program care citește pe *n* și *a*, calculează și afișează valoarea lui *a* la puterea a *n*-a (a^{*n}). *n* este întreg nenegativ, iar *a* este un număr oarecare. Calculurile se fac în flotanta *long double*.

O metoda simplă pentru a ridica pe a la puterea a n -a, pentru n natural, este de a înmulți pe a cu el însuși de n ori.

PROGRAMUL BIV19

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

main()
{
    int a,n,i;
    long double apow;

    printf("Baza a = ");
    if (scanf("%Lf", &a) != 1) {
        printf("nu s-a tastat un numar\n");
        exit(1);
    }
    printf("Exponentul n = ");
    if (scanf("%d", &n) != 1 || n<0) {
        printf("nu s-a tastat un intreg nenegativ\n");
        exit(1);
    }
    for(i=0,p=1.0L; i < n; i++)
        p *= a;
    printf("a=%Lf\nn=%d\na^%Ln", a, n, p );
}
```

Observații:

1. Funcția *pow*, de prototip:

```
double pow(double x,double y);
```

se utilizează pentru date de tip *double*.

2. Metoda utilizată în programul de față nu este eficientă pentru n relativ mare, deoarece necesită multe înmulțiri. Numărul lor poate fi redus substanțial procedind ca mai jos.

Fie

$f=a$

Executând repetat instrucțiunea de atribuire

$f=f*a;$

se generează puterile:

(1) $a^2, a^4, a^8, a^{16}, a^{32}, \dots$

adică exponenții lui a sunt puteri ale lui 2.

Termenii șirului (1) se generează printr-un număr relativ mic de operații de înmulțire, a^n se poate exprima folosind numai factori din șirul (1) și eventual și

pe a , dacă n este impar. Acest lucru rezultă din faptul că orice număr natural pozitiv, se poate exprima ca o sumă de puteri ale lui 2. Într-adevăr, nu avem decit să reprezentăm numărul n în binar și să însumăm puterile lui doi din șirul:

(2) 1,2,4,8,16,32,64,...

care corespund cifrelor 1 din această reprezentare.

Corespondența dintre termenii șirului (2) și cifrele reprezentării binare se face de la dreapta spre stînga adică 1 se pune în corespondență cu ultima cifră a reprezentării binare, 2 cu penultima, 4 cu antepenultima etc.

Exemplu:

Fie $n=43$. În binar n se reprezintă astfel:

$n=10101_1$

Rezultă că n este suma termenilor 1, 2, 8 și 32 aleși din șirul (2) conform cifrelor 1 din reprezentarea binară a lui:

$$n=1+2+8+32=43$$

Pentru a calcula pe a^n va fi suficient să realizăm produsul factorilor $a^{**}2$, $a^{**}8$, și $a^{**}32$, factori aleși din șirul (1) și a .

Din cele de mai sus rezultă că pentru a reduce numărul înmulțirilor la calculul lui a^n este suficient să generăm factorii șirului (1) și să-i alegem pe cei ai căror exponenți însumări ne dau pe n . Selectarea lor se poate face simplu pornind de la reprezentarea binară a lui n . Astfel, dacă ultima cifră binară a lui n este 1, atunci se selecteză ca prim factor chiar a . Apoi, făcind o deplasare spre dreapta a lui n cu o poziție binară, se alege sau nu factorul $a^{**}2$, după cum ultima cifră binară a lui n , după ce s-a făcut deplasarea, este 1 sau nu. Procedeul continuă pînă cînd n devine egal cu zero.

Procesul de calcul se definește astfel:

1. $p=1$
2. $f=a$
3. Cît timp n este diferit de zero se execută:
 - 3.1. Dacă ultima cifră binară a lui n este egală cu 1, atunci $p=p*f$
 - 3.2. Se generează termenul următor din șirul (1): $f=f*f$
 - 3.3. n se deplacează spre dreapta cu o poziție binară.

Se observă că punctul 3.2. este necesar numai dacă $n>1$. Într-adevăr, dacă $n=1$ atunci procesul de calcul se termină, deoarece prin deplasarea lui n la dreapta cu o poziție binară acesta devine egal cu zero și deci ciclul nu se mai reia. Avind în vedere faptul că termenii șirului (1) cresc rapid, este important ca să suprimăm termenul care s-ar genera pentru $n=1$, deoarece cu toate că el nu este necesar, calculul lui poate conduce la o depășire flotantă superioară. În acest fel, punctul 3.2. se schimbă cu

dacă $n > 1$ atunci $f = f * f$

Programul BIV19A ridică pe a la n folosind procesul de calcul descris mai sus.

PROGRAMUL BIV19A

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

main() /* citeste pe n si a, calculeaza si afiseaza pe a**n */
{
    int n,i;
    long double a,p,f;

    printf("Baza a = ");
    if(scanf("%Lf", &a) != 1) {
        printf("nu s-a tastat un numar\n"); exit(1);
    }
    printf("Exponentul n = ");
    if(scanf("%d", &n) != 1 || n<0) {
        printf("nu s-a tastat un intreg nenegativ\n"); exit(1);
    }
    i = n;
    for(p=1.0L,f = a; n ; n >= 1) {
        if( n & 1 ) p *= f; /* factor selectat din sirul (1) */
        if( n > 1 ) f *= f; /* genereaza termenul urmator din sirul (1) */
    }
    printf("a=%Lf\tn=%d\ta**n=%Lf\n",a,i,p );
}
```

Observații:

1. $exp1$ realizează inițializările de la punctele 1 și 2 din descrierea de mai sus:
 $p=1.0L$, $f=a$
2. $exp2$, condiția de continuare a ciclului s-a redus la n ; deci ciclul continuă atât timp cât n este *adevărat*, adică este diferit de zero.
3. Punctul 3.3 este partea de reinițializare a ciclului *for* și ea constă din expresia
 $n >= 1$
care deplasează pe n cu un ordin binar spre dreapta.
4. Expresia
 $n&1$
are valoarea 1, dacă ultimul ordin binar al lui n este egal cu 1. În acest caz factorul generat în variabila f se selectează, deci se înmulțește cu p .

4.8. Instrucțiunea do-while

Instrucțiunea *do-while* realizează structura *ciclică condiționată posterior*. Această instrucțiune poate fi realizată cu ajutorul celorlalte instrucțiuni definite pînă în prezent. Cu toate acestea, prezența ei în limbaj mărește flexibilitatea în

programare.

Ea are formatul:

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

Instrucțiunea *do-while* se execută în felul următor:

1. Se execută *instrucțiune*
2. Se evaluează *expresie*; dacă aceasta are o valoare diferită de zero, atunci se revine la punctul 1, pentru a executa din nou *instrucțiune*; altfel (expresia are valoarea zero) se trece în secvență, adică la instrucțiunea următoare instrucțiunii *do-while*.

Se observă că în cazul acestei instrucțiuni întii se execută *instrucțiune* și apoi se testează condiția de repetare a execuției ei.

Instrucțiunea *do-while* este echivalentă cu secvența:

```
instrucțiune
while(expresie)
    instrucțiune
```

Considerăm și în acest caz că, *instrucțiune* reprezintă corpul ciclului *do-while*.

În cazul instrucțiunii *do-while* corpul ciclului se execută cel puțin o dată, spre deosebire de cazul instrucțiunilor *while* și *for*, cind este posibil să nu se execute niciodată.

Exerciții:

- 4.20 Să se scrie un program care citește un număr nenegativ, subunitar, calculează și afișează rădăcina pătrată din numărul respectiv cu o eroare mai mică decit 0,0000000001.

Programul de față nu folosește funcția *sqr* din biblioteca standard.

Pentru a extrage rădăcina pătrată dintr-un număr a din intervalul $(0,1)$, se poate folosi metoda iterativă a lui Newton de mai jos.

Fie sirul:

$x[0], x[1], \dots, x[n], \dots$

unde:

$$(1) x[n+1] = 0,5(x[n] + a/x[n]) \text{ pentru } n=0,1,2,\dots$$

Se demonstrează că sirul de mai sus este convergent și are limita egală cu rădăcina pătrată din a . Convergența este foarte rapidă pentru a subunitar. În acest caz se poate lua $x[0]=1$.

Pentru a obține rădăcina pătrată cu precizia cerută, este suficient ca diferența,

în valoare absolută, dintre doi termeni consecutivi ai șirului să fie mai mică decit EPS=0.0000000001

Să observăm că pentru a rezolva problema cu metoda indicată mai sus, la fiecare pas al calculului, sunt necesari numai doi termeni ai șirului:

- Din $x[n]$ se determină $x[n+1]$ cu relația (1).
- Se compară $\text{abs}(x[n]-x[n+1])$ cu EPS;

Dacă valoarea absolută respectivă nu este mai mică decit EPS, atunci se calculează $x[n+2]$ folosind relația (1), în care $x[n]$ se înlocuiește cu $x[n+1]$. Deci se poate reveni la punctul a de mai sus, după ce lui $x[n]$ îi se atribuie valoarea $x[n+1]$.

Deci, în loc să utilizăm tabloul x , este suficient să folosim două variabile $x1$ și $x2$ care pastrează în fiecare moment doi termeni consecutivi ai șirului de mai sus. Termenii șirului se obțin executând repetat secvența de atribuire:

```
x1=x2;
x2=0.5*(x1+a/x1);
```

După fiecare execuție a acestei secvențe, $x1$ și $x2$ au ca valori doi termeni consecutivi ai șirului căutat.

Secvența se va repeta atât timp cît $(2) \text{abs}(x1-x2) >= \text{EPS}$, test care îl facem după fiecare execuție a secvenței de atribuire. Dacă el se confirmă, atunci se repetă secvența respectivă, altfel valoarea variabilei $x2$ aproximiază rădăcina patrată din a cu o eroare mai mică decit EPS.

Acest proces de calcul se realizează imediat printr-o instrucție *do-while*.

Secvența de atribuire este corpul ciclului, iar expresia (2) reprezintă condiția de continuare a lui. Intrucția primul termen al șirului are valoarea 1, instrucția *do-while* este precedată de atribuirea:

```
x2=1;
```

PROGRAMUL BIV20

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

#define EPS 1e-10

main() /* citește pe a din intervalul [0,1], calculează și afisează rădăcina patrată din a */
{
    double a,x1,x2,y;

    printf("Tastati valoarea lui a = ");
    if(scanf("%lf",&a) != 1 || a < 0 || a > 1) {
        printf("Nu s-a tastat un numar in intervalul [0,1]\n");
        exit(1);
    }
    if ( a == 0 )
        x2 = 0; /* rădăcina patrată din zero este zero */
    else
```

```
    if( a == 1)
        x2 = 1; /* rădăcina patrată din 1 este 1 */
    else { /* a este diferit de 0 sau 1 */
        x2 = 1;
        do {
            x1 = x2;
            x2 = 0.5*(x1 + a/x1);
            if(( y = x1 - x2) < 0 )
                y = -y;
            /* y = abs(x1-x2)*/
        } while( y >= EPS );
    } /* sfârșit else */
    printf("a=% .11g\tsqrt(a)=% .11g\n",a,x2);
}
```

Observații:

- Instrucția *do-while* poate fi scrisă mai compact astfel:

```
do {
    x1=x2;
    x2=0.5*(x1+a/x1);
}while((y=x1-x2)<0?-y:y>=EPS);
```

- Pentru numere supraunitare se poate aplica aceeași metodă. În acest caz se poate lua ca prim termen al șirului chiar numărul din care se extrage rădăcina pătrată. O altă posibilitate este de a extrage rădăcina pătrată din inversul numărului. Aceasta deoarece metoda este rapid convergentă pentru numere subunitare.
Fie $a > 1$. Atunci $x=1/a$. Dacă y este rădăcina pătrată din x , atunci $1/y$ este rădăcina pătrată din a .

- 2.1 Să se scrie un program care citește un întreg pozitiv ce reprezintă o sumă exprimată în lei. Se cere să se afișeze numărul minim de bancnote și monede de 1000 lei, 500 lei, 200 lei etc. necesare pentru a exprima suma respectivă.

Procesul de calcul începe cu determinarea numărului de bancnote de 1000 de lei. În acest scop se imparte suma respectivă s la 1000:

$s/1000$

Pentru a determina numărul bancnotelor de 500 de lei se determină restul împărțirii lui s la 1000 și acest rest se imparte la 500 și aşa mai departe. Se obțin pașii următori:

- $q=s/1000$; afișează q
- $s=s \% 1000$;
- $q=s/500$; afișează q

```

4. s=s%500;
5. q=s/200; afisează q
6. s=s%200;
7. q=s/100; afisează q
8. s=s%100;
9. q=s/50; afisează q
10. s=s%50;
11. q=s/20; afisează q
12. s=s%20;
13. q=s/10; afisează q
14. s=s%10;
15. q=s/5; afisează q
16. s=s%5;
17. q=s/1; afisează q
18. s=s%1.

```

Se observă că secvența:

```

q=s/n; afisează q
s=s%n;

```

se repetă pentru valorile lui n egale cu

1000, 500, 200, 100, 50, 20, 10, 5 și 1

Pentru a putea executa repetat această secvență este nevoie de o metodă simplă prin care să se atribuie lui n valorile corespunzătoare. Aceasta se poate realiza dacă păstrăm datele respective într-un tablou de tip *int*. Fie:

```

mon[0]=1,    mon[1]=5,    mon[2]=10,
mon[3]=20,   mon[4]=50,   mon[5]=100,
mon[6]=200,  mon[7]=500,  mon[8]=1000.

```

Atunci n poate fi înlocuit prin $\text{mon}[i]$, unde inițial $i=8$ și se decrementează la fiecare pas al ciclului. Deci urmează să se execute repetat secvența:

```

q=s/mon[i]; afisează q
s=s%mon[i];
i=i-1;

```

Această secvență se va executa repetat pînă cind s devine zero.

PROGRAMUL BIV21

```

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

main() /* exprimă o sumă de lei în bancnote și monede de 1000 lei, 500 lei etc. */
{

```

```

int mon[9];
long s,q;
int i;

printf("Tastati suma: ");
if(scanf("%ld",&s) != 1 || s <= 0 ) {
    printf("nu s-a tastat un intreg pozitiv\n");
    exit(1);
}
mon[0] = 1; mon[1] = 5; mon[2] = 10; mon[3] = 20;
mon[4] = 50; mon[5] = 100; mon[6] = 200;
mon[7] = 500; mon[8] = 1000;
i = 8;
do {
    q = s/mon[i];
    if(q) /* afisarea se face dacă q este diferit de zero */
        if( i < 6 ) /* monede */
            if( q == 1 ) /* există o singura monedă */
                if( i == 0 ) /* o monedă de 1 leu */
                    printf("o moneda de 1 leu\n");
                else
                    printf("o moneda a %d lei\n",
                           mon[i]);
            else /* mai multe monede */
                if( i == 0 ) /* monede de 1 leu */
                    printf("%ld monede de 1 leu",q);
                else /* monede a mon[i] lei */
                    printf("%ld monede a %d lei\n",
                           q, mon[i]);
        else /* bancnote */
            if( q == 1 ) /* o singura bancnotă */
                printf("o bancnota a %d lei\n",mon[i]);
            else /* mai multe bancnote */
                printf("%ld bancnote a %d lei",q,
                       mon[i]);
    } while ( s %= mon[i--]);
}

```

4.9. Instrucțiunea continue

Această instrucție se poate utiliza numai în corpul unui ciclu. Ea permite abandonarea iterației curente. Formatul ei este:

continue;

Efectul instrucției *continue* este următorul:

- În corpul instrucțiunilor *while* și *do-while*:

La întâlnirea instrucției *continue* se abandonează iterația curentă și se trece la evaluarea *expresiei* care stabilește continuarea sau terminarea ciclului respectiv (expresia inclusă între paranteze rotunde și care urmează după cuvintul cheie *while*).

b. În corpul instrucțiunii *for*:

Ea înălțirea instrucțiunii *continue* se abandonează iterația curentă și se trece la execuția pasului de reinițializare.

Instrucțiunea *continue* nu este o instrucțiune obligatorie. Prezența ei mărește flexibilitatea în scrierea programelor C. Ea conduce adesea la diminuarea nivelurilor de imbriicare ale instrucțiunilor *if* utilizate în corpul cîclurilor.

4.10. Funcțiile standard *sscanf* și *sprintf*

Biblioteca standard a limbajelor C și C++ conține funcțiile *sscanf* și *sprintf* care sunt analoge funcțiilor *scanf* și respectiv *printf*. Ele au un parametru în plus la apel și anume primul lor parametru este adresa unei zone de memorie în care se pot păstra caractere ale codului ASCII. Ceilalți parametrii sunt identici cu cei întlniți în corespondențele lor, *scanf* și respectiv *printf*.

Primul parametru al acestor funcții poate fi numele unui tablou de tip *char*, deoarece un astfel de nume are ca valoare chiar adresa de început a zonei de memorie care îi este alocată.

Funcția *sprintf* se folosește, ca și funcția *printf*, pentru a realiza conversii ale datelor de diferite tipuri din formatele lor interne, în formate externe reprezentate prin succesiuni de caractere. Diferența constă în aceea că, de data aceasta caracterele respective nu se afișează pe terminalul standard, ci se păstrează în zona de memorie definită de primul parametru al funcției *sprintf*. Ele se păstrează sub forma unui șir de caractere și pot fi afișate ulterior din zona respectivă cu ajutorul funcției *puts*. De aceea, un apel al funcției *printf* poate fi totdeauna înlocuit cu un apel al funcției *sprintf*, urmat de un apel al funcției *puts*. O astfel de înlocuire este utilă cînd dorim să afișăm de mai multe ori același date. În acest caz se apelează funcția *sprintf* o singura dată pentru a face conversiile necesare din format intern în format extern, rezultatele conversiilor pastrindu-se într-un tablou de tip caracter. În continuare se pot afișa datele respective apelind funcția *puts* ori de câte ori este necesara afișarea lor. Funcția *sprintf*, ca și funcția *printf* returnează numărul octetilor șirului de caractere rezultat în urma conversiilor efectuate.

Exemplu:

```
int zi,luna,an;
char data_calend[11];
...
sprintf(data_calend,"%02d/%02d/%d",zi,luna,an);
puts(data_calend);
...
puts(data_calend);
```

Funcția *sscanf* realizează, ca și funcția *scanf*, conversii din format extern în format intern. Deosebirea constă în faptul că de data aceasta caracterele nu sunt

citite din zona tampon corespunzătoare tastaturii, ci ele provin dintr-o zonă de memorie a cărei adresă este definită de primul parametru al funcției *sscanf*. Aceste caractere pot proveni în zona respectivă în urma apelului funcției *gets*. În felul acesta, apelul funcției *scanf* poate fi înlocuit prin apelul funcției *gets* urmat de apelul funcției *sscanf*. Astfel de înlocuiri sunt utile cînd dorim să eliminăm eventualele erori aparute la tastarea datelor.

Funcția *sscanf*, ca și funcția *scanf*, returnează numărul cimpurilor convertite corect conform specificatorilor de format prezenți în parametrul de control. La înălțirea unei erori, ambele funcții își intrerup execuția și se revine din ele cu numărul cimpurilor tratate corect. Analizind valoarea returnată, se poate stabili dacă au fost prelucrate corect toate cimpurile sau a survenit o eroare.

În caz de eroare se poate reveni pentru a introduce corect datele respective. În acest scop este necesar să se eliminate caracterele începînd cu cel din poziția eronată.

În cazul în care se utilizează secvența:

```
gets
sscanf
```

abandonarea caracterelor respective se face automat reapelindu-se funcția *gets*. În cazul utilizării funcției *scanf* este necesar să se avanseze pînă la caracterul *newline* aflat în zona tampon atașată tastaturii sau să se videze zona respectivă prin funcții speciale.

În exercițiile care urmează vom folosi secvențele formate din apelurile funcției *gets* urmate de apelurile lui *sscanf*. O astfel de secvență se apelează repetat în cazul în care se întâlnesc erori în datele de intrare.

Exemplu:

```
char tab[255];
int zi,luna,an;
...
gets(tab);
sscanf(tab,"%d %d %d",&zi,&luna,&an);
```

Amintim că funcția *gets* returnează valoarea *NULL* la înălțirea sfîrșitului de fișier.

Funcțiile *sscanf* și *sprintf* au prototipurile în fișierul *stdio.h*.

Exerciții:

4.22 Să se scrie un program care citește un întreg pozitiv de tip *long*, stabilește dacă acesta este prim și afișează un mesaj corespunzător.

PROGRAMUL BIV22

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

```

main() /* citeste un intreg pozitiv de tip long, stabileste daca acesta este un numar prim
        si afiseaza un mesaj corespunzator */
{
    long n;
    long i;
    int j;
    char tab[255];

    do {
        printf("tastati un intreg pozitiv:");
        if(gets(tab) == NULL ) {
            printf("nu s-a tastat EOF\n");
            exit(1);
        }
        if(sscanf(tab,"%ld",&n) != 1 || n <= 0 ) {
            printf("nu s-a tastat un intreg pozitiv\n");
            j = 1;
            continue; /* se va relua ciclul deoarece se trece la evaluarea expresiei j
                        care este diferita de zero */
        }
        j = 0; /* ciclul se intrerupe deoarece s-a citit corect un intreg pozitiv */
    } while (j);
    for( j = 1, i = 2; i*i <= n && j; i++)
        if(n%i == 0) /* numarul nu este prim */
            j = 0;
    printf("numarul: %ld", n);
    if( j == 0 )
        printf(" nu ");
    printf(" este prim\n");
}

```

Observații:

- Utilizarea instrucțiunii *continue* se poate omite folosind o instrucție *if* cu alternativă *else*:

```

if(sscanf(...)!=1||n<=0){
    ...
    j=1;
} else
    j=0;

```

- Ciclul *for* continuă atât timp cât expresia:

$i^*i <= n \& \& j$

este adevărată. Această expresie se evaluatează de la stanga spre dreapta și din această cauză expresia

$i^*i <= n$

se evaluatează și atunci cind $j=0$. De aceea expresia respectivă este mai eficientă sub forma:

$j \& i^*i <= n$

În acest caz, pentru $j=0$ nu se mai evaluatează restul expresiei.

Expresia:

$i^*i <= n$

rezultă din faptul că, pentru a stabili că un număr este prim, este suficient să incercăm divizibilitatea lui prin numerele al căror patrat nu-l depășește pe n .

O altă posibilitate este aceea de a înlocui expresia

$i^*i <= n$

cu

$i <= \sqrt{n}$

În acest caz este util să se calculeze radacina patrata în afara ciclului *for*:

```

...
long q;
...
q=sqrt((double)n);
for(j=1,i=2;j&&i<=q;i++)
    if(n%i==0)
        j=0;
...

```

O altă simplificare posibilă este schimbarea instrucției *if* din corpul ciclului *for* cu:

$j=n \% i;$

4.11. Instrucția break

Instrucția *break* este înrudită cu instrucția *continue*. Ea are formatul:
break;

Poate fi utilizată în corpul unui ciclu. În acest caz, la întâlnirea instrucției *break* se termină execuția ciclului în al cărui corp este inclusă și execuția continuă cu instrucția următoare instrucției ciclice respective.

Instrucția *break*, la fel ca și instrucția *continue*, mărește flexibilitatea la scrierea programelor în limbajele C și C++.

Exerciții:

- Să se scrie un program care citește măsurile a , b , c ale laturilor unui triunghi, calculează și afișează aria triunghiului respectiv.

Pentru determinarea ariei se va folosi formula lui Heron. Dacă notăm cu p semiperimetrul triunghiului, atunci aria se calculează cu ajutorul expresiei:

$\text{aria}=\sqrt{p*(p-a)*(p-b)*(p-c)}$

Amintim că măsurile laturilor triunghiului satisfac relațiile

$p-a > 0$, $p-b > 0$, și $p-c > 0$

deci se va testa condiția

$$p-a > 0 \text{ && } p-b > 0 \text{ && } p-c > 0$$

care trebuie să fie adevarata.

PROGRAMUL BIV23

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

main() /* citeste masurile laturilor unui triunghi si afiseaza aria lui */
{
    double a,b,c,p;
    char tab[255];

    do /* citeste masurile laturilor triunghiului */
        do /* citeste pe a */
            printf("a= ");
            if(gets(tab) == NULL ) {
                printf("s-a tastat EOF\n");
                exit(1);
            }
            if(sscanf(tab,"%lf",&a) == 1 && a>0)
                break; /* seiese din ciclul pentru citirea lui a */
            printf("nu s-a tastat un numar pozitiv\n");
            printf("se reia citirea lui a\n");
        } while (1);
        do /* citeste pe b */
            printf("b= ");
            if(gets(tab) == NULL ) {
                printf("s-a tastat EOF\n");
                exit(1);
            }
            if(sscanf(tab,"%lf",&b) == 1 && b>0)
                break; /* seiese din ciclul do-while deoarece s-a citit valoarea lui b */
            printf("nu s-a tastat un numar pozitiv\n");
            printf("se reia citirea lui b\n");
        } while (1);
        do {
            printf("c= ");
            if(gets(tab) == NULL ) {
                printf("s-a tastat EOF\n");
                exit(1);
            }
            if(sscanf(tab,"%lf",&c) == 1 && c>0)
                break; /* seiese din ciclul do-while deoarece s-a citit valoarea lui c */
            printf("nu s-a tastat un numar pozitiv\n");
            printf("se reia citirea lui c\n");
        } while(1);
    p = (a + b + c) / 2;
    if( p-a > 0 && p-b > 0 && p - c > 0 )
        break; /* seiese din ciclul do-while exterior deoarece a,b,c
                  pot fi masurile laturilor unui triunghi */
```

```
printf("a= %g\ tb =%g\ tc= %g\n",a,b,c);
printf("nu pot reprezenta masurile laturilor \
unui triunghi\n");
} while (1);
printf("aria = %g\n",sqrt(p*(p-a)*(p-b)*(p-c)) );
}
```

4.24 Să se scrie un program care citește două numere naturale și pozitive, calculează și afișează cel mai mare divizor comun al lor (c.m.m.d.c.).

Calculul celui mai mare divizor comun a două numere se poate face folosind algoritmul lui Euclid. Pași acestui algoritm sunt:

1. Se citesc cele două numere.
Fie acestea a și b .
2. Se împarte a la b ; fie q și r cîtul, respectiv restul acestei împărțiri:
 $a=q*b+r$.
3. Dacă r este diferit de zero, se fac atribuirile:
 $a=b$ și $b=r$
și se revine la pasul 2.
Altfel algoritmul se întrerupe și cel mai mare divizor comun al lor este b .

PROGRAMUL BIV24

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

main() /* citeste două numere întregi și pozitive și afisează c.m.m.d.c al lor */
{
    long m,n;
    long a,b;
    long r;
    char tab[255];

    do {
        printf("primul numar= ");
        if(gets(tab) == NULL ) {
            printf("s-a tastat EOF\n");
            exit(1);
        }
        if(sscanf(tab,"%ld",&m) == 1 && m > 0 )
            break;
        printf("nu s-a tastat un intreg pozitiv\n");
    } while(1);

    do {
        printf("al doilea numar= ");
        if(gets(tab) == NULL ) {
            printf("s-a tastat EOF\n");
            exit(1);
        }
        if(sscanf(tab,"%ld",&n) == 1 && n > 0 )
            break;
```

```

        break;
        printf("nu s-a tastat un intreg pozitiv\n");
    } while(1);

    a = m; b = n;

    do {
        r = a % b;
        if(r) {
            a = b; b = r;
        }
    } while(r);

    printf("(%ld,%ld) = %ld\n", m,n,b);
}

```

4.12. Instrucțiunea switch

Instrucțiunea *switch* permite realizarea structurii *selective*. Aceasta este o generalizare a structurii *alternative* și a fost introdusă de C.A.R. Hoare. Ea poate fi realizată prin instrucțiuni *if* imbricate. Utilizarea instrucțiunii *switch* face ca programul să fie mai clar decât dacă se utilizează varianta cu instrucțiuni *if* imbricate.

Structura *selectivă*, în forma în care a fost ea acceptată de către adeptii programării structurate, se realizează în limbajul C cu ajutorul următorului format al instrucțiunii *switch*:

```

switch(expresie){
    case c1:
        sir_1
        break;
    case c2:
        sir_2
        break;
    ...
    case cn:
        sir_n
        break;
    default:
        sir
}

```

unde:

c1, c2, ..., cn - Sint constante.

sir_1, sir_2, ..., sir_n - Sint succesiuni de instrucțiuni.

sir

Instrucțiunea *switch* cu formatul indicat mai sus se execută astfel:

1. Se evaluatează expresia din parantezele rotunde.
2. Se compară pe rind valoarea expresiei cu valorile constantelor *c1, c2, ..., cn*.
 - Dacă valoarea expresiei coincide cu una din constante sa zicem cu *ci*, atunci se execută secvența *sir_ci*, apoi se trece la instrucțiunea următoare instrucțiunii *switch*, adică la instrucțiunea aflată după acolada închisă care termină instrucțiunea.
 - Dacă valoarea respectivă nu coincide cu nici una din constantele *c1, c2, ..., cn*, atunci se execută succesiunea de instrucțiuni *sir* și apoi se ajunge la instrucțiunea următoare instrucțiunii *switch*.

Menționăm că este posibil să nu se ajungă la instrucțiunea următoare instrucțiunii *switch* în cazul în care succesiunea de instrucțiuni selectată pentru execuție (*sir_i* sau *sir*) va defini ea însăși un alt mod de continuare a execuției programului (de exemplu, execuția instrucțiunii de revenire dintr-o funcție, saltul la o instrucțiune etichetată etc.).

Succesiunile *sir, sir_1, ..., sir_n* se numesc *alternativele* instrucțiunii *switch*. Alternativa *sir* este opțională, deci într-o instrucțiune *switch*, secvența

default:

sir

poate fi absentă.

În acest caz, dacă valoarea expresiei nu coincide cu valoarea nici uneia dintre constantele *c1, c2, ..., cn*, atunci instrucțiunea *switch* nu are nici un efect și se trece la execuția instrucțiunii următoare.

Instrucțiunea *switch* de mai sus este echivalentă cu următoarea instrucțiune *if* imbricată:

```

if(expresie==c1)
    sir_1
else
    if(expresie==c2)
        sir_2
    else
        if(expresie==c3)
            sir_3
        else
            if
            ...
            else
                if(expresie==cn)
                    sir_n
                else
                    sir

```

Instrucțiunea *break* de la sfârșitul fiecărei alternative, după secvențele *sir_1*, *sir_2*, ..., *sir_n*, permite ca la intilnirea ei să se treaca la execuția instrucțiunii următoare instrucțiunii *switch*. Se obișnuiește să se spună că instrucțiunea *break* permite ieșirea din instrucțiunea *switch*.

Amintim că instrucțiunea *break* poate fi utilizată numai în corpurile ciclurilor și în alternativele instrucțiunii *switch*.

Prezența ei la sfârșitul fiecărei alternative a unei instrucțiuni *switch*, nu este obligatorie. În cazul în care instrucțiunea *break* este absenta la sfârșitul unei alternative, după execuția succesivă de instrucțiuni din compunerea alternativei respective se va trece la execuția succesivă de instrucțiuni din alternativa următoare a aceleiași instrucțiuni *switch*.

Astfel, dacă o instrucțiune *switch* are formatul:

```
switch(expresie){
    case c1:
        sir_1
    case c2:
        sir_2
}
```

atunci ea este echivalentă cu următoarea secvență:

```
if(expresie==c1){
    sir_1
    sir_2
}else
    if(expresie==c2){
        sir_2
    }
```

Exerciții:

- 4.25. Sa se scrie un program care citește o cifră din intervalul [1,7] și afișază denumirea zilei din săptămână corespunzătoare cifrei respective (1-luni, 2-martii etc).

PROGRAMUL BIV25

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

main() /* citește o cifră din intervalul [1,7] și afișează denumirea zilei din
         săptămână corespunzătoare cifrei respective */
{
    int i;
    char t[255];
    do {
        puts("tastati o cifra din intervalul [1,7]");
        if(gets(t) == NULL) {
            exit(1);
        }
        if(sscanf(t, "%d", &i) != 1 || i < 0 || i > 7) {
            puts("nu s-a tastat o cifra din intervalul [1,7]");
            continue;
        }
        switch(i) {
            case 1:
                puts("luni");
                break;
            case 2:
                puts("marti");
                break;
            case 3:
                puts("miercuri");
                break;
            case 4:
                puts("joi");
                break;
            case 5:
                puts("vineri");
                break;
            case 6:
                puts("sîmbata");
                break;
            case 7:
                puts("duminica");
                break;
        }
    } while(1);
}
```

```
    puts("s-a tastat EOF"); exit(1);
}
if(sscanf(t, "%d", &i) == 1 && i > 0 && i < 8 )
    break;
puts("nu s-a tastat o cifra din intervalul [1,7]");
} while(1);
switch(i) {
    case 1:
        puts("luni");
        break;
    case 2:
        puts("marti");
        break;
    case 3:
        puts("miercuri");
        break;
    case 4:
        puts("joi");
        break;
    case 5:
        puts("vineri");
        break;
    case 6:
        puts("sîmbata");
        break;
    case 7:
        puts("duminica");
        break;
}
```

- 4.26. Să se scrie un program care citește construcții de forma:

op1 op op2
unde:
- op1 și op2 sint intregi de tip long.
- op este unul din caracterele: +, -, * sau /,
și afișează valoarea expresiei citite.

Operația de impărțire este impărțirea întreagă.

Acet exemplu este intilnit frecvent în diferite lucrări relativ la limbajul C. De exemplu, autorii limbajului C, în lucrarea [2] propun acest exemplu pentru a simula un calculator rudimentar de birou.

PROGRAMUL BIV26

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

main() /* citește o expresie de forma: op1op2 unde op1 și op2 sint intregi de tip long,
         iar op este unul din operatorii +, -, *, / afișează valoarea expresiei respective */
{
    long op1, op2, rez;
    char op;
    char t[255];

    for ( ; ; ) {
        do {
            printf("tastati expresia:op1op2: ");
            if(gets(t) == NULL) {
                exit(1);
            }
            if(sscanf(t, "%ld %c %ld", &op1, &op, &op2) != 3) {
                puts("nu s-a tastat o expresie de forma: op1op2");
                continue;
            }
            if(op == '+') {
                rez = op1 + op2;
            }
            else if(op == '-') {
                rez = op1 - op2;
            }
            else if(op == '*') {
                rez = op1 * op2;
            }
            else if(op == '/') {
                if(op2 == 0) {
                    puts("nu s-a tastat un număr de la zero");
                    continue;
                }
                rez = op1 / op2;
            }
            else {
                puts("operatorul este invalid");
                continue;
            }
            printf("rezultatul este: %ld\n", rez);
        } while(1);
    }
}
```

```

if( gets(t) == NULL ) exit(0);
if(sscanf(t,"%ld %c %ld",&op1,&op,&op2) == 3) break;
printf("expresie eronata\n");
} while(1);

switch(op) {
    case '+':
        rez=op1 + op2; break;
    case '-':
        rez=op1 - op2; break;
    case '*':
        rez= op1 * op2; break;
    case '/':
        if(op2 == 0) {
            printf("divizor nul\n");
            rez = 0;
        }
        else
            rez = op1/op2;
        break;
    default:
        printf("operator eronat\n");
        rez = 0;
} /* sfirsit switch*/

printf("op1= %ld op= %c op2= %ld rez=%ld\n",
       op1,op,op2,rez);
} /* sfirsit for */
} /* sfirsit main */

```

Observație:

Programul de față permite să se evaluateze mai multe expresii separate prin caractere albe. El își intrerupe execuția la întâlnirea sfîrșitului de fișier.

4.13. Instrucțiunea goto

Instrucțiunea *goto* nu este o instrucțiune absolut necesară la scrierea programelor în limbajul C. Cu toate acestea, ea se dovedește utilă în anumite cazuri. Autorii limbajului recomandă utilizarea ei în cazul în care se dorește să se ieșă din mai multe cicluri imbicate. Astfel de situații apar adesea la întâlnirea unei erori. În astfel de situații, de obicei, se dorește să se facă un salt în afara ciclurilor în care a intervenit eroarea, pentru a se ajunge la o secvență externă lor de tratare a erorii respective.

Înainte de a indica formatul instrucțiunii *goto* să precizăm noțiunea de *etichetă*.

Prin *etichetă* înțelegem un nume urmat de două puncte:

nume:

nume utilizat în definirea unei etichete este numele etichetei respective.

După o etichetă urmărează o instrucțiune.

Se obișnuiește să se spună că eticheta *prefixează* instrucțiunea care urmărează după ea.

Etichetele sint *locale* în corpul funcției în care sunt definite.

Instrucțiunea *goto* are formatul:

goto *nume*;

unde:

nume - Este numele unei etichete definită în corpul aceleiași funcții în care se află instrucțiunea *goto*.

La întâlnirea instrucțiunii *goto*, se realizează un salt la instrucțiunea prefixată de eticheta al cărei *nume* se află după cuvintul cheie *goto*.

Deoarece o etichetă este locală în corpul unei funcții rezultă că ea este nedefinită în afara corpului funcției respective. În felul acesta, o instrucțiune *goto* poate realiza un salt numai la o instrucțiune din corpul aceleiași funcții în care este ea utilizată.

Deci, o instrucțiune *goto* nu poate face salt din corpul unei funcții la o instrucțiune din corpul altor funcții.

Menționăm că nu se justifică utilizarea abuzivă a acestei instrucțiuni. Se recomandă să fie utilizată pentru a simplifica ieșirea din cicluri imbicate.

Exemplu:

Presupunem că într-un punct al programului, aflat în interiorul mai multor cicluri, se depistează o eroare și se dorește să se continue execuția programului cu o secvență de tratare a erorii respective. În acest caz, vom folosi o instrucțiune *goto* ca mai jos.

```

for(...){
    ...
    while(...){
        ...
        do{
            ...
            for(...){
                ...
                if(i==0)
                    goto divzero;
                else
                    x=y/i;
                ...
            }
            ...
        }while(...);
        ...
    }
    ...
}

```

```

/* sevența de tratare a erorii */
divzero:
printf(...);
...

```

În absență instrucțiunii *goto* se poate realiza același lucru folosind un indicator și o serie de teste realizate asupra lui.

4.14. Programarea procedurală, funcții, apelul și revenirea din ele

Incepînd cu primele limbaje de programare de nivel înalt s-a utilizat *programarea procedurală*.

Aceasta s-a dezvoltat din necesitatea de a utiliza într-un program, aceeași sevență de calcul de mai multe ori. Pentru a evita o astfel de repetiție, sevența de instrucțiuni corespunzătoare se organizează ca o parte distință și se face un salt la ea, ori de cîte ori este nevoie în program de procesul de calcul respectiv. Acest salt este cîtu revenire la instrucțiunea următoare instrucțiunii care a făcut saltul și de aceea el diferă de saluturile realizate prin instrucțiunea *goto*. Sevența de instrucțiuni organizată în acest fel are diferențe denumiri în diverse limbaje de programare: subprogram, subrutină, procedură etc. Pentru început considerăm denumirea de *procedură*.

Uneori procedura trebuie să exprime același proces de calcul dar care se realizează cu date diferențite. În acest caz, procedura trebuie realizată generală, facînd *abstracție* de datele respective. De exemplu, pentru a evalua expresia:

$$(1) 4^{**}10-3^{**}20$$

putem construi o procedură pentru ridicarea la putere, care să fie generală și să facă abstracție de valorile efective pentru bază și exponent: 4 și 10 pentru prima ridicare la putere, 3 și 20 pentru cea de a doua. Această generalizare se realizează considerînd ca fiind variabile atît baza cît și exponentul, iar valorile lor se precizează la fiecare apel al procedurii implementate în acest fel.

Aceste variabile utilizate pentru a putea implementa o procedură generală și care se concretizează la fiecare apel al procedurii, se numesc *parametri formalăi*.

În felul acesta, procedura apare ca un rezultat al unui proces de generalizare necesar implementării ei. Ca orice proces de generalizare ea presupune o abstractizare care se realizează prin utilizarea parametrilor formalăi.

Valorile de la apel ale parametrilor formalăi se numesc *parametrii efectivi*.

Programarea *procedurală* are la bază utilizarea procedurilor, iar acestea la rîndul lor realizează o abstractizare prin parametri.

În lucrarea [14] se subliniază și un alt aspect al abstractizării realizate prin utilizarea procedurilor și anume acela că o procedură se poate asemăna cu o *cutie*

neagră, la care î se transferă date la apel, iar aceasta furnizează rezultatele la revenirea din ea. În momentul apelului, se face abstracție de metoda de prelucrare a datelor de intrare în rezultate care se mai numesc și date de ieșire.

În toate limbajele de programare se consideră două categorii de proceduri:

- Proceduri care definesc o valoare de revenire.
- Proceduri care nu definesc o valoare de revenire.

Procedurile din categoria a. de obicei se numesc *funcții*. *Valoarea de revenire* se mai numește și *valoare de întoarcere* sau *valoarea returnată* de funcție.

Procedura pentru calculul ridicării la putere este un exemplu de funcție. Ea are ca parametrii baza și exponentul, iar ca valoare de întoarcere sau returnată, rezultatul ridicării valorii bazei la valoarea exponentului, valori care sunt definite la apel.

În limbajele C și C++ atît procedurile din categoria a., cît și cele din categoria b. se numesc funcții. Deci, în aceste limbaje distingem funcții care returnează o valoare la revenirea din ele, precum și funcții care nu returnează nici o valoare.

O funcție are o *definiție* și atîtea *apeluri* într-un program, cîte sint necesare.

O definiție de funcție are formatul:

antet
corp

unde:

antet

- Are formatul:

tip nume (lista declarațiilor parametrilor formali)

corp

- Este o instrucțiune compusă.

Amintim că *tip* este cuvîntul cheie *void* pentru funcții care nu returnează nici o valoare la revenirea din ele. Pentru alte detalii vezi paragraful 1.4.

Apelul unei funcții trebuie să fie precedat de definiția sau de prototipul ei.

Prototipul unei funcții conține informații asemănătoare cu cele din antetul ei:

- tipul valorii returnate;
- numele funcției;
- tipurile parametrilor.

El poate avea același format ca și antetul funcției, în plus este urmat de punct și virgulă. Pentru alte detalii vezi paragraful 1.11.

Limbajul C se livrează cu o serie de funcții care au o utilizare frecventă în programe. Ele se păstrează într-un fișier special în format OBJ, adică compilat și se adaugă la fiecare program în fază de editare. Aceste funcții sunt numite *funcții standard de bibliotecă*. Ele au prototipurile în diferite fișiere de extensie .h. Exemple de astfel de funcții sunt funcțiile cu ajutorul căror se realizează operații de intrare/ieșire (printf, scanf, puts, gets, getch, putch etc.), funcțiile pentru calculul funcțiilor elementare (sqrt, sin, cos, atan, log, pow etc.), funcții de conversii (sscanf, sprintf etc.) etc.

Prototipurile funcțiilor standard se includ în program înaintea apelurilor lor folosind construcția `#include` tratată prin preprocesor (vezi paragraful 1.12.).

Exemple de fișiere cu prototipuri:

`stdio.h` pentru funcțiile `printf`, `scanf`, `gets`, `puts` etc.;

`conio.h` pentru funcțiile `putch`, `getch`, `getche` etc.;

`math.h` pentru funcțiile elementare `sqrt`, `sin`, `cos` etc.

Menționăm că pînă în prezent în toate exercițiile au fost apelate numai funcții standard.

Apelul unei funcții care nu returnează o valoare, se realizează printr-o *instrucțiune de apel*. Aceasta are formatul:

nume (*listă parametrilor efectivi*);

unde:

nume - Este numele funcției care se apelează.

listă parametrilor efectivi - Este fie vidă, cînd funcția nu are parametri, fie o expresie sau mai multe separate prin *virgulă*.

Deci un *parametru efectiv* este o expresie.

Parametrii efectivi de la apel se corespund cu cei formali prin ordine. Primul parametru efectiv corespunde primului parametru formal, cel de al doilea parametru efectiv corespunde celui de al doilea parametru formal și.a.m.d.

Amintim că parametrii unei funcții (formali sau efectivi) se mai numesc și *argumente*.

O funcție care returnează o valoare, poate fi apelată fie printr-o *instrucțiune de apel*, fie ca *operand* al unei expresii. În cazul în care funcția se apelează printr-o instrucțiune de apel, se pierde valoarea returnată. Cînd funcția se apelează ca operand al unei expresii, valoarea returnată de ea se utilizează la evaluarea expresiei respective.

Spre exemplificare să considerăm funcția `getch`. Noi am apelat această funcție atât ca operand în expresii de atribuire de forma:

```
c=getch()
```

cit și printr-o instrucțiune de apel de forma:

```
getch();
```

În primul caz, valoarea codului ASCII al caracterului citit de la tastatură se atribuie variabilei *c*. La cel de al doilea apel, valoarea codului ASCII al caracterului citit de la tastatură nu este utilizată. În acest caz apelul se face cu scopul de a afîşa ecranul utilizator și de a bloca execuția programului pînă la acționarea unei taste oarecare, corespunzătoare caracterelor ASCII.

La apelul unei funcții, valorile parametrilor efectivi se atribuie parametrilor formali corespunzători, apoi execuția continuă cu prima instrucțiune din corpul

funcției apelate.

În cazul în care tipul unui parametru efectiv difera de tipul parametrului formal care-i corespunde, în limbajul C se convertește automat valoarea parametrului efectiv spre tipul parametrului formal respectiv. În limbajul C++ se utilizează o regulă mai complexă pentru apelul funcțiilor și de aceea se recomandă să nu se folosească nici în limbajul C, regula de conversie automată amintită mai sus. În acest scop se poate utiliza operatorul de conversie explicită ((tip)), adică folosind aşa numitele expresii *cast*.

De exemplu, dacă funcția *f* are un parametru de tip *double* și *n* este o variabilă de tip *int*, atunci în locul apelului:

```
f (n);
```

se recomandă a folosi apelul cu conversie explicită a lui *n* spre tipul *double*:

```
f ((double)n);
```

Acest al doilea apel se realizează identic în ambele limbiage.

La revenirea dintr-o funcție apelată printr-o instrucțiune de apel se va continua cu execuția instrucțiunii următoare celei care a facut apelul. În cazul în care o funcție este apelată ca un operand al unei expresii, la revenirea din ea se continuă cu evaluarea expresiei respective.

Revenirea dintr-o funcție se poate realiza în următoarele două moduri:

- După execuția ultimei instrucțiuni din corpul funcției (s-a ajuns la acolada închisă care termină corpul funcției).
- La intîlnirea instrucțiunii *return*.

Instrucțiunea *return* este instrucțiunea de *revînire* dintr-o funcție. Ea are formatele:

- return;*
sau
- return expresie;*

Cel de al doilea format se folosește în corpul unei funcții care returnează o valoare la revenirea din ea. Valoarea expresiei din instrucțiunea *return* este chiar valoarea returnată de funcție.

În cazul în care tipul acestei expresii diferă de tipul care precede numele din antetul funcției, valoarea expresiei se convertește automat spre tipul din antet, înainte de a se reveni din funcție.

Formatul I al instrucțiunii de revenire se utilizează numai în corpul unei funcții care nu returnează nici o valoare. Dintr-o astfel de funcție se poate reveni și în modul indicat la punctul *a* de mai sus.

Faptul că instrucțiunea *return* (în ambele formate) poate fi scrisă în orice punct al corpului unei funcții, permite o mai mare flexibilitate în programarea în

limbajele C și C++.

Exerciții:

- 4.27 Sa se scrie o funcție care are ca parametru un întreg n din intervalul $[0,170]$, calculeaza și returneaza pe $n!$

Numim *factorial* aceasta funcție. Ea returneaza o valoare flotantă în dublu precizie. Rezulta că funcția are antetul:

```
double factorial(int n)
```

La început funcția verifică dacă n aparține intervalului $[0,170]$. În cazul în care n nu aparține acestui interval, funcția va returna valoarea -1. Metoda de calcul este aceeași cu cea utilizată în exercițiul 4.12.

FUNCȚIA BIV27

```
double factorial(int n)
/* calculeaza și returneaza pe n! pentru n în intervalul [0,170];
   altfel returneaza -1 */
{
    double f;
    int i;

    if( n < 0 || n > 170)
        return -1.0;
    for( i=2, f=1.0; i <= n; i++)
        f *= i;
    return f;
}
```

- 4.28 Sa se scrie un program care calculează și listează pe $m!$ pentru $m=0, 1, 2, \dots, 170$.

Acest program apelează funcția *factorial* definită mai sus pentru a-l calcula pe $m!$ pentru o valoare data a lui m .

Definiția funcției *factorial* și funcția principală care o apelează pot fi editate în același fișier sursă. Cele două funcții pot fi editate în orice ordine. În cazul în care definiția funcției *factorial* se află în fișierul sursă după funcția principală, apelul funcției *factorial* din funcția principală, trebuie să fie precedat de prototipul ei.

PROGRAMUL BIV28

```
double factorial(int); /* prototipul funcției factorial */

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

```
main() /* afiseaza pe m! pentru m = 0,1,2,...,170 */
{
    int m;

    for(m=0; m<171; m++) {
        printf("m=%d\tm!=%g\n", m, factorial(m));
        if((m+1)%23 == 0) {
            printf("actionati o tasta pentru a continua\n");
            getch();
        } /* sfîrșit if*/
    } /* sfîrșit for */
} /* sfîrșit main */

double factorial(int n)
/* calculeaza și returneaza n! pentru n în intervalul [0,170];
   altfel returneaza -1 */
{
    double f;
    int i;

    if( n < 0 || n > 170) return -1.0;
    for( i=2, f=1.0; i <= n; i++) f *= i;
    return f;
} /* sfîrșit factorial */
```

Observații:

1. Funcția *factorial* este apelată ca parametru efectiv în apelul funcției *printf*. La apelul funcției *factorial*, valoarea parametrului ei efectiv m se atribuie parametrului formal n . Această valoare este apoi folosită în corpul funcției *factorial* pentru a-l calcula pe $n!$ și deci și pe $m!$. Valoarea respectivă se obține ca valoare a lui f . La revenirea din funcție se returnează valoarea lui f , adică chiar $m!$, care se afisează folosind specificatorul de format %g.
2. Cele două funcții pot fi editate în fișiere separate. De exemplu, presupunem că funcția *factorial* se editează în fișierul BIV27.CPP, iar funcția principală în fișierul BIV28A.CPP. În acest caz putem include fișierul BIV27.CPP folosind construcția #include pentru a compila împreună cele două funcții. Includerea se poate face înainte de definiția funcției *main* sau după ea. În cazul în care facem includerea în fața funcției *main*, nu mai este necesar să indicăm prototipul funcției *factorial* deoarece apelul ei este precedat chiar de definiția funcției. Procedind în acest fel se obține varianta de mai jos.

PROGRAMUL BIV28A

```
#include <stdio.h>
#include <conio.h>
#include "biv27.cpp"

main() /* afiseaza pe m! pentru m = 0,1,2,...,170 */
{
```

```

int m;
for(m=0; m<171; m++) {
    printf("m=%d\tn!=%g\n",m,factorial(m));
    if((m+1)%23 == 0) {
        printf("actionati o tasta pentru a continua\n");
        getch();
    }
}

```

3. O altă posibilitate de compilare și link-editare a funcțiilor unui program, editate în mai multe fișiere, este aceea de a utiliza un fișier de tip *Project* (cu extensia .prj). Un astfel de fișier, conține numele fiecărui fișier (împreună cu extensia lui) care se compilează și link-editează în vederea obținerii fișierului executabil al programului. În acest fișier pot fi indicate nu numai fișiere de tip sursă (cu extensia .CPP), ci și fișiere de tip obiect (cu extensia .OBJ) sau chiar fișiere cu biblioteci de funcții, altele decât cele ale sistemului.

Fișierele de tip *project* pentru limbajele Turbo C și C++ nu sunt compatibile. Ele se construiesc folosind meniul *Project* al celor două sisteme integrate de dezvoltare.

Avantajele fișierelor de tip *Project* constau în aceea că, la fiecare lansare se compilează în mod automat numai sursele în care s-au făcut modificări. De aceea, în cazul programelor sursă mari se recomandă împărțirea lor în mai multe fișiere sursă și compilarea lor prin utilizarea fișierelor de tip *Project*. De obicei, într-un fișier sursă se grupează funcții "inrudite", adică funcții care prelucrează în comun subseturi de date sau sunt logic legate între ele.

Întrucit exercițiile pe care le prezentăm nu sunt de dimensiuni mari, vom utiliza frecvent incluziile de fișiere folosind construcția #include a preprocesorului.

- 4.29 Să se scrie o funcție care are ca parametri doi întregi x și y , calculează și returnează numărul aranjamentelor de x obiecte luate cîte y .

La început funcția testează dacă x aparține intervalului $[1,170]$, iar y intervalului $[1,x]$. În caz de eroare, funcția returnează valoarea -1.

Dacă notăm cu $A(x,y)$ numărul aranjamentelor de x obiecte luate cîte y , atunci:

$$A(x,y)=x*(x-1)*(x-2)*...*(x-y+1)$$

FUNCȚIA BIV29

```

double aranjamente ( int x, int y)
/* calculeaza si returneaza numarul aranjamentelor de x obiecte luate cîte y */
{
    double a;

```

```

int i;

if(x < 1 || x > 170 ) return -1.0;
if(y < 1 || y > x ) return -1.0;
a = 1.0;
i = x - y + 1;
while( i <= x ) a *= i++;
return a;
}

```

- 4.30 Să se scrie un program care calculează și afișează numărul aranjamentelor de n obiecte luate cîte k , pentru $n=1,2,\dots,170$ și $k=1,2,\dots,n$.

Programul de față apelează funcția *aranjamente* definită în exercițiul precedent.

PROGRAMUL BIV30

```

#include <stdio.h>
#include <conio.h>
#include "biv29.cpp" /* contine functia aranjamente */

main() /* calculeaza si afiseaza numarul aranjamentelor de n obiecte
         luate cîte k, pentru n in intervalul [1,170] si k in intervalul [1,n] */
{
    int k,n;

    for(n = 1; n <= 170; n++) {
        printf("actionati o tasta pentru a continua\n");
        getch();
        printf("\nn= %d\n",n);
        for(k=1; k <= n; k++) {
            printf("k=%d\tA(n,k)=%g\n",k,aranjamente(n,k));
            if(k%23 == 0 ) {
                printf("actionati o tasta pentru a continua\n");
                getch();
            } /* sfîrșit if*/
        } /* sfîrșit for interior */
    } /* sfîrșit for exterior */
} /* sfîrșit main*/

```

- 4.31 Să se scrie o funcție care are ca parametri doi întregi x și y , calculează și returnează numărul combinațiilor de x elemente luate cîte y .

Funcția testează dacă x aparține intervalului $[1,170]$, iar y intervalului $[0,x]$. În caz de eroare, funcția returnează valoarea -1.

Pentru calculul numărului combinațiilor de x elemente luate cîte y se determină raportul dintre numărul aranjamentelor de x elemente luate cîte y și $y!$.

În acest scop, funcția de față apelează funcțiile *aranjamente* și *factorial* definite în exercițiile 4.29. și respectiv 4.27.

FUNCȚIA BIV31

```
#include "biv27.cpp"
#include "biv29.cpp"

double combinari(int x, int y)
/* calculeaza si returneaza numarul combinariilor de x obiecte luate cte y */
{
    if( x < 1 || x > 170 ) return -1.0;
    if( y < 0 || y > x ) return -1.0;
    if( y == 0 || y == x ) return 1.0;
    return aranjamente(x,y)/factorial(y);
}
```

Observații:

1. Un calcul mai rapid se realizeaza daca se ține seama de relația $c(x,y)=c(x,x-y)$ unde prin $c(x,y)$ s-a notat numărul combinărilor de x obiecte luate cte y . Aceasta relație este util sa se aplice pentru $y > x/2$. În acest scop ultima instrucție se înlocuiește cu:


```
if( y > x/2 )
    return aranjamente (x,x-y)/factorial(x-y);
else
    return aranjamente (x,y)/factorial(y);
```
2. O altă variantă pentru calculul numărului combinărilor este relația: $c(x,y)=(x!)((x-1)!)((x-2)!)...((x-y+1))/y!$ În acest caz se evita calculul valorilor $A(x,y)$ și $y!$ care cresc rapid odată cu creșterea valorilor lui x și y . Aceasta se realizează cu ajutorul funcției de mai jos.

FUNCȚIA BIV31A

```
double combinari(int x, int y)
/* calculeaza si returneaza numarul combinariilor de x obiecte luate cte y */
{
    int i;
    double c;

    if( x < 1 || x > 170 ) return -1.0;
    if( y < 0 || y > x ) return -1.0;
    if( y == 0 || y == x ) return 1.0;
    c = 1.0;
    for( i = 1; i <= y; i++ ) c = (c * (x - i + 1)) / i;
    return c;
}
```

- 4.32 Să se scrie un program care calculează și afișează numărul combinărilor de n obiecte luate cte k , pentru $n=1,2,...,170$ și $k=0,1,2,...,n$.

Programul utilizează funcția BIV31A.

PROGRAMUL BIV32

```
#include <stdio.h>
#include <conio.h>
#include "biv31a.cpp"

main() /* afiseaza numarul combinariilor de n obiecte luate cte k
         pentru n in intervalul [1,170] si k in intervalul [0,n] */
{
    int k,n;

    for(n=1; n<= 170; n++) {
        printf("actionati o tasta pentru a continua\n");
        getch();
        printf("\nn=%d\n",n);
        for( k=0; k <=n ; k++) {
            printf("k=%d\tC(n,k)=%g\n", k, combinari(n,k));
            if((k+1)%23 == 0) {
                printf("actionati o tasta pentru a continua\n");
                getch();
            } /* sfirsit if */
        } /* sfirsit for interior */
    } /* sfirsit for exterior */
} /* sfirsit main */
```

- 4.33 Să se scrie o funcție care ridică la o putere întreagă nenegativă un număr flotant de tip *long double*.

Funcția utilizează metoda expusă în exercițiul 4.19.

FUNCȚIA BIV33

```
long double ldrp( long double x, int n)
/* ridică pe x la puterea n */
{
    long double f,p;

    for(p=x,f=1.0L; n ; n >>= 1) {
        if( n&1) f *= p;
        if( n > 1) p *= p;
    }
    return f;
}
```

- 4.34 Să se scrie un program care tabelizează puterile întregi ale unui număr flotant de tip *long double*, exponentul variind, cu pasul 1, între două limite m și n .

PROGRAMUL BIV34

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

```

#include "biv33.cpp"

main() /* tabelaza puterile intregi ale unui numar*/
{
    char t[255];
    long double f,g;
    int i,m,n;

    do {
        printf("tastati numarul care se ridica la puteri: ");
        if( gets(t) == NULL ) {
            printf("s-a tastat EOF\n");
            exit(1);
        }
        if(sscanf(t,"%Lf",&f) == 1 ) break;
        printf("nu s-a tastat un numar\n");
    } while(1);
    do {
        do {
            printf("tastati limita inferioara a exponentului: ");
            if(gets(t) == NULL ) {
                printf("s-a tastat EOF\n");
                exit(1);
            }
            if(sscanf(t,"%d",&m) == 1 ) break;
            printf("nu s-a tastat un intreg\n");
        } while (1);

        do {
            printf("tastati limita superioara a exponentului: ");
            if(gets(t) == NULL ) {
                printf("s-a tastat EOF\n");
                exit(1);
            }
            if(sscanf(t,"%d", &n ) == 1 ) break;
            printf("nu s-a tastat un intreg\n");
        } while(1);

        if(m <= n)
            break;
        printf("limita inferioara:%d o depaseste pe cea\ncu superioara:%d\n", m,n);
    } while (1);
    if( f == 0 && m < 0 ) {
        printf("putere negativa pentru zero\n");
        exit(1);
    }
    for( i=m; i <= n; i++) {
        if( i < 0 )
            g = 1.0/ldrp(f,-i);
        else
            g = ldrp(f,i);
        printf("%Lg la puterea %d este: %Lg\n",f,i,g);
    }
}

```

4.15. Apel prin valoare și apel prin referință

În limbajul C, la apelul unei funcții, fiecărui parametru formal î se atribuie *valoarea* parametrului efectiv care-i corespunde. Deci, la apelul unei funcții se transferă valorile parametrilor efectivi. Din această cauză, se spune că apelul este prin *valoare* (*call by value*).

Exemplu:

Fie funcția *factorial* de prototip:

double factorial(int n);

definită în exercițiul 4.27. La apelul:

fact=factorial(10);

se atribuie parametrului formal *n* valoarea 10. Această atribuire se face înainte de execuția primei instrucțiuni a funcției *factorial*. De aceea, prin acest apel funcția *factorial* calculează pe 10!.

În unele limbi de programare, la apel nu se transferă valorile parametrilor efectivi, ci *adresele* acestor valori. În acest caz, se spune că apelul este prin *referință* (*call by reference*).

Între cele două tipuri de apeluri există o diferență esențială și anume:

- în cazul apelului prin valoare, funcția apelată nu poate modifica parametrii efectivi din funcția care a facut apelul, neavind acces la ei;
- în cazul apelului prin referință, funcția apelata, disponind de adresa parametrilor efectivi, ii poate modifica.

Așa se întimplă, de exemplu, în cazul limbajului FORTRAN, unde apelul este prin referință.

De aici, rezultă că la apelul prin valoare transferul datelor este *unidirectional*, adică valorile se transferă prin parametri numai de la funcția care face apelul spre cea care a fost apelată, nu și invers.

Există situații cind se dorește ca funcția apelata să modifice valorile parametrilor efectivi. Unele limbi permit ambele tipuri de apeluri. Așa este, de exemplu, cazul limbajului Pascal. Parametri declarați în Pascal prin *var* se transferă prin referință.

În limbajul C++, spre deosebire de limbajul C, există ambele apeluri, ca și în limbajul Pascal.

Apelul prin valoare este util în cazul în care funcția apelată nu trebuie să modifice valorile parametrilor efectivi, deoarece în felul acesta parametrii efectivi sunt protejați față de eventualele modificări care ar putea apărea din greșelă.

Acumătă înseamnă că o funcție poate să modifice parametrii ei formali fară riscul ca prin modificarea respectivă să influențeze valoile parametrilor efectivi corespunzători. Așa de exemplu, funcția *ldrp*, definită în exercițiul 4.31 modifica valoarea parametrului formal *n* la zero. Cu toate acestea, valoare parametrului efectiv corespunzător lui *n* nu suferă nici o modificare, având aceeași valoare la revenirea din funcție ca și în momentul apelului.

În limbajul C, un parametru efectiv poate fi numele unui tablou. De exemplu o funcție, pe care o numim *sum* și care insumează elementele unui tablou tip *int*. Această funcție are doi parametri:

- tab* - Este numele tabloului ale căruia elemente se insumează.
- n* - Este numarul elementelor tabloului.

Fie returneaza suma:

```
tab[0]+tab[1]+...+tab[n].
```

Conform celor spuse mai sus, funcția are următorul aspect:

```
int sum(int tab[], int n)
```

Amintim că dacă un parametru efectiv este un număr de tablou unidimensional, atunci parametrul formal corespunzător se poate declara fără a începe în parantezele patrate, limita superioară a indicelui sau.

Dacă *a* este un tablou declarat ca mai jos:

```
int a[10];
```

atunci pentru a insuma elementele tabloului respectiv vom putea folosi acest lucru:

```
x=sum(a,10);
```

Deoarece *a* este numele unui tablou, *a* are ca valoare adresa primului sau element, adică adresa lui *a[0]*. Apelul fiind prin valoare, parametrul *tab* primește valoarea lui *a* deci *tab* are și el ca valoare adresa lui *a[0]*. Rezultă că *tab[0]*, *tab[1]*, ..., *tab[9]* reprezintă aceleși elemente ca *a[0]*, *a[1]*, ..., *a[9]*. De aceea, corpul funcției *sum* referindu-se la elementele lui *tab*, se referă de fapt chiar la elementele lui *a*:

```
for (n=0; n<10; n++)
    s+=tab[n];
```

Se observă că, în acest caz, deși apelul s-a făcut prin valoare (valoarea lui *a* s-a atribuit lui *tab*), s-a realizat un apel prin referință, deoarece valoarea atribuită parametrului formal este o adresa.

În acest caz, funcția apelată poate modifica elementele tabloului al cărui nume s-a folosit ca parametru efectiv.

Într-adevăr, în exemplul de mai sus, atribuirea:

```
tab[3]=100;
```

să fie în corpul funcției *sum* are ca efect atribuirea valoii 100 elementului *a[3]*.

În concluzie, în limbajul C, apelul este prin *valoare*. Acest apel *devine* prin *referință* în cazul în care parametrul efectiv este numele unui tablou.

Ulterior să vedem că apelul prin referință poate fi realizat în limbajul C și în cazul variabilelor simple, folosind *pointeri*.

De asemenea, într-un alt capitol să vedem că în limbajul C++ putem să stabilim, pentru fiecare parametru, tipul de apel: prin *valoare* sau prin *referință*.

Exerciții:

4.35 Să se scrie o funcție care calculează valoarea unui polinom folosind schema lui Horner.

Parametrii funcției sunt:

- | | |
|----------|--|
| <i>c</i> | - Tabloul cu coeficienții polinomului. |
| <i>n</i> | - Este de tip <i>long double</i> . |
| <i>x</i> | - Gradul polinomului. |
| | - Este de tip <i>int</i> . |
| | - Valoarea variabilei. |
| | - Este de tip <i>long double</i> . |

Tabloul *c* conține coeficienții polinomului în ordine descrescătoare a puterilor lui *x*.

- | | |
|---------------|---|
| <i>c[0]</i> | - Este coeficientul lui <i>x**n</i> . |
| <i>c[1]</i> | - Este coeficientul lui <i>x**(n-1)</i> . |
| ... | - |
| <i>c[n-1]</i> | - Este coeficientul lui <i>x</i> . |
| <i>c[n]</i> | - Este termenul liber. |

Se știe că un polinom *p(x)*, cu coeficienții *c[0]*, *c[1]*, ..., *c[n]*, în ordine descrescătoare ale puterilor lui *x*, se poate pune sub forma:

$$p(x) = ((...((c[0]*x+c[1])*x+c[2])*x+...+c[n-2])*x+c[n-1])*x+c[n]$$

Fie

$$s=c[0]$$

Se observă că paranteza cea mai interioară are ca valoare, valoarea expresiei:

$$s*x+c[1]$$

Să atribuim lui *s* valoarea acestei expresii:

$$s = s*x+c[1]$$

În continuare trebuie să se calculeze valoarea expresiei:

$$s*x+c[2]$$

Atribuim din nou lui *s* valoarea acestei expresii:

```
s=s*x+c[2]
```

În general, la al i -lea pas se evaluează expresia

```
s*x+c[i]
```

și apoi valoarea ei se atribuie lui s :

```
s=s*x+c[i]
```

Deci, calculul valorii polinomului $p(x)$ revine la execuția repetată a atribuirii $s=s*x+c[i]$, pentru $i=1, 2, \dots, n$,

valoarea inițială a lui s fiind $c[0]$. Aceasta conduce la instrucțiunea *for* de mai jos:

```
for(s=c[0], i=1; i<=n; i++)
    s=s*x+c[i];
```

FUNCȚIA BIV35

```
long double polinom(long double x, int n, long double c[])
/* calculeaza valoarea polinomului
   p(x)=c[0]*x**n + c[1]*x**(n-1) +...+ c[n-1]*x + c[n] */
{
    long double s;
    int i;

    for(s = c[0], i = 1; i <= n; i++) s = s*x + c[i];
    return s;
}
```

Observație:

Biblioteca standard a limbajului C conține funcția *poly* care calculează valoarea unui polinom de gradul n cu coeficienții de tip *double*. Ea are prototipul:

```
double poly(double x, int n, double c[]);
```

care se află în fisierul *math.h*.

În acest caz, $c[0]$ este termenul liber, $c[1]$ coeficientul lui x . În general $c[i]$ este coeficientul lui x la puterea i .

4.36 Să se scrie un program care realizează următoarele:

- citește pe x de tip *long double*;
 - citește pe n de tip *int* din intervalul $[0,100]$;
 - citește coeficienții $a[0], a[1], \dots, a[n]$ ai polinomului $p(x)$ de tip *long double*;
 - calculează și afișează valoarea polinomului $p(x)$.
- $a[0]$ este coeficientul lui x^{**n} , $a[1]$ este coeficientul lui $x^{**(n-1)}$ etc.

PROGRAMUL BIV36

```
#include <stdio.h>
#include <stdlib.h>
#include "biv35.cpp"

#define MAXGRAD 100
main() /* citește pe x, n, a[0], a[1], ..., a[n], calculeaza și afiseaza valoarea polinomului
          p(x)=a[0]*x**n + a[1]*x***(n-1) +...+a[n-1]*x + a[n] */
{
    long double x,a[MAXGRAD+1];
    int i,n;
    char t[255];

    /* citeste valoarea lui x */
    do {
        printf("valoarea lui x: ");
        if(gets(t) == NULL ) {
            printf("s-a tastat EOF\n");
            exit(1);
        }
        if(sscanf(t,"%Lf", &x) == 1 ) break;
        printf("nu s-a tastat un numar\n");
    } while(1);

    /* citeste pe n */
    do {
        printf("valoarea lui n: ");
        if(gets(t) == NULL ) {
            printf("s-a tastat EOF\n");
            exit(1);
        }
        if(sscanf(t,"%d", &n) == 1 && n >= 0 && n <= 100 )
            break;
        printf("nu s-a tastat un intreg din intervalul
               [0,100]\n");
    } while (1);

    /* citeste coeficienții polinomului */
    for( i=0; i <= n; i++)
        do {
            printf("coeficientul a[%d]: ",i);
            if(gets(t) == NULL ) {
                printf("s-a tastat EOF\n");
                exit(1);
            }
            if(sscanf(t,"%Lf", &a[i]) == 1)
                break; /* ieșire din do-while */
            printf("nu s-a tastat un numar\n");
        } while(1);
    /* sfârșit citire coeficienți polinom */

    printf("x=%Lg\tngrad=%d\tval pol=%Lg\n",x,n,polinom(x,n,a));
}
```

- 4.37 Să se scrie o funcție care citește cel mult n numere și le păstrează în tabloul $ndtab$ care este de tip *double*. Funcția returnează numărul numerelor citite.

FUNCȚIA BIV37

```
int ndcit(int n, double ndtab[])
/* citește cel mult n numere și le păstrează în tabloul ndtab;
   returnează numărul numerelor citite */
{
    int i;
    double d;
    char t[255];

    for(i=0; i < n; i++) {
        printf("elementul [%d]= ", i);
        if(gets(t) == NULL) return i;
        if(sscanf(t, "%lf", &d) != 1) break;
        ndtab[i] = d;
    }
    return i;
}
```

- 4.38 Să se scrie o funcție care citește componentele unui vector precedate de numărul lor. Funcția returnează numărul componentelor vectorului respectiv.

Accastă funcție folosește funcția definită în exercițiul precedent.

FUNCȚIA BIV38

```
int dvcit(int nmax, double dvector[])
/* citește un întreg n și cele n componente ale vectorului dvector;
   returnează numărul componentelor citite */
{
    int i,n;
    char t[255];

    do {
        printf("numarul elementelor= ");
        if(gets(t) == NULL) {
            printf("s-a tastat EOF\n");
            exit(1);
        }
        if(sscanf(t, "%d", &n) == 1 && n > 0 && n <= nmax )
            break;
        printf("nu s-a tastat un întreg din intervalul [1,%d]\n",
               nmax);
    } while(1);
    if((i=ndcit(n,dvector)) != n) {
        printf("nu s-a reusit citirea celor n=%d componente\n",n);
        printf("s-au citit numai %d componente\n",i);
    }
    return i;
}
```

-)
- 4.39 Să se scrie un program care citește componentele vectorilor x și y , calculează și afișează produsul lor scalar.

Componentele celor doi vectori se tastează astfel:

- numărul componentelor vectorilor (n);
- cele n componente ale lui x ;
- cele n componente ale lui y .

Programul de față utilizează funcția definită în exercițiul precedent.

PROGRAMUL BIV39

```
#include <stdio.h>
#include <stdlib.h>
#include "biv37.cpp"
#include "biv38.cpp"

#define MAX 1000

main() /* citește componente vectorilor x si y precedate de numarul lor;
           calculeaza si afiseaza produsul lor scalar */
{
    int i,n;
    double x[MAX],y[MAX],s;

    /* citește pe n si componente vectorului x */
    n = dvcit(MAX,x);

    /* citește componente vectorului y */
    if(ndcit(n,y) != n) {
        printf("nu sunt n=%d componente pentru y\n",n);
        exit(1);
    }

    /* calculeaza produsul scalar */
    s = 0.0;
    for(i=0; i < n; i++) s += x[i]*y[i];
    printf("(x,y)= %g\n", s);
}
```

- 4.40 Să se scrie o funcție care sortează în ordine crescătoare, elementele unui tablou de tip *double*.

Spunem despre un tablou *tab*, de elemente numerice, că este sortat crescător (descreșcător), dacă elementele lui satisfac relația:

$\text{tab}[i] \leqslant (\geqslant) \text{tab}[i+1]$

pentru

$i=0,1,2,\dots,n-2$

unde:

n - Este numărul elementelor tabloului tab .

Funcția care realizează sortarea are doi parametrii:

tab - Tabloul de sortat.

n - Numărul elementelor tabloului.

La ora actuală, există o serie de metode de sortare, care diferă între ele, mai ales, prin ordinul de mărime al pașilor necesari pentru a realiza sortarea. Pentru amănunte se poate consulta [3].

Cele mai simple metode sunt cele mai puțin eficiente și necesită un număr de pași de ordinul lui n^2n , n fiind numărul elementelor de sortat. Prezentăm mai jos o astfel de metodă, care este cunoscută sub denumirea de *metoda bulelor (bubble sort)*.

Ea constă în următoarele:

Se compară primele două elemente ale tabloului de sortat și dacă nu sunt în ordinea cerută (de exemplu crescătoare), se permute între ele. Apoi se compară elementul al doilea cu al treilea și se procedează ca mai sus. În general, se compară două elemente învecinate, să zicem al i -lea cu al $i+1$ -lea și dacă nu sunt în ordinea cerută, atunci ele se permute. Se procedează în acest fel cu toate perechile de elemente învecinate ale tabloului de sortat. Apoi procesul respectiv se reia de la început. El se întrerupe cind se constată că toate elementele învecinate ale șirului sunt în ordinea cerută (în cazul de față, crescătoare).

Exemplu:

Fie șirul de numere:

8,3,9,5,4,7

pe care dorim să-l ordonăm crescător.

Prima parcurgere a șirului:

8 3 9 5 4 7 șirul inițial

3 8 9 5 4 7 șirul după permutarea lui 8 cu 3

3 8 5 9 4 7 șirul după permutarea lui 9 cu 5

3 8 5 4 9 7 șirul după permutarea lui 9 cu 4

3 8 5 4 7 9 șirul după permutarea lui 9 cu 7.

A doua parcurgere a șirului:

3 8 5 4 7 9 șirul după prima parcurgere

3 5 8 4 7 9 șirul după permutarea lui 8 cu 5

3 5 4 8 7 9 șirul după permutarea lui 8 cu 4

3 5 4 7 8 9 șirul după permutarea lui 8 cu 7.

A treia parcurgere a șirului:

3 5 4 7 8 9 șirul după a doua parcurgere a lui

3 4 5 7 8 9 șirul după permutarea lui 5 cu 4.

La a patra parcurgere a șirului nu se mai face nici o permutare deoarece perechile de elemente vecine sunt în ordine crescătoare, deci șirul este sortat crescător.

O proprietate a acestei metode este aceea că la fiecare parcurgere a șirului de numere, cel puțin un element ajunge să-și ocupe poziția sa finală în conformitate cu ordonarea avută în vedere. Astfel, după prima parcurgere a șirului, elementul ajuns pe ultimul loc nu va mai fi permuat, ocupându-și locul final. În exemplul de mai sus, 9 a ajuns pe ultimul loc după prima permutare, poziție care este finală pentru el. După cea de-a doua parcurgere, 8 a ajuns și el pe poziția lui finală. Întimplător și 7 a ajuns pe poziția lui finală la cea de-a doua parcurgere și așa mai departe.

Dacă se ordonează crescător un tablou de numere, atunci după prima parcurgere a elementelor lui, elementul maxim va avea indicele cel mai mare. După cea de-a doua parcurgere, maximul dintre elementele rămase (fără a considera ultimul element) va ocupa penultima poziție în tablou etc.

Se obișnuiește să se spună că elementele urcă în pozițiile lor la fel ca și bulele de aer în apa care fierbe, de unde și denumirea metodei.

Funcția de mai jos conține un ciclu pentru parcurgerea elementelor tabloului tab . În acest ciclu se compară elementele vecine curente și dacă acestea nu sunt în ordine crescătoare, ele se permute. Deoarece, după o parcurgere a șirului se reințelege parcurgerea lui, acest ciclu este conținut într-un altul care realizează execuția repetată a ciclului de parcurgere a șirului. Se ajunge în acest fel la următorii pași:

1. Cât timp șirul nu este sortat se execută pasul 2.
Altfel se întrerupe procesul de sortare.
2. Se parcurge șirul și se permute perechile de elemente învecinate care sunt în ordine descrescătoare.

Pasul 2 se realizează printr-un ciclu *for* de forma:

```
for (i=0; i<n-1; i++) {
    if (tab[i]>tab[i+1]) {
        /* se permute elementele tab[i] și tab[i+1] */
        t=tab[i];
        tab[i]=tab[i+1];
        tab[i+1]=t;
    }
}
```

Acest ciclu trebuie repetat atât timp cât elementele tabloului tab nu sunt în ordine crescătoare. Problema care se pune în legătură cu execuția repetată a acestui ciclu *for* este aceea de a ști cind s-a ajuns la situația în care elementele tabloului tab sunt în ordine crescătoare. Observăm că în acest caz ciclul *for* se execută fără a mai face permutări. Deci, tabloul este sortat cind ciclul *for* îl parcurge fără a face permutări. Acest fapt poate fi detectat folosind o variabilă, să zicem *ind*, care se anulează înaintea fiecărei execuții a ciclului *for* și se setează la

valoarea 1 în cazul în care ciclul respectiv face o permutare. În felul acesta, variabila *ind* are valoarea 1 după execuția ciclului *for*, dacă acesta a făcut cel puțin o permutare și zero în caz contrar.

În concluzie, variabila *ind* are valoare zero cind sirul este sortat și 1 în caz contrar. Variabila *ind* se gestionează astfel:

- inițial *ind* are valoarea 1;
- înainte de instrucțiunea *for*, *ind* se face egal cu zero;
- în secvența de permutare a două elemente învecinate, *ind* se face egal cu 1.

Se obține secvența:

```
ind = 0;
for (i=0; i<=n-1; i++)
    if(tab[i]>tab[i+1])
        t=tab[i];
        tab[i]=tab[i+1];
        tab[i+1]=t;
        ind=1;
}
```

Această secvență se execută repetat atât timp cît *ind* nu este egal cu zero, adică tabloul nu este sortat. În acest scop se folosește un ciclu *while* care are ca și corp chiar secvența de mai sus:

```
while(ind){
    ind=0;
    for(i=0; i<=n-1; i++)
        if(tab[i]>tab[i+1]){
            ...
        } /* sfîrșit if */
} /* sfîrșit while */
```

FUNCȚIA BIV40

```
void ordcresc(double tab[], int n)
/* ordenează elementele lui tab în ordine crescătoare */
{
    int i, ind;
    double t;

    ind = 1;
    while(ind) {
        ind = 0;
        for(i=0; i< n-1; i++)
            if(tab[i] > tab[i+1]) {
                t = tab[i]; tab[i] = tab[i+1]; tab[i+1] = t;
                ind = 1;
            } /* sfîrșit if */
    } /* sfîrșit while */
}
```

4.41 Să se scrie un program care citește un sir de numere separate prin caractere albe și le afișează în ordine crescătoare. După ultimul număr se tastează un caracter nenumeric. Se presupune că la intrare există cel puțin două numere și cel mult 1000.

Numerelor se citesc apelind funcția *ndcit* definită în exercițiul 4.37. Sortarea lor în ordine crescătoare se realizează folosind funcția *ordcresc* definită în exercițiul precedent.

PROGRAMUL BIV41

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include "biv37.cpp"
#include "biv40.cpp"

#define MAX 1000

main() /* citește un sir de numere și le afișează în ordine crescătoare */
{
    double tnr[MAX];
    int n,i;

    if((n=ndcit(MAX,tnr)) < 2) {
        printf("s-au citit mai puțin de 2 numere\n");
        exit(1);
    }
    ordcresc(tnr,n);
    for(i=0; i < n; i++)
        printf("tnr[%d]=%g\n", i,tnr[i]);
    if((i+1)%23 == 0) {
        printf("actionati o tasta pentru a continua\n");
        getch();
    }
}
```

4.42 Să se scrie o funcție care căută un element de valoare dată *a*, într-un tablou de numere sortat crescător. Funcția returnează indicele unui element de valoare egală cu *a* sau -1 în cazul în care nu există un astfel de element.

Elementul de valoare *a* poate fi găsit comparând fiecare element al tabloului cu *a* pînă cind se găsește un element egal cu *a*. În cazul în care nu există un astfel de element în tabloul respectiv, compararea se face cu toate elementele tabloului pentru a putea constata că nu există elementul căutat. O astfel de metodă de căutare, numită căutare *liniară*, deși este foarte simplă, ea necesită un număr relativ mare de comparații.

În cazul tablourilor sortate se poate utiliza o metodă de căutare mult mai rapidă, numită metoda *căutării binare*. Această metodă constă în următoarele.

Fie tab tabloul în care se caută un element de valoare a și care este sortat crescător. Deci:

$tab[0] \leq tab[1] \leq \dots \leq tab[n-1]$

1. Se compară elementul din mijlocul tabloului (de indice $[(n-1)/2]$) cu a .
2. Dacă acesta are valoarea egală cu a , înseamnă că s-a găsit un element de valoare a și se returnează indicele lui.
Altfel se continuă cu punctul 3.
3. Dacă elementul din mijlocul tabloului este diferit de a , atunci din cauză că tabloul este sortat, elementul căutat se află fie în prima jumătate a tabloului, fie în a două jumătate.
Mai mult decât atât, se poate stabili exact în care jumătate urmează să se facă căutarea și anume:

- Dacă $tab[(n-1)/2] > a$, atunci elementul căutat se poate afla numai în prima jumătate a tabloului tab . Deoarece elementele tabloului următoare lui $tab[(n-1)/2]$, sunt mai mari decât el, deci urmează să se facă căutarea în subtabloul:

$tab[0] \leq tab[1] \leq \dots \leq tab[(n-1)/2-1]$.

- Dacă $tab[(n-1)/2] < a$, atunci elementul căutat se află numai în jumătatea a două a tabloului tab . Deoarece elementele tabloului, precedente lui $tab[(n-1)/2]$, sunt mai mici decât el, deci căutarea se va face în subtabloul:
 $tab[(n-1)/2+1] \leq tab[(n-1)/2+2] \leq \dots \leq tab[n-1]$.

4. După determinarea subtabloului în care trebuie căutat elementul de valoare a , se procedează în mod analog, adică se determină elementul aflat la mijlocul acestui subtablou și apoi se procedează ca la pașii 2 și 3 de mai sus și așa mai departe.

Exemplu:

Fie tabloul tab de 10 elemente care au valorile:

$tab[0]=3 \quad tab[1]=4 \quad tab[2]=7$
 $tab[3]=10 \quad tab[4]=13 \quad tab[5]=20$
 $tab[6]=21 \quad tab[7]=35 \quad tab[8]=41$
 $tab[9]=48$

Se cere indicele elementului de valoare 20.

Avem:

$$n=10, [(n-1)/2]=[9/2]=4.$$

Se compară $tab[4]$ cu 20:

$$tab[4]=13 < 20$$

decid pentru căutarea următoare se alege jumătatea a două a tabloului:

$$tab[5]=20, tab[6]=21, tab[7]=35, tab[8]=41, tab[9]=48.$$

Elementul din mijlocul acestui tablou este de indice $[(5+9)/2]=7$. Avem:

$tab[7]=35 > 20$

deci se va face căutarea în prima jumătate a acestui subtablou, adică se selectează subtabloul:

$tab[5]=20, tab[6]=21$

În continuare, se alege elementul de indice $[(5+6)/2]=5$, deci $tab[5]$. Cum $tab[5]=20$, rezultă că în acest moment a fost găsit elementul căutat și se revine din funcție cu valoarea 5 (indicele elementului căutat).

Pentru determinarea, la fiecare pas, a subtabloului în care să se facă căutarea este bine să utilizăm două variabile inf și sup , prima avind ca valoare indicele primului element al subtabloului curent selectat, iar cea de-a doua, indicele ultimului element al aceluiași subtablou. Pașii procesului de căutare vor fi:

1. $inf=0$
Indicele primului element al tabloului.
2. $sup=n-1$
Indicele ultimului element al tabloului.
3. $i = (inf+sup)/2$
Indicele elementului din mijlocul tabloului.
4. Dacă
 $tab[i]=a$
căutarea se termină și se returnează valoarea lui i .
Altfel se continuă cu 5.
5. Dacă
 $tab[i] > a$
atunci
 $sup=i-1$
Limita superioară se coboară la valoarea $i-1$ pentru a selecta prima jumătate a tabloului.
Se reia de la punctul 3.
Altfel înseamnă că $tab[i] < a$ și atunci
 $inf=i+1$
Limita inferioară crește la valoarea $i+1$ pentru a selecta jumătatea a două a tabloului.
Se reia de la punctul 3.

Acest proces se desfășoară corect în cazul în care, în tablou, există cel puțin un element de valoare a . În caz contrar, el continuă indefinitely. Se observă că la pasul 5 de mai sus, una din limite se modifică și anume, dacă se modifică limita sup , atunci aceasta *descrescă*, iar dacă se modifică limita inf , atunci aceasta *crescă*. Inițial $inf \leq sup$.

Deoarece la fiecare pas la care nu s-a găsit elementul căutat, una dintre variabilele inf sau sup se modifică și inf crește, iar sup descrește, rezultă că inegalitatea de mai sus nu va mai fi satisfăcută după un număr finit de pași, adică se ajunge ca să fie satisfăcută relația:

inf>*sup*

Aceasta se intimplă cind în tablou nu există un element de valoare *a*. Deci procesul de calcul indicat mai sus se intrerupe fie la găsirea elementului căutat, fie cind se ajunge ca *inf* să-l depășească pe *sup*.

Să reluăm exemplul de mai sus pentru *a*=22. Își în acest caz se va ajunge la subtabloul:

tab[5]=20, tab[6]=21
și

inf=5, *sup*=6

La pasul 3 se determină:

i=[(5+6)/2]=5

Apoi se ajunge la pasul 5, deoarece

tab[5] != 22.

Cum tab[5]=20 < 22, rezultă că *inf*=*i*+1=5+1=6. Se revine la pasul 3 unde se atribuie lui *i* valoarea:

i=[(*inf*+*sup*)/2]=[(6+6)/2]=6

Se ajunge din nou la pasul 5, deoarece

tab[6] != 22.

Cum tab[6]=21 < 22, se modifică din nou *inf*:

inf=*i*+1=7

În acest moment *inf*>*sup* și deci procesul de calcul se intrerupe, concluzia fiind aceea că tabloul respectiv nu conține un element de valoare *a*.

FUNCȚIA BIV42

```

int dcautbin(double tab[], int n, double a)
/* cauta in tab un element de valoare egala cu a;
   returnaza indexele elementului respectiv sau -1, daca nu exista un astfel de element */
{
    int i, inf, sup;

    inf = 0;
    sup = n-1;
    while(inf <= sup) {
        i = (inf+sup)/2;
        if(tab[i] == a) return i;
        if(tab[i] > a) sup = i-1;
        else
            inf = i+1;
    } /* sfarsit while */
    /* se ajunge aici cind inf>sup, ceea ce inseamna ca in tab nu exista

```

```

        un element de valoare a */
    return -1;
}

```

- 4.43 Să se scrie un program care citește un sir de intregi separati prin caractere albe și-i afișeză pe cei care sunt numere prime și sunt în intervalul (0,1000). După ultimul număr se tastează un caracter nenumeric.

Acest program generează la început un tablou cu numere prime. În acest scop se realizează o funcție pe care o numim *prim*. Ea are doi parametri: *tprim* și *n*. Primul parametru este un tablou de tip *int*, ale cărui elemente sunt numerele prime care nu-l depășesc pe *n*:

tprim[0]=1, *tprim*[1]=2, *tprim*[2]=3, *tprim*[3]=5 etc.

Ei este sortat crescător și ultimul element *tprim*[*m*] este cel mai mare număr prim care nu-l depășește pe *n*:

tprim[*m*] <= *n*.

Funcția *prim* returnează numarul numerelor prime păstrate în *tprim* (adică *m*+1).

Pentru determinarea numerelor prime vom proceda ca mai jos:

Primele 3 numere prime (1, 2 și 3) se păstrează în mod automat, dacă *n*>3.

Pentru a determina că un număr *m* > 2 este prim, este suficient să stabilim că el nu este divizibil cu nici unul din numerele prime mai mici decit el. De fapt, este suficient să testăm divizibilitatea lui cu numerele prime ale căror pătrate nu-l depășesc. De asemenea, vom observa că singurul număr prim par este 2. Deci un număr *m* > 2 este prim dacă:

1. *m* este impar.
2. *m* nu se divide cu nici unul din numerele prime consecutive *p*₁, *p*₂, ..., *p*_{*k*}, ale căror pătrate nu-l depășesc pe *m* (*p*₁=3).

Deci *p*_{*k*} este cel mai mare număr prim pentru care *p*_{*k*}**p*_{*k*} < *m*.

Dacă tabloul conține deja *i* numere prime:

tprim[0], *tprim*[1], ..., *tprim*[*i*-1] (*i* > 2)

atunci pentru a determina numărul prim următor vom proceda astfel:

1. *tprim*[*i*]=*tprim*[*i*-1]+2, deoarece numerele prime mai mari decit doi nu sunt pare.
2. Se testează dacă *tprim*[*i*] este prim; în acest scop se caută dacă există printre numerele prime:

tprim[2]=3, *tprim*[3], ...

vreunul care să-l dividă pe *tprim*[*i*].

Conform celor spuse mai sus, nu trebuie incercate toate elementele tabloului *tab* ci numai acele elemente care satisfac relația:

(1) *tprim*[*j*]**tprim*[*j*] <= *tprim*[*i*] (*j*=2, 3, ...).

În cazul în care există un element care satisfacă atât relația (1), cit și relația:

(2) $tprim[i] \% tprim[j] == 0$

numărul $tprim[i]$ nu este prim și se va trece la pasul 3 de mai jos.

Altfel $tab[i]$ este prim, se mărește i cu o unitate și procesul de calcul se reia de la punctul 1.

3. Cum $tprim[i]$ nu este prim, acesta se mărește cu 2 și procesul se reia de la punctul 2.

Procesul de calcul de mai sus se continuă atât timp cît $tprim[i] \leq n$.

După generarea numerelor prime, programul citește un întreg și dacă acesta aparține intervalului $(0,1000)$, îl caută în tabela de numere prime. Apoi se afișează numărul respectiv insotit de un mesaj corespunzător. După afișare se revine la citirea numărului următor cu care se procedează analog. Execuția programului se întrerupe la întâlnirea unui caracter care nu aparține unui întreg.

Căutarea numărului citit în tabloul de numere prime se face prin metoda căutării binare. În acest scop se utilizează o funcție analogă cu funcția *dcautbin* definită în exercițiul precedent. În acest caz se va folosi funcția *icautbin*. Diferența între cele două funcții constă în aceea că *dcautbin* căută o valoare de tip *double* într-un tablou de tip *double*, iar *icautbin* căută o valoare de tip *int* într-un tablou de tip *int*.

PROGRAMUL BIV43

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

#define MAX 1000

int icautbin(int tab[], int n, int a)
/* cauta în tab un element de valoare egală cu a;
   returnează indicele elementului respectiv sau -1, dacă nu există un astfel de element */
{
    int i, inf, sup;

    inf = 0;
    sup = n-1;
    while(inf <= sup) {
        i = (inf+sup)/2;
        if(tab[i] == a) return i;
        if(tab[i] > a)
            sup = i-1;
        else
            inf = i+1;
    } /* sfîrșit while */

    return -1;
} /* sfîrșit icautbin */

int prim( int tprim[], int n)
/* păstrează în tprim numerele prime care nu-l depășesc pe n */
```

```
{
    int i, j, k;

    tprim[0] = 1;
    if( n <= 1 ) return 1;
    tprim[1] = 2;
    if( n <= 2 ) return 2;
    tprim[2] = 3;
    if( n <= 3) return 3;
    tprim[3] = 5;
    i = 3;

    while(tprim[i] <= n) {
        /* se cauta existenta unui divizor prim >= 3 pentru tprim[i] */
        j = 2;
        while((k=tprim[j]*tprim[j]) < tprim[i] &&
              tprim[i] % tprim[j])
            /* ciclul continua atât timp cît tprim[j]*tprim[j] < tprim[i] și restul împărțirii lui tprim[i]
               la tprim[j] este diferit de zero */
            j++;

        /* ciclul while se termină cind cel puțin una din condițiile de mai sus nu este adevarată */
        if( k > tprim[i] )
            /* tprim[i] este prim deoarece nu există un divizor prim a lui tprim[i] al căruia patrat să fie
               mai mic sau egal cu tprim[i] */
            i++;
        tprim[i] = tprim[i-1] + 2;

        /* tprim[i-1] fiind prim, urmatorul număr prim va fi cel puțin cu 2 mai mare decât el */

    } /* sfîrșit if pentru tprim[i] prim */

    else /* tprim[i] nu este prim deoarece s-a determinat un j asa incit
          tprim[j]*tprim[j] = tprim[i]
          sau restul împărțirii lui tprim[i] la tprim[j] este zero;
          în ambele cazuri tprim[i] nu este prim;
          se mărește cu 2 */

        tprim[i] += 2;

    } /* sfîrșit while exterior */
    return i;
} /* sfîrșit prim */

main() /* citește un sir de intregi și-i afișează pe cei din intervalul
         (0,1000), care sunt numere prime */
{
    int nrprime[MAX];
    int nrcrt;
    int n;
```

```

n=prim(nr_prime, MAX);

/* n este numărul numerelor prime care nu-l depășesc pe 1000 */
do {
    printf("Tastati un întreg de cel mult 3 cifre:");
    if (scanf ("%d", &nrcrt) != 1) exit(0);
    if (nrcrt < 1 || nrcrt > 3 || nrcrt == 3) {
        printf ("%d este un număr prim din intervalul\n"
               "(0,1000)\n", nrcrt);
        continue;
    }
    if (nrcrt > 3 && nrcrt < 1000) {
        i=j=ntb(mprim, 0, nrcrt) + 1;
        printf ("%d este un număr prim din intervalul\n"
               "(0,1000)\n", nrcrt);
    } else
        printf ("%d nu este un număr prim din\n"
               "intervalul (0,1000)\n", nrcrt);
} while(1);
/* sfîrșit main */

```

Observație:

Condiția

$tprim[j] * tprim[j] < tprim[i]$
poate fi înlocuită cu echivalenta ei:

$tprim[j] < \sqrt{tprim[i]}$

care este mai eficientă pentru numere relativ mari, dacă se calculează rădâcina patrată în afara ciclului *while* interior:

```

while (tprim[i] <= n) {
    j=2;
    k=sqrt(tprim[i]);
    while(tprim[j] < k && tprim[i] % tprim[j])
        j++;
    if(tprim[j] > k)
        ...

```

5. CLASE DE MEMORIE

La compilarea unui text sursă, compilatorul alocă memorie pentru variabilele programului. Se pot aloca variabile pe stivă, în regiștri calculatorului, cit și în alte zone de memorie.

Programatorul definește, cu ajutorul declarațiilor, modul de alocare al variabilelor. Se pot defini variabile utilizabile (*vizibile*) în *tot* programul, precum și variabile care au utilizări *locale* (cu vizibilitate *restrinsă*). Variabilele care pot fi utilizate în *tot* programul se numesc variabile *globale*, iar cele care pot fi utilizate numai într-o anumită parte a programului se numesc variabile *locale*.

5.1. Variabile globale

Variabilele globale au o *definiție* și eventual una sau mai multe declarații de variabilă *externă*.

Definiția unei variabile globale coincide cu o declarație obișnuită care însă este scrisă în afara corpului oricărei funcții a programului.

O astfel de definiție, de obicei, se scrie la începutul unui fișier sursă. Aceasta deoarece ea este valabilă din locul în care este scrisă și pînă la sfîrșitul fișierului sursă respectiv.

În cazul în care programul se compune din mai multe fișiere sursă, o variabilă globală poate fi utilizată într-un fișier sursă în care nu este definită dacă este declarată ca variabilă externă în acel fișier. O declarație de *variabilă externă* coincide cu o declarație de variabilă obișnuită precedată de cuvintul cheie *extern*.

Exemplu:

Un program se compune din două fișiere sursă, *fis1.cpp* și *fis2.cpp*. În fișierul *fis1.cpp* se definesc variabilele globale *zi*, *luna* și *an* de tip *int*. Aceste variabile se utilizează atât în *fis1.cpp*, cit și în *fis2.cpp*. Dacă aceste fișiere se compilează separat, folosind un fișier de tip *project* sau dacă *fis2.cpp* se include în fișierul *fis1.cpp* înainte de a defini variabilele globale respective, atunci ele nu sunt definite în *fis2.cpp* și în consecință nu pot fi utilizate în funcțiile din fișierul *fis2.cpp*. Pentru a elibera acest neajuns, vom declara variabilele respective ca externe în funcțiile din fișierul *fis2.cpp*.

Fișierul *fis1.cpp*

```

int zi,luna,an; /*Definiție de variabile globale. Aceste variabile pot fi folosite în toate
                    funcțiile din fișierul fis1.cpp care urmărează după aceasta definitie*/
...
tip f(...)

{

```

```

int i;
...
i=an % 4==0 && an % 100 != 0 || an % 400==0;
...
if(luna==2)
    zi=28+i;
...
tip g(...)

...
if(zi <= 27)
    zi++;
...
if(zi==31&&luna==12)
    an++;
...

```

Fisierul fis2.cpp

```

...
main()
{
    extern int zi,luna,an; /* declaratie de extern pentru variabilele globale zi, luna si
                           an, definite in fis1.cpp; este valabila in corpul functiei
                           main */

    ...
    if (zi<1||zi>31)      printf("ziua eronata\n");
    if(luna<1||luna>12)   printf("luna eronata\n");
    if(an<1600||an>4900) printf("anul eronat\n");
    ...
} /* sfarsit main */

tip p(...)
{
    ...
    /* nu exista declaratii de extern pentru variabilele globale zi,luna,an; in acest caz
       nu se pot utiliza variabilele respective numai daca fis2.cpp se include in fis1.cpp
       dupa definitia acestor variabile globale */
    ...
    zi=15; /* zi apare la compilare ca nedefinita */
    ...
}

```

Observație:

Întrucit repartizarea funcțiilor în fișiere poate să se modifice pe măsură ce se construiește programul, se recomandă ca orice utilizare a unei variabile globale să fie precedată de declarația de *extern* a ei.

Variabilele globale sunt alocate în momentul compilării. Lor li se alocă memorie, la întlnirea definițiilor lor, într-o zonă de memorie prevăzută special

pentru variabilele globale ale programului.

5.2. Variabile locale

Variabilele locale, spre deosebire de cele globale, nu sunt valabile în tot programul. Ele au o valabilitate locală, în unitatea în care sunt declarate.

Variabilele locale pot fi alocate pe *stivă*. În acest caz ele se numesc *automatice*. Acestea se declară în mod obișnuit, în corpul unei funcții sau la începutul unei instrucțiuni compuse. O astfel de variabilă se alocă la execuție (nu la compilare).

La apelul unei funcții, variabilele automate (declarate în mod obișnuit înaintea primei instrucțiuni din corpul funcției respective) se alocă pe stivă. În momentul în care se revine din funcție, variabilele automate alocate la apel, se dezalocă (elimină) și stivă revine la starea dinaintea apelului (operăția de curățire a stivei). Aceasta înseamnă că variabilele automate își pierd existența la revenirea din funcție în care sunt declarate. De aceea, o variabilă automatică este valabilă (vizibilă) numai în corpul funcției în care a fost declarată.

În același mod se comportă variabilele automate declarate la începutul unei instrucțiuni compuse. O astfel de variabilă se alocă pe stivă în momentul în care controlul programului ajunge la instrucțiunea compusă în care este declarată variabilă respectivă și se elimină de pe stivă în momentul în care controlul programului trece la instrucțiunea următoare celei compuse.

Variabilele locale pot și să nu fie alocate pe stivă. În acest scop ele se declară ca fiind *static*. O declarație de variabilă statică este o declarație obișnuită precedată de cuvintul cheie *static*.

Variabilele statice pot fi declarate atât în corpul unei funcții cât și în afara corpului oricărui funcție.

O variabilă statică declarată în corpul unei funcții este definită numai în corpul funcției respective.

Spre deosebire de variabilele automate, o variabilă statică nu se alocă pe stivă la execuție, ci la compilare într-o zonă de memorie destinată acestora.

O variabilă statică declarată în afara corpurilor funcțiilor este definită (vizibilă) din punctul în care este declarată și pînă la sfîrșitul fișierului sursă care conține declarația respectivă. Spre deosebire de variabilele globale, o astfel de variabilă nu poate fi declarată ca externă. Deci ea nu poate fi utilizată în alte fișiere dacă acestea se compilează separat sau se includ înaintea declarației respective.

Puteți spune că o variabilă statică declarată în afara corpurilor funcțiilor este locală fișierului sursă în care este declarată.

Ea se alocă la compilare într-o zonă specială rezervată variabilelor statice corespunzătoare fișierului sursă în care au fost declarate.

Dacă prin *modul* înțelegem un text sursă compilat separat față de restul textului sursă al unui program, atunci putem afirma că o variabilă statică

declarata in afara corpurilor functiilor este locala modului in care a fost declarata. La aceasta mai trebuie sa adaugam si faptul ca utilizările unei variabile statice trebuie sa fie precedate de declararea ei.

In cazul variabilelor globale, acestea pot fi utilizate in orice modul, daca sunt precedate in modulul respectiv, fie de definitia lor, fie de declararea de extern.

Functiile sunt analoge cu variabilele globale. Ele, ca si variabilele globale, pot fi utilizate (apelate) in orice modul al programului daca apelul lor este precedat, in modulul respectiv, fie de definitia, fie de prototipul lor. In acest caz, prototipul functiei inlocuieste declararea de *extern* a variabilei globale.

Limbajele C si C++ permit declarari de functii statice. Aceasta se face precedind antetul functiei de cuvintul cheie *static*. In acest caz, functia respectiva este locala modului in care este definita.

Impartirea pe module a unui program nu se face prin reguli definite riguros.

Se obisnuieste sa se spuna ca un modul este format din functii "inrudite" si contine date pe care acestea le prelucraza in comun.

Exemplu:

Un program se compune din 2 fisiere *sursa1.cpp* si *sursa2.cpp* care formeaza fiecare cte un modul (se compileaza separat, utilizand, de exemplu, un fisier de tip *project*).

Fisierul sursa1.cpp contine:

```
int i, j, k; /* - definitie de variabile globale;
   - se pot utiliza in tot modulul;
   - se pot declara ca externe in celalalt modul pentru a putea fi utilizate si acolo */

static double x, y; /* - declaratie de variabile statice;
   - sunt utilizabile in tot modulul de fata si numai in acesta */

...

main()
{
    int p, q; /* - declaratie de variabile automatice;
   - sunt utilizabile numai in corpul functiei main */

    static int u, v, r; /* - declaratie de variabile statice;
   - sunt utilizabile numai in corpul functiei main */

    ...
    /* se pot utiliza variabilele i, j, k, x, y, p, q, u, v si r */
}

tip f(...)
{
    float a, b; /* - declaratie de variabile automatice;
   - sunt utilizabile numai in corpul functiei f */
```

```
static char c, temp; /*-declaratie de variabile statice;
   - sunt utilizabile numai in corpul functiei f */
...
/* se pot utiliza variabilele i, j, k, x, y, a, b, c si temp */
}
```

Fisierul sursa2.cpp contine:

```
static float s, t; /*- declaratie de variabile statice;
   - sunt utilizabile in fisierul sursa2.cpp */
tip g(...)
{
    extern int i, j, k; /* -declaratie de extern pentru variabilele i, j, k;
   - sunt utilizabile in corpul functiei g */

    char d, e; /* - declaratie de variabile automatice;
   - sunt utilizabile in corpul functiei g */

    static double d1, d2, d3; /*- declaratie de variabile statice;
   - sunt utilizabile in corpul functiei g */
...
/* se pot utiliza variabilele i, j, k, s, t, d, e, d1, d2 si d3 */
}

tip h(...)
{
    extern int k; /* -declaratie de extern pentru variabila k;
   - este utilizabila in corpul functiei h */

    ...
/* se pot utiliza variabilele k, s si t */
}
```

Variabilele pot fi *redeclareate* indiferent de modul lor de alocare (global, static sau automatic).

Exemplu:

```
int i, c; /* variabile globale */
static float a; /* variabila statica */
...
main()
{
    double c; /* variabila automatica diferita de variabila globala c */
    char a; /* variabila automatica diferita de variabila statica a */
    ...
    c=123; /* se atribuie variablei automaticce c valoarea 123 dupa
   conversia ei in flotanta dubla precizie */
```

```

... i=123; /* se atribuie variabilei globale i valoarea 123 */
...
a='1'; /* se atribuie variabilei automatee a codul ASCII al caracterului 1 */

/* - in aceasta functie nu se pot utiliza variabila globala c si variabila
   statica a deoarece acestea au fost redeclarate;
   - cu toate acestea, in limbajul C++ este posibila utilizarea lor folosind
     operatorul :: pentru globale. */
}

tip f(...)

{
    ...
    /* nu sunt redeclarari pentru i,c si a */

    c=123; /* se atribuie variabilei globale c valoarea 123 */
    a=123; /* se atribuie variabilei statice a valoarea 123, dupa
              ce a fost convertita spre floatant simpla precizie */
    ...
}

```

Uneori, tablourile de dimensiuni mari se alocă globale sau statice deoarece alocarea lor pe stivă poate conduce la depășirea stivei.

5.3. Alocarea parametrilor

Parametrii formali se alocă pe stivă ca și variabile automate. De aceea, ei se consideră a fi variabile locale și sunt utilizabili numai în corpul funcției în antetul căreia sunt declarati.

La apelul unei funcții, se alocă pe stivă parametri formalii, dacă există, li se atribuie valorile parametrilor efectivi care le corespund. Apoi se alocă pe stivă variabilele automate declarate la începutul corpului funcției respective.

La revenirea din funcție, se realizează curățirea stivei, adică sunt eliminate de pe stivă (dezalocate) atât variabilele automate, cât și parametri. În felul acesta, la revenirea din funcție, stiva ajunge la starea dinaintea apelului.

5.4. Utilizarea parametrilor și a variabilelor globale

Am văzut că parametrii efectivi, se utilizează pentru a concretiza valorile care intervin în procesul de calcul definit printr-o funcție. Se obișnuiește să se spună că valorile respective se "transferă" funcției prin intermediul parametrilor.

Din această cauză, parametri se mai consideră ca fiind o "interfață" între funcții. În limbajul C această interfață este unilaterală, adică ea transferă date de la funcția care face apelul, la funcția apelată.

Variabilele globale reprezintă și ele o interfață între funcții. În acest caz nu se transferă date de la o funcție la alta, ci orice funcție are acces la valorile variabilelor globale. Dacă o funcție trebuie să transfere o valoare funcției apelate, atunci aceasta are două posibilități:

1. Să folosească un parametru la care i se atribuie valoarea respectivă înainte de apel.
2. Să atribuie valoarea respectivă unei variabile globale înainte de apel.

În primul caz, funcția apelată are destinat un parametru formal la care i se atribuie în mod automat valoarea respectivă la apel. În al doilea caz, funcția apelată trebuie să aibă acces la variabila globală căreia i s-a atribuit valoarea de transferat. În acest scop, este necesar să se cunoască *numele* variabilei globale respective în momentul în care se programează funcția respectivă. Aceasta este un neajuns al variabilelor globale față de parametri. Numele și sensul variabilelor globale trebuie să fie cunoscut înainte de a începe programarea diferitelor funcții care utilizează în comun variabilele respective.

Un alt dezavantaj al variabilelor globale este sursa mare de erori pe care o reprezintă. Într-adevăr, orice funcție din program având acces la o variabilă globală, ea poate modifica valoarea acesteia și de aceea posibilitatea modificării eronate este mare. De asemenea, depistarea unei astfel de erori este destul de complicată, deoarece este necesar să se studieze un număr mare de funcții (toate funcțiile care au acces la variabila globală a cărei valoare s-a constat că este eronată).

În cazul utilizării parametrilor, se realizează o protecție a datelor, deoarece funcția apelată nu poate modifica valorile parametrilor efectivi în mod direct.

Avantajul variabilelor globale decurge din faptul că ele reprezintă o interfață simplă între funcții, interfață care este în ambele sensuri. O funcție poate modifica valoarea unei variabile globale, modificare care rămâne valabilă la revenirea din ea, deci funcția care a facut apelul poate beneficia de modificarea făcută prin intermediul funcției apelate. Dar chiar acest fapt se consideră ca este o sursă de erori. De aceea, se consideră că nu este bine să se exagereze cu utilizarea variabilelor globale, ele constituind o sursă de erori.

De obicei, folosim variabilele globale cind rezultatele unei funcții sunt folosite în comun de mai multe funcții ale programului. În rest, se recomandă utilizarea parametrilor pentru realizarea interfețelor dintre funcții.

5.5. Variabile registru

O variabilă alocată într-un registru al calculatorului se numește *variabilă regisztr*. Ea se declară printre o declarație obișnuită precedată de cuvintul cheie *register*.

Se pot declara, ca variabile regisztr, numai variabilele de tip *int*, *char* și *pointer* (acest tip va fi introdus într-un capitol următor). De asemenea, numai

parametrii și variabilele automate simple pot fi declarate ca variabile registru.

Numai un numar limitat de variabile se pot aloca în regiștri. Alocarea se face la apelul unei funcții și în ordinea în care au fost declarate aceste variabile în funcția respectivă. Alocarea ramâne valabilă pînă la revenirea din funcție. În cazul în care nu pot fi alocate în regiștri toate variabilele registru declarate de programator, se alocă cîte se pot în ordinea declarării lor, iar restul se alocă în mod obișnuit pe stiva, ca orice variabilă automatică sau parametru.

Se recomandă să se declare ca variabile registru, acele variabile ale unei funcții care au o utilizare mare în funcția respectivă. Alocarea unei variabile în registru conduce la economie de memorie, precum și la creșterea vitezei de calcul.

Menționăm că, în lipsa declarațiilor registru, compilatorul aloca în mod implicit anumite variabile automate sau parametri în regiștri.

În legătura cu variabilele registru amintim că lor nu li se poate aplica operatorul adresa (&).

Exerciții:

5.1 Să se scrie o funcție care citește:

- valoarea variabilei m de tip int;
- valoarea variabilei n de tip int;
- $m \times n$ numere care reprezintă elementele unei matrice de ordinul $m \times n$.

Funcția are următorii parametri:

- | | |
|-------------|---|
| <i>dmat</i> | - Tablou care păstrează elementele matricei citite. |
| <i>max</i> | - Maximul produsului $m \times n$ admis. |

Funcția returnează produsul $m \times n$ și atribuie valorile lui m și n , variabilelor globale *nrlin*, respectiv *nrcol*.

Tabloul *dmat* este unidimensional, matricea pastrindu-se prin *liniarizare*.

Dacă notăm cu $a[i,j]$, $i=0,1,\dots,m-1$; $j=0,1,\dots,n-1$; elementele matricei, atunci elementul $a[i,j]$ se atribuie lui *dmat*[k], unde:

$$(1) k = i \times n + j$$

Această relație poartă denumirea de relație de *liniarizare*. Ea permite pastrarea pe linii a elementelor matricei citite. Astfel, prima linie a matricei este formată din elementele $a[0,j]$, pentru $j=0,1,\dots,n-1$. Conform relației (1), aceste elemente se păstrează în tabloul *dmat* prin intermediul elementelor:

dmat[0], *dmat*[1], ..., *dmat*[$n-1$].

Linia a doua, adică elementele $a[1,j]$, pentru $j=0,1,\dots,n-1$, se păstrează în tabloul *dmat* ca elemente ce au indicei $k=1 \times n + j$, adică:

dmat[n], *dmat*[$n+1$], ..., *dmat*[$n+n-1$].

În general, linia formată din elementele $a[i,j]$ pentru $j=0,1,\dots,n-1$ se păstrează în tabloul *dmat* ca:

dmat[$i \times n$], *dmat*[$i \times n + 1$], ..., *dmat*[$i \times n + n - 1$]

Funcția de față folosește funcția *ndcit* definită în exercițiul 4.37.

FUNCȚIA BV1

```
int gcditmat(double dmat[], int max)
/* citește
   m - numar de linii;
   n - numar de coloane;
   m*n - numere de tip double pe care le păstreaza in matricea dmat prin liniarizare;
returneaza valoarea m*n;
realizeaza atribuirile:
   nrlin=m;
   nrcol=n; */
{
extern int nrlin,nrcol;
int i,m,n;
char t[255];

do { /* se citesc valorile lui m si n */
    do { /* citeste pe m */
        printf("numarul de linii= ");
        if(gets(t) == NULL ) {
            printf("s-a tastat EOF\n");
            exit(1);
        }
        if(sscanf(t,"%d", &m) == 1 && m > 0 && m <= max)
            break;
        printf("nu s-a tastat un intreg in\
               intervalul [1,%d]\n",max);
    } while(1);
    do { /* citeste pe n */
        printf("numarul de coloane= ");
        if(gets(t) == NULL ) {
            printf("s-a tastat EOF\n");
            exit(1);
        }
        if(sscanf(t,"%d", &n) == 1 && n > 0 && n <= max )
            break;
        printf("nu s-a tastat un intreg in\
               intervalul [1,%d]\n",max);
    } while(1);
    i = m*n;
    if(i <= max) break;
    printf("produsul m*n=%d depaseste pe max=%d\n", i, max );
    printf("se reiau citirile lui m si n\n");
} while(1);

/* sc citese elementele matricei tasteate pe linii */
if(ndcit(i,dmat) != i ) {
```

```

        printf("nu s-au tastat %d elemente\n", i);
        exit(1);
    }
    nrlin = m; nrcol = n; return i;
}

```

5.2 Să se scrie o funcție care calculează produsul dintre o matrice de ordinul $m \times n$ și o matrice coloană (de ordinul $n \times 1$).

Funcția are următorii parametri:

- | | |
|----------------|---|
| <i>m</i> | - Numărul liniilor matricei. |
| <i>n</i> | - Numărul coloanelor matricei. |
| <i>dmat</i> | - Tablou unidimensional de tip <i>double</i> care păstrează elementele matricei de ordinul $m \times n$ prin liniarizare. |
| <i>dmatcol</i> | - Tablou unidimensional de tip <i>double</i> care păstrează elementele matricei coloana. |
| <i>dmatrez</i> | - Tablou unidimensional de tip <i>double</i> care păstrează elementele matricei rezultat. |

Matricea rezultat este de ordinul $m \times 1$. Elementele ei se calculează prin relația de mai jos:

$$dmatrez[i] = a[i,0]*dmatcol[0]+a[i,1]*dmatcol[1]+\dots+a[i,n-1]*dmatcol[n-1]$$

pentru $i=0,1,\dots,m-1$, unde prin $a[i,j]$ am notat un element al matricei de ordinul $m \times n$. Înind seama de relația de liniarizare, suma de mai sus se scrie astfel:

$$\begin{aligned} dmatrez[i] &= dmat[i*n]*dmatcol[0] + \\ &+ dmat[i*n+1]*dmatcol[1] + \dots + dmat[i*n+n-1]*dmatcol[n-1]. \end{aligned}$$

FUNCȚIA BV2

```

void pmatcol(int m, int n, double dmat[],
             double dmatcol[], double dmatrez[])
{
    /* calculeaza produsul dintre matricea dmat de ordinul m*n si vectorul coloana dmatcol de ordinul n*1; se obtine matricea coloana dmatcol de ordinul m*1 */
    {
        int i,j,k;

        for(i=0; i < m; i++) {
            dmatrez[i] = 0;
            k=i*n;
            for(j=0; j <= n-1; j++)
                dmatrez[i] += dmat[k+j] * dmatcol[j];
        }
    }
}

```

5.3 Să se scrie un program care citește elementele unei matrice *a* de ordinul $m \times n$, elementele unei matrice coloane *b* de ordinul $n \times 1$, calculează și afișează

produsul $a \cdot b$.

În acest scop se vor apela trei funcții:

- funcția *gdcitmat* care citește elementele matricei *a* (vezi exercițiul 5.1.);
- funcția *dvcit* care citește elementele matricei *b* (vezi exercițiul 4.38.);
- funcția *pmatcol* care calculează produsul $a \cdot b$ (vezi exercițiul 5.2.).

Deoarece funcțiile *gdcitmat* și *dvcit* apelează funcția *ndcit*, se va include și fișierul care conține definiția acestei funcții (vezi exercițiul 4.37.).

Matricea *a* va fi precedată de doi întregi care reprezintă valorile lui *m* și *n*.

Matricea *b* va fi precedată de un întreg care reprezintă valoarea lui *n*. Această valoare, deși pare că nu este necesară înaintea elementelor matricei *b*, ea se tastează pe de o parte pentru a face o verificare cu privire la numărul datelor tastate, iar pe de altă parte, prezența ei este cerută de funcția *dvcit*.

PROGRAMUL BV3

```

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include "biv37.cpp" /* functia ndcit */
#include "biv38.cpp" /* functia dvcit */
#include "bv1.cpp"   /* functia gdcitmat */
#include "bv2.cpp"   /* functia pmatcol */

int nrlin,nrcol; /* variabile globale utilizate la citirea matricei
                    cu ajutorul functiei gdcitmat */

#define MAX 1000

main() /* citește elementele matricei a de ordinul m*n si elementele
          matricei b de ordinul n*1, calculeaza si afiseaza produsul c = a*b */
{
    int m;
    double a[MAX],b[MAX],c[MAX];

    gdcitmat(a,MATX); /* citește matricea a; la revenire, nrlin are ca valoare numarul
                         liniilor (m), iar nrcol numarul coloanelor(n) */

    /* citește matricea b */
    if(dvcit(MAX,b) != nrcol) {
        printf("matricea coloana nu are %d linii\n",nrcol);
        exit(1);
    }

    /* calculeaza produsul c = a*b */
    pmatcol(nrlin,nrcol,a,b,c);

    /* listeaza matricea rezultat */
    for(m = 0; m < nrlin; m++) {
        printf("c[%d]=%g\n",m,c[m]);
        if((m+1)%23 == 0) {
            printf("actionati o tasta pentru a continua\n");
        }
    }
}

```

```

        getch();
    }
}

```

- 5.4 Să se scrie o funcție care calculează produsul a două matrice cu elemente de tip *double*.

Fie *a* o matrice de ordinul $m \times n$ și *b* o matrice de ordinul $n \times s$. Matricea:

$$c = a * b$$

este de ordinul $m \times s$. Elementele ei se calculează folosind relația:

$$c[i,j] = a[i,0]*b[0,j] + a[i,1]*b[1,j] + \dots + a[i,n-1]*b[n-1,j]$$

pentru $i=0,1,2,\dots,m-1$ și $j=0,1,2,\dots,s-1$.

Elementele celor trei matrice se păstrează liniarizat în tablourile unidimensionale *dmatr*, *dmatb* și *dmatc*. Folosind relația de liniarizare, suma de mai sus se reprezintă astfel:

$$\begin{aligned} dmatc[i*s+j] &= dmatr[i*n]*dmatb[j] + dmatr[i*n+1]*dmatb[s+j] + \dots + \\ &\quad dmatr[i*n+k]*dmatb[k*s+j] + \dots + \\ &\quad dmatr[i*n+n-1]*dmatb[(n-1)*s+j] \end{aligned}$$

FUNCȚIA BV4

```

void dprodmat(int m, int n, int s, double dmatr[],
              double dmatb[], double dmatc[])
{
    /* calculeaza produsul matricelor a si b:
       c = a*b;
       dmatr pastreaza matricea a de ordinul m*n;
       dmatb pastreaza matricea b de ordinul n*s;
       dmatc pastreaza matricea c de ordinul m*s */
    {
        int i, j, k;
        int in, is;

        for(i=0; i < m; i++) {
            is=i*s;
            in= i*n;
            for(j=0; j < s; j++) {
                dmatc[is+j] = 0;
                for(k=0; k < n; k++)
                    dmatc[is+j] += dmatr[in+k]*dmatb[k*s+j];
            }
        }
    }
}

```

- 5.5 Să se scrie un program care citește elementele a două matrice de tip *double*, calculează și afișează produsul lor.

Programul utilizează funcția *gdcitmat* pentru a citi elementele celor două matrice. De aceea, elementele matricelor vor fi precedate de ordinul lor (numărul

de linii și numărul coloanelor - vezi exercițiul 5.1.).

PROGRAMUL BV5

```

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include "biv37.cpp" /* functia ndcit */
#include "bvl.cpp"   /* functia gdcitmat */
#include "bv4.cpp"   /* functia dprodmat */

#define MAX 1000

int nrlin, nrcol;

main() /* citeste elementele a două matrice, calculeaza si afiseaza matricea produs */
{
    int m, n, s;
    double a[MAX], b[MAX], c[MAX];
    int i, j, is, k;

    /* citeste matricea a */
    gdcitmat(a, MAX);

    m=nrlin; /* numarul liniilor matricei a */

    n=nrcol; /* numarul coloanelor matricei a */

    /* citeste matricea b */
    gdcitmat(b, MAX);
    if(nrlin != n) {
        printf("matricea a are %d coloane\n", n);
        printf("matricea b are %d linii\n", nrlin);
        exit(1);
    }

    s=nrcol; /* nr coloanelor matricei b */

    /* se realizeaza produsul c = a*b */
    dprodmat(m, n, s, a, b, c);

    /* afiseaza elementele matricei produs */
    k=0;
    for(i=0; i < m; i++) {
        printf("\n\tlinia nr.%d\n", i+1);
        is=i*s;
        k++;
        for(j=0; j < s; j++) {
            printf("%g ", c[is+j]);
            if(j%5 == 4) {
                /* afiseaza 5 elemente pe un rind */
                printf("\n");
                k++; /* numara rindurile */
            }
        }
    }
}

```

```
if(k==23) {  
    printf("actionati o tasta pentru a continua\n");  
    getch();  
    k=1;  
}  
}  
}
```

6.INIȚIALIZARE

Adesea dorim ca unele variabile din program să aibă valori inițiale. Acest lucru este posibil și anume, variabilelor globale li se pot da valori inițiale la definirea lor, iar celelalte clase de variabile pot fi inițializate la declararea lor.

6.1. Inițializarea variabilelor simple

O variabilă simplă poate fi inițializată printr-o construcție de forma:

tip nume=expresie;

sau

static tip nume=expresie;

dacă variabila este statică.

Prima formă corespunde variabilelor automatic sau globale.

Variabilelor globale sau statice li se atribuie valori inițiale la compilare, deci ele au valorile respective la lansarea programului. Faptul că aceste variabile se inițializează la compilare, cere ca expresiile utilizate pentru inițializare să fie constante.

Variabilele automatic simple se inițializează la execuție, de fiecare dată cind ele se alocă pe stivă, adică la apelul funcției care conține declarațiile lor. În acest caz, expresiile utilizate pentru inițializare pot să nu fie constante. Cu toate acestea, operanții variabili ai unei astfel de expresii trebuie să aibă valori definite în prealabil. În caz contrar, expresia de inițializare nu poate fi evaluată în momentul în care se aloca pe stivă variabila automatică pe care o inițializează.

Variabilele globale și statice neinițializate au în mod implicit valoarea *zero*. Acest lucru nu mai are loc în cazul variabilelor automatic.

O variabilă automatică neinițializată are o valoare imprevizibilă în momentul apelului funcției în al cărui corp este declarată. Aceasta rezultă din faptul că ea se alocă pe stivă și în lipsa expresiei de inițializare în declarația sa, ei nu își se atribuie nici o valoare. În felul acesta, valoarea ei rămîne nedefinită pînă în momentul în care își se atribuie o valoare printr-o instrucție de atribuire.

În toate cazurile se fac conversii dacă tipul expresiei de inițializare nu coincide cu tipul variabilei pe care o inițializează.

Exemple:

1. int n=100; /*- definiție de variabila globală;
- lui n îi se atribuie valoarea 100;
- la lansarea programului n are valoarea 100. */

```

2. double x=3; /*- definitie de variabila globala;
   - lui x i se atribuie valoarea 3 dupa ce se converteste spre double;
   - la lansarea programului x are valoarea 3. */

3. #define MAX 100
int i = MAX*2; /* - i este variabila globala initializata cu valoarea 200;
   - la lansarea programului i are valoarea 200. */

4. int k; /*- definitie de variabila globala;
   - la lansarea programului k are valoarea zero. */

5. static char c='a'; /*- declaratie de variabila statica;
   - lui c i se atribuie codul ASCII al caracterului a(97);
   - la lansarea programului c are valoarea 97. */

6. static int s; /*- declaratie de variabila statica;
   - la lansarea programului s are valoarea zero. */

7. static int p='a'+1; /*- declaratie de variabila statica;
   - la lansarea programului s are ca valoare
   codul ASCII al literii b (97+1). */

8. tip f(int n)
{
    int x=123; /*- declaratie de variabila automatica;
   - la fiecare apel al functiei f, x se aloca pe stiva si i se atribuie valoarea 123. */
    int a=x+n; /*- declaratie de variabila automatica;
   - la fiecare apel al functiei f, a se aloca pe stiva si i se atribuie
   valoarea expresiei x+n;
   - la intilnirea declaratiei ambii operanzi ai expresiei x+n sunt definiti. */
}

```

Exerciții:

- 6.1 Să se scrie o funcție care are ca parametri doi intregi x și y , calculează și returnează numărul aranjamentelor de x obiecte luate cîte y .

Funcția de față este analogă funcției definite în exercițiul 4.29.

FUNCȚIA BVI1

```

double aranjamente ( int x, int y)
/* calculeaza si returneaza numarul aranjamentelor de x obiecte luate cîte y */
{
    double a=1.0;
    int i=x-y+1;

    if(x < 1 || x > 170 ) return -1.0;
    if(y < 1 || y > x ) return -1.0;
    while( i <= x ) a *= i++;
    return a;
}

```

}

Observație:

Pentru a testa funcția de față se poate utiliza programul din exercițiul 4.30. Programul respectiv apelează funcția *aranjamente* definită în exercițiul 4.29.

Pentru a apela funcția *aranjamente* definită mai sus, în programul din exercițiul 4.30. se va schimba construcția

```
#include "BIV29.CPP"
```

cu

```
#include "BVI1.CPP"
```

6.2. Inițializarea tablourilor

Tablourile, ca și variabilele simple, pot fi inițializate. Tablourile *globale* se inițializează prin *definițiile lor*. Tablourile *static* și *automatice* se inițializează prin *declarațiile lor*.

În limbajul C, în toate cazurile, inițializările se fac prin expresii *constante*.

Tablourile *globale* și *static* se inițializează la *compilare*. De aceea, la lansarea programului, elementele lor au ca valori, valorile expresiilor cu care au fost inițializate.

Tablourile *automatice* se inițializează la *execuție*, de fiecare dată cînd se apelează funcția în care sunt declarate. Menționăm că în unele versiuni ale limbajului C nu se pot inițializa tablourile automatice.

Limbajul Turbo C permite inițializarea tablourilor automatice.

Inițializarea unui tablou unidimensional se realizează printr-o construcție de forma:

```
tip nume[ec] = {ec0,ec1,...,eci};
```

sau

```
static tip nume [ec] = {ec0,ec1,...,eci};
```

dacă tabloul este static.

Prin ec, ec0, ec1, ..., eci am notat expresii constante.

Prin aceste construcții, elementul *nume[k]* se inițializează cu valoarea expresiei constante eck, pentru $k=0,1,2,\dots,i$.

În general, $i \leq ec-1$. Dacă $i \geq ec$, atunci construcția este eronată. Dacă $i < ec-1$, atunci elementele

```
nume[i+1], nume[i+2], ..., nume[ec-1]
```

rămân neinițializate.

Elementele *neinitializate* ale unui tablou *global sau static* au în mod implicit valoarea inițială egală cu *zero*.

Elementele *neinitializate* ale unui tablou *automatic* au valori inițiale *nedefinite*.

În cazul în care se inițializează toate elementele unui tablou unidimensional, se poate omite expresia din parantezele pătrate care definește numărul elementelor tabloului. Deci construcțiile:

`tip nume[] = {ec0,ec1,...,ecn};`

și

`static tip nume[] = {ec0,ec1,...,ecn};`

sunt corecte și în ambele cazuri tablourile au $n+1$ elemente, iar elementul

`nume[i]`

este inițializat cu valoarea expresiei *eci*.

Menționăm că, la fel ca și în cazul variabilelor simple, dacă expresia de inițializare are un tip diferit de cel al tabloului, atunci valoarea ei se convertește spre tipul tabloului înainte de a fi atribuită elementului pe care-l inițializează.

Observație:

Într-o declarație sau definiție de tablou unidimensional se poate omite expresia care definește numărul elementelor tabloului în următoarele cazuri:

1. Definiția sau declarația de tablou conține expresii constante pentru inițializarea fiecărui element al tabloului.
2. Declarația se referă la un tablou unidimensional care este parametru formal.
3. Declarația de tablou extern unidimensional:
`extern int tab[];`

Exemple:

1. `int tab[10] = {0,1,2,3,4,5,6,7,8,9};`
Prin această definiție/declarație, lui `tab[i]` i se atribuie valoarea *i*.

2. `int tab[] = {0,1,2,3,4,5,6,7,8,9};`
Această construcție este identică, ca efect, cu cea precedentă.

3. `double d[20] = {0,1,-1,3,-2};`
Primele 5 elemente ale tabloului *d* se inițializează astfel:

`d[0]=0;`
`d[1]=1;`
`d[2]=-1;`
`d[3]=3;`
`d[4]=-2.`

Celelalte elemente ale tabloului `d[5], d[6], ..., d[19]` au valoarea inițială zero

dacă *d* este un tablou global și o valoare inițială imprevizibilă dacă tabloul este automatic.

4. `#define V ('A'-10)`

`static int chexa[] = {'A'-V, 'B'-V, 'C'-V, 'D'-V, 'E'-V, 'F'-V};`

Elementele tabloului *chexa* se inițializează astfel:

`chexa[0]='A'-(A'-10)=10`

`chexa[1]='B'-(A'-10)=11`

`chexa[2]=12`

`chexa[3]=13`

`chexa[4]=14`

`chexa[5]=15.`

5. `char er[] = {'e', 'r', 'o', 'a', 'r', 'e', '\0'};`

Tabloul are 7 elemente care se inițializează astfel:

`er[0]='e'`

`er[1]='r'`

`er[2]='o'`

`er[3]='a'`

`er[4]='r'`

`er[5]='e'`

`er[6]='\0'`

Se observă că tabloul *er* păstrează sirul de caractere "eroare", inclusiv caracterul *NUL* care termină orice sir de caractere.

Având în vedere faptul că inițializarea tablourilor de tip *char* se utilizează frecvent, autorii limbajului C au introdus o simplificare pentru inițializarea tablourilor de acest tip. Astfel, pentru un tablou de tip *char* se pot utiliza una din construcțiile:

`char nume[ec]=sir;`

sau

`static char nume[ec]=sir;`

unde prin *sir* se înțelege o succesiune de caractere delimitată prin ghilimele. Prin aceste construcții, elementele tabloului *nume* se inițializează cu codurile ASCII ale caracterelor din compunerea sirului de caractere *sir*, iar după ultimul caracter al sirului se memorează caracterul *NUL*. Expresia constantă *ec*, dacă este prezentă, atunci ea este cel puțin egală cu numărul caracterelor proprii ale sirului mărit cu unu.

Din cele de mai sus rezultă că inițializarea tabloului *er* se poate realiza folosind construcția:

`char er[]="eroare";`

Tablourile multidimensionale pot fi și ele inițializate prin construcții analoge. Așa de exemplu, un tablou bidimensional se poate inițializa printr-o construcție

de forma:

```
tip nume [n][m]={  
    /ec11,ec12,...,ec1m/,  
    /ec21,ec22,...,ec2m/,  
    ...  
    /ecn1,ecn2,...,ecnm/  
};
```

unde:

n, m, ecij – Pentru $i=1,2,\dots,n$ și $j=1,2,\dots,m$ sunt expresii constante.

Numărul expresiilor constante poate fi mai mic decit m în oricare din acoladele corespunzătoare celor n linii ale tabloului bidimensional.

În cazul tablourilor statice, declarația este precedată de cuvintul cheie *static*.

Într-o declarație sau definiție de tablou bidimensional se poate omite numai expresia constantă din prima paranteză pătrată, adică se poate omite numai n din construcția de mai sus.

În general, într-o declarație sau definiție de tablou multidimensional se poate omite numai expresia constantă din prima paranteză pătrată, adică limita superioară pentru primul indice. Ea poate fi omisă în una din cele trei cazuri care au fost amintite la tablourile unidimensionale:

- la declarații sau definiții care conțin initializari;
- la declarația de parametri;
- la declarații de *extern*.

Exemple:

```
1. int tab[3][4]={  
    {-1,0,1,-1},  
    {-1,0,1,2},  
    {10,20,30,40}  
};
```

Prin această construcție, elementele tabloului *tab* se initializează cu valorile:

```
tab[0][0]=-1, tab[0][1]=0, tab[0][2]=1, tab[0][3]=-1;  
tab[1][0]=-1, tab[1][1]=0, tab[1][2]=1, tab[1][3]=2;  
tab[2][0]=10, tab[2][1]=20, tab[2][2]=30, tab[2][3]=40.
```

```
2. int tab[] [4]={  
    {-1,0,1,-1},  
    {-1,0,1,2},  
    {10,20,30,40}  
};
```

Această construcție este identică, ca efect, cu cea din exemplul precedent.

```
3. double t[] [3]={  
    {0,1},  
    {-1},
```

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

t reprezintă o matrice de ordinul 3×3 ale cărei elemente se initializează astfel:
 $t[0][0]=0, t[0][1]=1, t[1][0]=-1, t[2][0]=1, t[2][1]=2, t[2][2]=3.$

Elementele $t[0][2], t[1][1]$ și $t[1][2]$ nu sunt initialize. Ele au valoarea inițială zero dacă tabloul este global sau valori inițiale nedeterminate dacă tabloul este automatic.

Construcțiile utilizate pentru inițializarea tablourilor bidimensionale se extind imediat pentru tablouri cu un număr mai mare de dimensiuni.

Exemplu:

```
int t3[] [3] [2]={  
    {{100,200}, {-1,1}, {0,1}},  
    {{123,456}, {789,10}, {11,12}}  
};
```

Exerciții:

- 6.2 Să se scrie un program care citește un întreg pozitiv ce reprezintă o sumă exprimată în lei. Se cere să se afișeze numărul minim de bancnote și monede de 10000 lei, 5000 lei, 1000 lei, 500 lei, 200 lei etc. necesare pentru a exprima suma respectivă.

Procesul de calcul a fost indicat în exercițiul 4.21. În cazul de față nu se mai folosesc instrucțiuni de atribuire pentru a da valori inițiale elementelor tabloului *mon* ci ele se initializează prin declarația tabloului *mon*.

PROGRAMUL BVI2

```
#include <stdio.h>  
#include <stdlib.h>  
  
main() /* exprimă o suma de lei în bancnote și monede de 10000 lei, 5000 lei, 1000 lei etc. */  
{  
    int mon[]={1,5,10,20,50,100,200,500,1000,5000,10000};  
    int i=sizeof mon/sizeof(int)-1;  
    long s,q;  
    char t[255];  
  
    for ( ; ; ) {  
        printf("suma= ");  
        if(gets(t) == NULL) {  
            printf("s-a tastat EOF\n");  
            exit(1);  
        }  
        if(sscanf(t,"%ld",&s) == 1 && s > 0) break;  
        printf("nu s-a tastat un întreg pozitiv\n");  
    } /* sfîrșit for */
```

```

do {
    q=s/mon[i];
    if(q)
        if(i<6) /* monede */
            if( q == 1 ) /* o moneda */
                if(i==0) /* o moneda de 1 leu */
                    printf("o moneda de 1 leu\n");
                else
                    printf("o moneda a %d lei\n",mon[i]);
            else /* mai multe monede */
                if(i==0) /* monede de 1 leu */
                    printf("%ld monede de 1 leu\n",q);
                else /* monede a mon[i] lei */
                    printf("%ld monede a %d lei\n",q,mon[i]);
        else /* bancnote */
            if(q==1) /* o bancnota */
                printf("o bancnota a %d lei\n",mon[i]);
            else /* mai multe bancnote */
                printf("%ld bancnote a %d lei\n",q,mon[i]);
    } while ( s != mon[i--]);
}

```

Observație:

La declararea lui *i* s-a făcut inițializare cu ajutorul expresiei:

```
sizeof mon / sizeof(int) - 1
```

Această expresie are ca valoare numărul elementelor tabloului *mon* micșorat cu 1, adică indicele ultimului element al tabloului. Într-adevăr, *sizeof mon* are ca valoare dimensiunea, în octeți, a zonei de memorie alocată tabloului *mon*, iar *sizeof(int)* are ca valoare numărul octețiilor necesari pentru a reprezenta o dată de tip *int*. Raportul lor ne dă chiar numărul elementelor tabloului *mon*.

6.3 Să se scrie o funcție care afișează elementele unui tablou de tip *char* și citește un intreg de tip *int*.

Funcția are parametrul *text* care este un tablou unidimensional de tip *char*. Ea afișează sirul de caractere păstrat în tabloul *text* înainte de a citi întregul. Întregul citit se atribuie variabilei globale *v_int*. Funcția returnează valoarea zero la întîlnirea sfîrșitului de fișier și 1 în caz contrar. La eroare, se reia citirea întregului după restarea acestuia de către operator.

FUNCȚIA BVI3

```

int cit_int(char text[])
/* - citește un intreg și-l atribuie variabilei globale v_int;
   - returneaza:
     0 - la întîlnirea sfîrșitului de fisier;
     1 - altfel.*/
{
    char t[255];

```

```

char texter[]="nu s-a tastat un intreg\n";
extern int v_int;

for ( ; ; ) {
    printf(text);
    if(gets(t) == NULL) return 0;
    if(sscanf(t,"%d",&v_int) == 1) return 1;
    printf(texter);
}
}

```

6.4 Să se scrie o funcție care citește un intreg de tip *int* care aparține unui interval dat.

Funcția are parametri:

- | | |
|-------------|---|
| <i>text</i> | - Tablou unidimensional de tip <i>char</i> care are aceeași semnificație ca în exercițiul precedent. |
| <i>inf</i> | - Întreg de tip <i>int</i> care reprezintă limita inferioară a intervalului căreia trebuie să-i aparțină numărul citit. |
| <i>sup</i> | - Întreg de tip <i>int</i> care reprezintă limita superioară a aceluiași interval. |

Funcția atrbuie întregul citit variabilei globale *v_int*. Ea returnează valoarea zero la întîlnirea sfîrșitului de fișier și 1 în caz contrar.

FUNCȚIA BVI4

```

int cit_int(char []);
int cit_int_lim(char text[], int inf,int sup)
/* - citește un intreg de tip int ce aparține intervalului [inf,sup] și-l atribuie variabilei v_int;
   - returneaza:
     0 - la întîlnirea sfîrșitului de fisier;
     1 - altfel.*/
{
    extern int v_int;

    for ( ; ; ) {
        if(cit_int(text) == 0) return 0; /* s-a întîlnit EOF */
        if(v_int >= inf && v_int <= sup) return 1;
        printf("intregul tastat nu aparține intervalului :");
        printf("%d,%d]\n", inf,sup);
        printf("se reia citirea\n");
    }
}

```

6.5 Să se scrie o funcție care validează o dată calendaristică.

Funcția are 3 parametri de tip *int*:

- | | |
|-------------|-----------------------------------|
| <i>zi</i> | - Numărul zilei din cadrul lunii. |
| <i>luna</i> | - Numărul lunii. |

an

- Anul calendaristic.

Funcția returnează:

- Dacă data calendaristică este validă.
- Altfel.

În exercițiile din aceasta carte se presupune că anul calendaristic aparține intervalului [1600,4900]. În acest caz, anul este bisect dacă expresia:

$an \% 4 == 0 \&\& an \% 100 != 0 \parallel an \% 400 == 0$

are valoarea adevarată (vezi exercițiul 3.12.).

Funcția utilizează un tablou global de tip *int* ale cărui elemente au ca valori numărul zilelor din lunile calendaristice. Numele acestui tablou este *nrzile*.

nrzile[i] are ca valoare numărul zilelor din luna *i*-a (*i*=1,2,...,12).

FUNCȚIA BVI5

```
int v_calend(int zi,int luna,int an)
/* validează data calendaristică;
   - returnează:
     1 - dacă este corectă;
     0 - altfel. */
{
    extern int nrzile[];

    if(an < 1600 || an > 4900) {
        printf("anul nu este în intervalul [1600,4900]!");
        return 0;
    }
    if(luna < 1 || luna > 12) {
        printf("luna=%d eronată\n", luna);
        return 0;
    }
    if(zi < 1 || zi > nrzile[luna] +
       (luna == 2 && (an%4 == 0 && an%100 || an%400 == 0)))
    {
        printf("ziua=%d eronată\n", zi);
        return 0;
    }
    return 1; /* data calendaristică validă */
}
```

Observație:

Elementul *nrzile[2]* are ca valoare numărul zilelor din luna februarie pentru un an nebisect, deci 28.

Dacă anul este bisect, atunci expresia:

(1) $an \% 4 == 0 \&\& an \% 100 \parallel an \% 400 == 0$

are valoarea 1. În acest caz numărul zilelor din luna februarie se mărește cu 1.

Accasta nu este valabil și pentru celelalte luni. Deci

nrzile[luna]

se mărește cu 1 cind expresia (1) are valoarea 1 și cind expresia

(2) $luna == 2$

este adevarată. Cu alte cuvinte, numărul zilelor din februarie se mărește cu 1, cind atât expresie (1), cit și expresie (2) sunt adevărate și numai atunci. Aceasta înseamnă că expresia:

(2) $\&\&$ (1)

are valoarea 1 atunci și numai atunci cind anul este bisect și luna este februarie.

6.6 Să se scrie o funcție care determină dintr-o dată calendaristică de forma zi, lună și an, numărul zilei din an pentru data respectivă.

Funcția returnează numărul determinat din parametri zi, luna și an.

Dacă, de exemplu, data calendaristică este 1 martie 1994, atunci ea reprezintă ziua 60 din anul respectiv. Într-adevăr, pînă la 1 martie 1994 au trecut lunile ianuarie și februarie, deci împreună cu 1 martie sunt:

31+28+1=60 zile

Într-un an bisect data de 1 martie este ziua 61 din anul respectiv.

Procesul de calcul constă din insumarea zilelor din lunile precedente lunii calendaristice definită de *luna*, la ziua curentă definită de *zi*.

Funcția de față presupune că data calendaristică este validă.

FUNCȚIA BVI6

```
int zi_din_an(int zi,int luna,int an)
/* determină ziua din an și returnează numărul astfel determinat */
{
    extern int nrzile[];
    int bisect=an%4 == 0 && an%100 || an%400 == 0;
    int i;

    for( i=1; i < luna; i++ )
        zi += nrzile[i] + (i == 2 && bisect);
    return zi;
}
```

6.7 Să se scrie o funcție care dintr-o dată calendaristică definită prin numărul zilei din an și anul respectiv, determină luna și ziua din luna respectivă.

Această funcție este inversa funcției *zi_din_an* definită în exercițiul precedent. Ea definește două valori:

numărul lunii

și

numărul zilei din luna respectivă.

Funcția poate fi realizată în una din variantele:

- Să existe un parametru formal care să fie un tablou de tip *int*. Fie acesta *tzzll* și care să corespundă unui parametru efectiv care să fie un tablou de 2 elemente de tip *int*. În acest caz se poate atribui, de exemplu, lui *tzzll[0]* numărul zilei din lună și lui *tzzll[1]* numărul lunii.
- Să existe două variabile globale la care să li se atribuie cele două valori: ziua din lună și respectiv luna.

Funcția realizată mai jos utilizează parametrul *tzzll*.

FUNCȚIA BVI7

```
void luna_si_ziua ( int zz, int an,int tzzll[])
/* - determină luna și ziua din luna;
   zz - ziua din an;
   an - anul calendaristic;
   - la revenire:
     tzzll[0] are ca valoare ziua din luna;
     tzzll[1] are ca valoare luna calendaristică */
{
    int bisect=an%4==0 && an%100 ==0 || an%400 ==0;
    int i;
    extern int nrzile[];
    for(i=1; zz > nrzile[i] + (i==2 && bisect);i++)
        zz -=(nrzile[i] + ( i==2 && bisect));
    tzzll[0] = zz;
    tzzll[1] = i;
}
```

6.8 Să se scrie o funcție care citește o dată calendaristică compusă din zi, luna și an.

Funcția validează data calendaristică respectivă. Ea returnează valoarea:

- 0* - La întîlnirea sfîrșitului de fișier.
1 - Altfel (data calendaristică corectă).

Are un parametru *tzzllaa*, care este un tablou de tip *int*. La revenire, elementele acestui tablou au valorile:

- | | |
|-------------------|----------------|
| <i>tzzllaa[0]</i> | - Ziua citită. |
| <i>tzzllaa[1]</i> | - Luna citită. |
| <i>tzzllaa[2]</i> | - Anul citit. |

FUNCȚIA BVI8

```
int cit_data_calend( int tzzllaa[])
/* - citește o dată calendaristică, o validează și o păstrează în tabloul tzzllaa:
   tzzllaa[0] - ziua;
   tzzllaa[1] - luna;
   tzzllaa[2] - anul;
   - funcția returnează:
     0 - la întîlnirea sfîrșitului de fisier;
     1 - altfel */
{
    extern int v_int;
    static char ziua[] = "ziua: ";
    static char luna[] = "luna: ";
    static char anul[] = "anul: ";
    static char eroare[] = "s-a tastat EOF";
    for( ; ; ) {
        /* se citește ziua */
        if(cit_int_lim(ziua,1,31) == 0) {
            printf("%s\n",eroare);
            return 0;
        }
        tzzllaa[0] = v_int;
        /* se citește luna */
        if(cit_int_lim(luna,1,12) == 0) {
            printf("%s\n",eroare);
            return 0;
        }
        tzzllaa[1] = v_int;
        /* se citește anul */
        if(cit_int_lim(anul,1600,4900) == 0) {
            printf("%s\n",eroare);
            return 0;
        }
        tzzllaa[2] = v_int;
        /* validare data calendaristică */
        if(v_calend(tzzllaa[0],tzzllaa[1],tzzllaa[2])) return 1;
        printf("data calendaristică este eronată\n");
        printf("se reia citirea datei calendaristice\n");
    }
}
```

6.9 Să se scrie un program care citește o dată calendaristică și afișează data calendaristică pentru ziua următoare.

Programul se realizează conform urmatorilor pași:

1. Se citește și se validează data calendaristică prin apelul funcției *v_calend*.

2. Dacă data citită este 31 decembrie dintr-un anumit an, atunci ziua următoare este 1 ianuarie din anul următor, apoi se trece la pasul 6.
Altfel se trece la pasul 3.
3. Se determină ziua din an (funcția *zi_din_an*).
4. Se incrementează ziua din an.
5. Se determină luna și ziua din lună (funcția *luna_si_ziua*).
6. Se afișează data calendaristică a zilei următoare.

PROGRAMUL BVI9

```
#include <stdio.h>
#include <stdlib.h>
#include "bvi3.cpp" /* functia cit_int */
#include "bvi4.cpp" /* functia cit_int_lim */
#include "bvi5.cpp" /* functia v_calend */
#include "bvi6.cpp" /* functia zi_din_an */
#include "bvi7.cpp" /* functia luna_si_ziua */
#include "bvi8.cpp" /* functia cit_data_calend */

/* variabile globale */
int v_int; /* se utilizeaza in functiile:
              cit_int; cit_int_lim; cit_data_calend. */

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

/* se utilizeaza in functiile:
   v_calend; zi_din_an; luna_si_ziua */

main() /* citește o data calendaristica, o validează și în caz că este
         corecta, afișează data calendaristica a zilei următoare */
{
    int data_calend[3];
    int z1[2];

    /* citește și validează data calendaristică */
    if(cit_data_calend(data_calend) == 0) exit(1);

    if(data_calend[0] == 31 && data_calend[1] == 12) {
        /* 31 decembrie; ziua următoare este 1 ianuarie din anul urmator */
        z1[0] = 1; /* 1 */
        z1[1] = 1; /* ianuarie */
        data_calend[2]++;
    }
    else /* se determină ziua următoare */
        luna_si_ziua(zi_din_an(data_calend[0],
                                data_calend[1], data_calend[2])+1,
                                data_calend[2], z1);

    /* afișează data calendaristica a zilei următoare */
    printf("ziua:%d\\tluna:%d\\tanul:%d\\n", z1[0], z1[1],
           data_calend[2]);
}
```

7. PROGRAMARE MODULARĂ

Un program poate fi format din mai multe *module*.

Numim *modul sursă*, o parte a textului sursă al programului care se compilează într-o singură compilare și separat de restul textului sursă al programului respectiv. Rezultatul compilării unui modul sursă este un *modul obiect*.

În cele ce urmează, prin *modul* vom înțelege un modul sursă. Modulele obiect le vom mai numi și module de *tip OBJ*.

De obicei, programele simple se compun dintr-un singur modul sau un număr relativ mic de module (4-5).

În componența unui modul intră proceduri "înrudite". Această noțiune nu este definită riguros. De obicei, spunem că mai multe proceduri sunt înrudite dacă utilizează în comun diferite date. De exemplu, funcțiile *zi_din_an* și *luna_si_ziua*, definite în capitolul precedent, se consideră că sunt înrudite. Ele utilizează în comun tabloul global *nrzile*.

Un modul se compune din unul sau mai multe fișiere. În cazul în care el se compune din mai multe fișiere, acestea se compilează împreună incluzindu-le unul în altul cu ajutorul construcției #include.

Modulele de tip OBJ, obținute prin compilările modulelor din compunerea unui program, pot fi reunită într-un fișier executabil (cu extensia .EXE) cu ajutorul editorului de legături.

Modulele unui program se definesc ca rezultat al procesului de *descompunere* a unei probleme în *subprobleme mai simple*. Acest proces de descompunere se poate continua, adică subproblemele obținute la o primă descompunere pot fi și ele la rindul lor descompuse în altele mai simple și aşa mai departe, pînă cînd se ajunge că toate subproblemele de la nivelurile inferioare să fie relativ simple. Diferite componente ale unei astfel de descompuneri se realizează prin module. De obicei, aceste module se pot realiza în paralel și relativ independent de către mai mulți programatori, ceea ce conduce la creșterea eficienței în programare.

Un avantaj mare pe care îl oferă modulele, este să numita posibilitate de "ascundere a datelor". Aceasta înseamnă că la o parte sau chiar la toate datele unui modul au acces direct numai procedurile din compunerea modulului respectiv. Adesea se spune că datele respective sunt *vizibile* numai la nivel de modul. Ele pot fi utilizate în alte module numai prin intermediul procedurilor modulului în care sunt vizibile. Mai mult decît atât, pot exista cazuri în care este avantajos ca și anumite proceduri dintr-un modul să fie *ascunse*. La acestea nu se pot face apeluri din alte module. Ele pot fi apelate numai de procedurile modulului în care sunt definite și ascunse.

Facilitatea de "ascundere" a datelor și procedurilor în module constituie o *protecție* împotriva deteriorării datelor prin accese neautorizate din alte module. Această protecție devine importantă mai ales în cazul problemelor complexe,

cind participă colective mari pentru programarea și implementarea lor.

Prin *programarea modulară* înțelegem stilul de programare care are la bază utilizarea de module în vederea "ascunderii" datelor și procedurilor pentru a realiza protecția datelor respective față de acese neautorizate.

Limbajul C a fost proiectat în ideea de a permite utilizarea programării modulare. În acest caz modulul conține funcții "înrudite". Datele statice, declarate în afara funcțiilor modulului, pot fi utilizate în comun de către aceste funcții. Ele sunt ascunse în modul, deoarece funcții din alte module nu pot face acces direct la ele.

De asemenea, se pot defini și funcții statice, ceea ce conduce la ascunderea lor în modulul în care au fost definite.

O funcție statică poate fi apelată numai de funcții definite în același modul cu ea.

Exerciții:

7.1 Să se realizeze un modul care implementează o stivă ale cărei elemente sunt de tip *int*.

Prin *stivă* înțelegem o mulțime ordonată de elemente la care se are acces în conformitate cu principiul *LIFO* (*Last In First Out*).

Cel mai simplu procedeu de implementare a unei stive este păstrarea elementelor ei într-un tablou unidimensional. Ulterior vom vedea și alte moduri de implementare a unei stive.

În zona de memorie alocată stivei se pot păstra elementele ei, unul după altul. De asemenea, ele se pot scoate din zona de memorie respectivă, unul cîte unul, în ordine inversă păstrării lor.

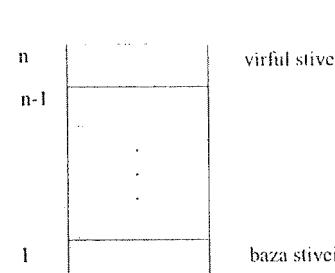
Ultimul element pus pe stivă se spune că este în *virful* stivei. Primul element pus pe stivă se află la *baza* stivei.

Virful stivei se modifică atunci cind se pun un element pe stivă sau cind se scoate de pe stivă. În primul caz *lungimea* stivei (numărul elementelor din stivă) crește, iar în cel de al doilea caz descrește.

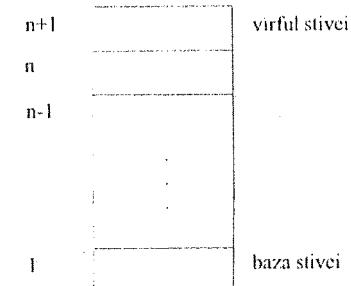
Într-adevăr, cind se pun un element pe stivă, atunci acesta se pună după cel aflat în *virful* stivei și prin aceasta *virful* stivei se modifică astfel încît elementul nou pus pe stivă să fie în *virful* ei. Cind se ia un element de pe stivă, atunci acesta este cel aflat în *virful* stivei și apoi *virful* stivei se modifică astfel încât, elementul care a fost pus pe stivă înaintea celui scos, să fie în *virful* ei.

Punerea unui element pe stivă

Înainte de a pune
un element pe stivă

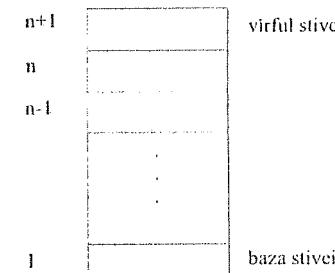


După ce s-a pus
un element pe stivă

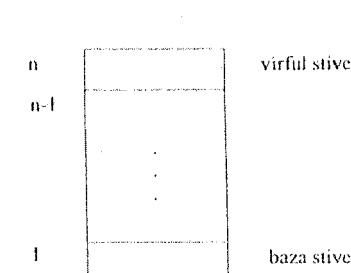


Scoaterea unui element de pe stivă

Înainte de a scoate
un element de pe stivă



După ce s-a scos
un element de pe stivă



Se observă că totdeauna, ultimul element pus pe stivă este primul care se scoate din stivă. De aici provine afirmația că o stivă se gestionează conform principiului *LIFO*.

Pentru a implementa o stivă vor fi necesare două funcții:

- una care să pună un element pe stivă
- și
- una care să scoată un element din stivă.

Aceste funcții au denumiri deja consacrate în literatura de specialitate și anume, funcția care pună un element pe stivă se numește *push*, iar cealaltă *pop*.

Vom păstra și noi aceste denumiri.

Am spus mai sus că vom implementa stiva folosind un tablou unidimensional. Acesta va fi în cazul de față de tip *int*. Numim *stack* acest tablou.

stack[0] este elementul de la baza stivei.

Functiile *push* și *pop* gestionează virful stivei și de aceea ele au nevoie de o variabilă care să definească indicele elementului din virful stivei. De obicei, se pastrează indicele *primului element liber*, adică indicele elementului tabloului *stack* care urmează imediat după elementul aflat în virful stivei (vezi [2]).

Numele *next* variabilei a cărei valoare curentă este primul element liber al tabloului *stack*.

Evident inițial *next=0*, deoarece la început nefiind nici un element pe stivă, *stack[0]* este primul element liber.

Utilizatorul stivei nu trebuie să aibă acces direct la elementele stivei și de aceea atât tabloul *stack*, cit și variabila *next* trebuie "ascunse" în modul. În felul acesta, utilizatorul are posibilitatea să pună un element pe stivă în virful ei și apoi să scoată elementul din virful stivei apelând funcția *pop*. Rezultă că aceste două funcții nu se "ascund" în modul.

De obicei, se definesc și alte funcții (vezi [5]) cum ar fi:

- | | |
|--------------|--|
| <i>clear</i> | - Pentru a vîda stivă. |
| <i>empty</i> | - Stabilește dacă stiva este vîda sau nu. |
| <i>full</i> | - Stabilește dacă stiva este plină sau nu. |
| <i>top</i> | - Permite acces la elementul din virful stivei, fără a-l scoate din stivă. |

Toate aceste funcții trebuie să aibă acces atât la *stack*, cit și la variabila *next*.

De aceea, *stack* și *next* se vor declara statice în afara corporilor lor.

Cele 6 funcții amintite mai sus vor avea un caracter global, ele putind fi apelate din orice modul al programului.

Funcția *push* are prototipul:

```
void push(int x);
```

În principiu, ea pune pe stivă valoarea lui *x*, deci trebuie să facă atribuirea:

```
stack[next] = x;
```

și apoi să incrementeze pe *next* pentru ca aceasta să definească locul liber curent. De aceea, atribuirea de mai sus se poate realiza împreună cu incrementarea lui *next*:

```
stack[next++] = x.
```

Funcția trebuie să testeze, în prealabil, dacă există loc liber pentru a păstra pe *x* și numai în acest caz se va executa atribuirea de mai sus. În caz că stiva este plină (depășită), se dă un mesaj de eroare.

Funcția *pop* are prototipul:

```
int pop(void);
```

Ea returnează valoarea din virful stivei. În acest scop se va decrementa *next* și apoi se va returna valoarea elementului

```
stack[next].
```

Aceste două acțiuni se pot realiza cu ajutorul instrucțiunii:

```
return stack[--next];
```

Se observă că prin aceasta virful stivei coboară cu un element, deoarece *next*, care exprimă primul element liber, este chiar indicele elementului eliminat de pe stivă.

Înainte de a executa instrucțiunea de mai sus, funcția *pop* va testa dacă stiva nu cumva este vidă. În cazul în care stiva este vidă, funcția *pop* va afișa un mesaj de eroare și va returna valoarea zero.

Funcția *top* este ca și *pop*, cu deosebirea că nu se elimină elementul din virful stivei, ci numai se returnează valoarea lui.

Funcția *clear* are prototipul:

```
void clear(void);
```

Ea videază stiva, ceea ce se realizează simplu prin atribuirea valorii zero variabilei *next*. În felul acesta, tot tabloul devine liber.

Funcția *empty* are prototipul:

```
int empty(void);
```

Ea returnează valoarea 1 dacă stiva este vidă și zero în caz contrar.

Funcția *full* are prototipul:

```
int full(void);
```

Ea returnează valoarea 1 dacă stiva este plină și zero în caz contrar.

ACESTE FUNCȚII, îMPREUNĂ CU DECLARAȚIILE LUI *stack* și *next* SE PĂSTREAZĂ ÎNTR-UN FIȘIER CARE FORMEAZĂ MODULUL CE IMPLEMENTEAZĂ STIVA *stack* PRIN INTERMEDIUL UNUI TABLOU DE TIP *int*.

MODULUL BVII

```
#define MAX 1000  
  
static int stack[MAX];  
static int next = 0;  
  
void push( int x ) /* pun x pe stiva */  
{  
    if(next < MAX ) stack[next++] = x;  
    else printf("stiva este plina\n");  
}  
  
int pop() /* scoate din stivă elementul din virful ei */  
{
```

```

if( next > 0 ) return stack[--next];
else printf("stiva vida\n");
}

int top() /* returnaza elementul din virful stivui */
{
    if(next > 0 ) return stack[next-1];
    else printf("stiva vida\n");
}

void clear() /* vidcaza stiva */
{
    next = 0;
}

int empty() /* returnaza 1 daca stiva estevida si 0 altfel */
{
    return !next;
}

int full() /* returnaza 1 daca stiva esteplina si 0 altfel */
{
    return next == MAX;
}

```

7.2 Să se scrie un program care transcrie o expresie cu operanzi numere naturale, în formă poloneză postfixată.

Prin expresie aritmetică înțelegem o expresie în care pot fi utilizate numai cele 4 operații binare: adunare (+), scădere (-), înmulțire (*) și împărțire (/).

Se pot utiliza parantezele rotunde pentru a schimba prioritatea operatorilor.

Forma poloneză postfixată a unei expresii aritmetice se poate defini ca mai jos:

- forma poloneză postfixată a unui operand coincide cu el însuși dacă nu este inclus în paranteze rotunde;
- forma poloneză postfixată a unui operand de forma
 (exp)
coincide cu forma poloneză postfixată a lui exp ;
- dacă operatorul binar op leagă expresiile $fexp_1$ și $fexp_2$ care sunt în formă poloneză postfixată:

$fexp_1 \ op \ fexp_2$

Atunci forma poloneză postfixată a acestei expresii este

$fexp_1 \ fexp_2 \ op$

Exemple:

Expresie aritmetică	Echivalentul ei în formă poloneză postfixată
$1+2$	$1\ 2\ +$
$1+2*3$	$1\ 2\ 3\ *\ +$
$(1+2)*3$	$1\ 2\ +\ 3\ *$

Un algoritm simplu pentru a transforma o expresie aritmetică în formă poloneză postfixată se bazează pe utilizarea unei stive în care se păstrează temporar operatorii expresiei. În acest scop se va folosi stiva implementată prin modulul din exemplul precedent.

Expresia aritmetică în formă poloneză postfixată, pe măsură ce se construiește, se păstrează într-un tablou $tespp$ de tip caracter.

Algoritmul de traducere are următorii pași:

- Un operand se păstrează automat în tabloul $tespp$. Acest lucru rezultă din faptul că prin această transformare operanții își păstrează ordinea din expresia inițială. Apoi se continuă cu următorul element al expresiei.
- Paranteza deschisă se introduce automat în stivă.
- Un operator se introduce în stivă dacă în virful stivei se află:
 - paranteza deschisă;
 - un operator de prioritate mai mică;
 - sau
 - stiva este vidă.
 Dacă în virful stivei se află un operator de prioritate mai mare sau egală cu celui curent, atunci operatorul din virful stivei se scoate din stivă și se păstrează în tabloul $tespp$. Apoi se compară din nou operatorul din virful stivei cu cel curent și se reia pasul c.
- La închiderea unei paranteze inchise, se scoad pe rind operatorii din stivă și se introduc în tabloul $tespp$, pînă la închiderea parantezei deschise din stivă. Aceasta pur și simplu se elimină din stivă. Apoi se continuă cu următorul element al expresiei.
- La terminarea parcurgerii expresiei, se scoad pe rînd toți operatorii care se mai află în stivă și se trăie în $tespp$.

Se presupune că expresia se termină cu punct și virgulă.

La baza stivei se va afla caracterul NUL ('\0').

Programul de față poate transforma mai multe expresii, pînă la tastarea sfîrșitului de fișier.

Mentionăm că metoda de transcriere a expresiilor descrisă mai sus, funcționează exact numai dacă expresia respectivă este corectă din punct de vedere sintactic.

În mod normal acest program se compune din două module, unul care a fost definit în exercițiul precedent (fișierul sursă BVII1.CPP) și prin care se implementează o stivă pe care se pun elementele de tip *int*, iar cel de al doilea

modul definește funcția principală.

Cele două module se pot compila distinct obținându-se două module de tip OBJ care urmează apoi a fi link-editate împreună pentru a obține imaginea executabilă a programului.

Cele două fișiere de tip sursă pot fi compilate și împreună, procedind ca pînă acum, adică incluzind fișierul BVII1.CPP în fișierul BVII2.CPP.

De asemenea, fișierele respective pot fi compilate și link-editate folosind un fișier de tip *Project*. Acesta se editează utilizând meniul *Project* al mediului integrat de dezvoltare Turbo C++.

Mai jos s-a adoptat varianta cu includerea fișierului BVII1.CPP.

Fișierul BVII2.CPP, alături de funcția principală mai conține o funcție denumită *sca*. Aceasta are prototipul:

```
int sca(int c);
```

Ea realizează saltul peste caracterele albe.

Dacă *c* are ca valoare codul ASCII al unui caracter alb, atunci se citesc caracterele care urmăză, pînă la întîlnirea primului caracter care nu este alb. La revenire, se returnează codul ultimului caracter citit. Dacă *c* are ca valoare codul unui caracter ASCII care nu este alb, atunci se revine din funcție cu codul caracterului respectiv.

PROGRAMUL BVII2

```
#include <stdio.h>
#include "bvi11.cpp"

#define MAXTEFPP 1000

int sca(int); /* prototipul functiei sca */

/* prototipul functiilor push, pop si clear sunt necesare cînd cele două
   fișiere se compilează separat; în cazul de fata sunt în plus */
void push(int);
int pop(void);
void clear(void);

main() /* transformă expresii aritmétice în forma poloneza postfixată */
{
    int c,i,j;
    char tefpp[MAXTEFPP];

    for ( c=getchar(); c != EOF; ) {
        i=0; /* indice pentru tefpp */
        clear(); /* vidică stiva */
        push('0'); /* zero la baza stivei */
        while((c=sca(c)) != ';' && c != EOF) {
            /* 1 */
            /* c nu este alb, punct și virgula sau sfîrșitul de fișier */
            while(c>='0' && c<='9') { /* 2 */
                /* 3 */
                /* 4 */
                /* 5 */
                /* 6 */
                /* 7 */
                /* 8 */
                /* 9 */
                /* 10 */
                /* 11 */
                /* 12 */
                /* 13 */
                /* 14 */
                /* 15 */
                /* 16 */
                /* 17 */
                /* 18 */
                /* 19 */
                /* 20 */
                /* 21 */
                /* 22 */
                /* 23 */
                /* 24 */
                /* 25 */
                /* 26 */
                /* 27 */
                /* 28 */
                /* 29 */
                /* 30 */
                /* 31 */
                /* 32 */
                /* 33 */
                /* 34 */
                /* 35 */
                /* 36 */
                /* 37 */
                /* 38 */
                /* 39 */
                /* 40 */
                /* 41 */
                /* 42 */
                /* 43 */
                /* 44 */
                /* 45 */
                /* 46 */
                /* 47 */
                /* 48 */
                /* 49 */
                /* 50 */
                /* 51 */
                /* 52 */
                /* 53 */
                /* 54 */
                /* 55 */
                /* 56 */
                /* 57 */
                /* 58 */
                /* 59 */
                /* 60 */
                /* 61 */
                /* 62 */
                /* 63 */
                /* 64 */
                /* 65 */
                /* 66 */
                /* 67 */
                /* 68 */
                /* 69 */
                /* 70 */
                /* 71 */
                /* 72 */
                /* 73 */
                /* 74 */
                /* 75 */
                /* 76 */
                /* 77 */
                /* 78 */
                /* 79 */
                /* 80 */
                /* 81 */
                /* 82 */
                /* 83 */
                /* 84 */
                /* 85 */
                /* 86 */
                /* 87 */
                /* 88 */
                /* 89 */
                /* 90 */
                /* 91 */
                /* 92 */
                /* 93 */
                /* 94 */
                /* 95 */
                /* 96 */
                /* 97 */
                /* 98 */
                /* 99 */
                /* 100 */
                /* 101 */
                /* 102 */
                /* 103 */
                /* 104 */
                /* 105 */
                /* 106 */
                /* 107 */
                /* 108 */
                /* 109 */
                /* 110 */
                /* 111 */
                /* 112 */
                /* 113 */
                /* 114 */
                /* 115 */
                /* 116 */
                /* 117 */
                /* 118 */
                /* 119 */
                /* 120 */
                /* 121 */
                /* 122 */
                /* 123 */
                /* 124 */
                /* 125 */
                /* 126 */
                /* 127 */
                /* 128 */
                /* 129 */
                /* 130 */
                /* 131 */
                /* 132 */
                /* 133 */
                /* 134 */
                /* 135 */
                /* 136 */
                /* 137 */
                /* 138 */
                /* 139 */
                /* 140 */
                /* 141 */
                /* 142 */
                /* 143 */
                /* 144 */
                /* 145 */
                /* 146 */
                /* 147 */
                /* 148 */
                /* 149 */
                /* 150 */
                /* 151 */
                /* 152 */
                /* 153 */
                /* 154 */
                /* 155 */
                /* 156 */
                /* 157 */
                /* 158 */
                /* 159 */
                /* 160 */
                /* 161 */
                /* 162 */
                /* 163 */
                /* 164 */
                /* 165 */
                /* 166 */
                /* 167 */
                /* 168 */
                /* 169 */
                /* 170 */
                /* 171 */
                /* 172 */
                /* 173 */
                /* 174 */
                /* 175 */
                /* 176 */
                /* 177 */
                /* 178 */
                /* 179 */
                /* 180 */
                /* 181 */
                /* 182 */
                /* 183 */
                /* 184 */
                /* 185 */
                /* 186 */
                /* 187 */
                /* 188 */
                /* 189 */
                /* 190 */
                /* 191 */
                /* 192 */
                /* 193 */
                /* 194 */
                /* 195 */
                /* 196 */
                /* 197 */
                /* 198 */
                /* 199 */
                /* 200 */
                /* 201 */
                /* 202 */
                /* 203 */
                /* 204 */
                /* 205 */
                /* 206 */
                /* 207 */
                /* 208 */
                /* 209 */
                /* 210 */
                /* 211 */
                /* 212 */
                /* 213 */
                /* 214 */
                /* 215 */
                /* 216 */
                /* 217 */
                /* 218 */
                /* 219 */
                /* 220 */
                /* 221 */
                /* 222 */
                /* 223 */
                /* 224 */
                /* 225 */
                /* 226 */
                /* 227 */
                /* 228 */
                /* 229 */
                /* 230 */
                /* 231 */
                /* 232 */
                /* 233 */
                /* 234 */
                /* 235 */
                /* 236 */
                /* 237 */
                /* 238 */
                /* 239 */
                /* 240 */
                /* 241 */
                /* 242 */
                /* 243 */
                /* 244 */
                /* 245 */
                /* 246 */
                /* 247 */
                /* 248 */
                /* 249 */
                /* 250 */
                /* 251 */
                /* 252 */
                /* 253 */
                /* 254 */
                /* 255 */
                /* 256 */
                /* 257 */
                /* 258 */
                /* 259 */
                /* 260 */
                /* 261 */
                /* 262 */
                /* 263 */
                /* 264 */
                /* 265 */
                /* 266 */
                /* 267 */
                /* 268 */
                /* 269 */
                /* 270 */
                /* 271 */
                /* 272 */
                /* 273 */
                /* 274 */
                /* 275 */
                /* 276 */
                /* 277 */
                /* 278 */
                /* 279 */
                /* 280 */
                /* 281 */
                /* 282 */
                /* 283 */
                /* 284 */
                /* 285 */
                /* 286 */
                /* 287 */
                /* 288 */
                /* 289 */
                /* 290 */
                /* 291 */
                /* 292 */
                /* 293 */
                /* 294 */
                /* 295 */
                /* 296 */
                /* 297 */
                /* 298 */
                /* 299 */
                /* 300 */
                /* 301 */
                /* 302 */
                /* 303 */
                /* 304 */
                /* 305 */
                /* 306 */
                /* 307 */
                /* 308 */
                /* 309 */
                /* 310 */
                /* 311 */
                /* 312 */
                /* 313 */
                /* 314 */
                /* 315 */
                /* 316 */
                /* 317 */
                /* 318 */
                /* 319 */
                /* 320 */
                /* 321 */
                /* 322 */
                /* 323 */
                /* 324 */
                /* 325 */
                /* 326 */
                /* 327 */
                /* 328 */
                /* 329 */
                /* 330 */
                /* 331 */
                /* 332 */
                /* 333 */
                /* 334 */
                /* 335 */
                /* 336 */
                /* 337 */
                /* 338 */
                /* 339 */
                /* 340 */
                /* 341 */
                /* 342 */
                /* 343 */
                /* 344 */
                /* 345 */
                /* 346 */
                /* 347 */
                /* 348 */
                /* 349 */
                /* 350 */
                /* 351 */
                /* 352 */
                /* 353 */
                /* 354 */
                /* 355 */
                /* 356 */
                /* 357 */
                /* 358 */
                /* 359 */
                /* 360 */
                /* 361 */
                /* 362 */
                /* 363 */
                /* 364 */
                /* 365 */
                /* 366 */
                /* 367 */
                /* 368 */
                /* 369 */
                /* 370 */
                /* 371 */
                /* 372 */
                /* 373 */
                /* 374 */
                /* 375 */
                /* 376 */
                /* 377 */
                /* 378 */
                /* 379 */
                /* 380 */
                /* 381 */
                /* 382 */
                /* 383 */
                /* 384 */
                /* 385 */
                /* 386 */
                /* 387 */
                /* 388 */
                /* 389 */
                /* 390 */
                /* 391 */
                /* 392 */
                /* 393 */
                /* 394 */
                /* 395 */
                /* 396 */
                /* 397 */
                /* 398 */
                /* 399 */
                /* 400 */
                /* 401 */
                /* 402 */
                /* 403 */
                /* 404 */
                /* 405 */
                /* 406 */
                /* 407 */
                /* 408 */
                /* 409 */
                /* 410 */
                /* 411 */
                /* 412 */
                /* 413 */
                /* 414 */
                /* 415 */
                /* 416 */
                /* 417 */
                /* 418 */
                /* 419 */
                /* 420 */
                /* 421 */
                /* 422 */
                /* 423 */
                /* 424 */
                /* 425 */
                /* 426 */
                /* 427 */
                /* 428 */
                /* 429 */
                /* 430 */
                /* 431 */
                /* 432 */
                /* 433 */
                /* 434 */
                /* 435 */
                /* 436 */
                /* 437 */
                /* 438 */
                /* 439 */
                /* 440 */
                /* 441 */
                /* 442 */
                /* 443 */
                /* 444 */
                /* 445 */
                /* 446 */
                /* 447 */
                /* 448 */
                /* 449 */
                /* 450 */
                /* 451 */
                /* 452 */
                /* 453 */
                /* 454 */
                /* 455 */
                /* 456 */
                /* 457 */
                /* 458 */
                /* 459 */
                /* 460 */
                /* 461 */
                /* 462 */
                /* 463 */
                /* 464 */
                /* 465 */
                /* 466 */
                /* 467 */
                /* 468 */
                /* 469 */
                /* 470 */
                /* 471 */
                /* 472 */
                /* 473 */
                /* 474 */
                /* 475 */
                /* 476 */
                /* 477 */
                /* 478 */
                /* 479 */
                /* 480 */
                /* 481 */
                /* 482 */
                /* 483 */
                /* 484 */
                /* 485 */
                /* 486 */
                /* 487 */
                /* 488 */
                /* 489 */
                /* 490 */
                /* 491 */
                /* 492 */
                /* 493 */
                /* 494 */
                /* 495 */
                /* 496 */
                /* 497 */
                /* 498 */
                /* 499 */
                /* 500 */
                /* 501 */
                /* 502 */
                /* 503 */
                /* 504 */
                /* 505 */
                /* 506 */
                /* 507 */
                /* 508 */
                /* 509 */
                /* 510 */
                /* 511 */
                /* 512 */
                /* 513 */
                /* 514 */
                /* 515 */
                /* 516 */
                /* 517 */
                /* 518 */
                /* 519 */
                /* 520 */
                /* 521 */
                /* 522 */
                /* 523 */
                /* 524 */
                /* 525 */
                /* 526 */
                /* 527 */
                /* 528 */
                /* 529 */
                /* 530 */
                /* 531 */
                /* 532 */
                /* 533 */
                /* 534 */
                /* 535 */
                /* 536 */
                /* 537 */
                /* 538 */
                /* 539 */
                /* 540 */
                /* 541 */
                /* 542 */
                /* 543 */
                /* 544 */
                /* 545 */
                /* 546 */
                /* 547 */
                /* 548 */
                /* 549 */
                /* 550 */
                /* 551 */
                /* 552 */
                /* 553 */
                /* 554 */
                /* 555 */
                /* 556 */
                /* 557 */
                /* 558 */
                /* 559 */
                /* 560 */
                /* 561 */
                /* 562 */
                /* 563 */
                /* 564 */
                /* 565 */
                /* 566 */
                /* 567 */
                /* 568 */
                /* 569 */
                /* 570 */
                /* 571 */
                /* 572 */
                /* 573 */
                /* 574 */
                /* 575 */
                /* 576 */
                /* 577 */
                /* 578 */
                /* 579 */
                /* 580 */
                /* 581 */
                /* 582 */
                /* 583 */
                /* 584 */
                /* 585 */
                /* 586 */
                /* 587 */
                /* 588 */
                /* 589 */
                /* 590 */
                /* 591 */
                /* 592 */
                /* 593 */
                /* 594 */
                /* 595 */
                /* 596 */
                /* 597 */
                /* 598 */
                /* 599 */
                /* 600 */
                /* 601 */
                /* 602 */
                /* 603 */
                /* 604 */
                /* 605 */
                /* 606 */
                /* 607 */
                /* 608 */
                /* 609 */
                /* 610 */
                /* 611 */
                /* 612 */
                /* 613 */
                /* 614 */
                /* 615 */
                /* 616 */
                /* 617 */
                /* 618 */
                /* 619 */
                /* 620 */
                /* 621 */
                /* 622 */
                /* 623 */
                /* 624 */
                /* 625 */
                /* 626 */
                /* 627 */
                /* 628 */
                /* 629 */
                /* 630 */
                /* 631 */
                /* 632 */
                /* 633 */
                /* 634 */
                /* 635 */
                /* 636 */
                /* 637 */
                /* 638 */
                /* 639 */
                /* 640 */
                /* 641 */
                /* 642 */
                /* 643 */
                /* 644 */
                /* 645 */
                /* 646 */
                /* 647 */
                /* 648 */
                /* 649 */
                /* 650 */
                /* 651 */
                /* 652 */
                /* 653 */
                /* 654 */
                /* 655 */
                /* 656 */
                /* 657 */
                /* 658 */
                /* 659 */
                /* 660 */
                /* 661 */
                /* 662 */
                /* 663 */
                /* 664 */
                /* 665 */
                /* 666 */
                /* 667 */
                /* 668 */
                /* 669 */
                /* 670 */
                /* 671 */
                /* 672 */
                /* 673 */
                /* 674 */
                /* 675 */
                /* 676 */
                /* 677 */
                /* 678 */
                /* 679 */
                /* 680 */
                /* 681 */
                /* 682 */
                /* 683 */
                /* 684 */
                /* 685 */
                /* 686 */
                /* 687 */
                /* 688 */
                /* 689 */
                /* 690 */
                /* 691 */
                /* 692 */
                /* 693 */
                /* 694 */
                /* 695 */
                /* 696 */
                /* 697 */
                /* 698 */
                /* 699 */
                /* 700 */
                /* 701 */
                /* 702 */
                /* 703 */
                /* 704 */
                /* 705 */
                /* 706 */
                /* 707 */
                /* 708 */
                /* 709 */
                /* 710 */
                /* 711 */
                /* 712 */
                /* 713 */
                /* 714 */
                /* 715 */
                /* 716 */
                /* 717 */
                /* 718 */
                /* 719 */
                /* 720 */
                /* 721 */
                /* 722 */
                /* 723 */
                /* 724 */
                /* 725 */
                /* 726 */
                /* 727 */
                /* 728 */
                /* 729 */
                /* 730 */
                /* 731 */
                /* 732 */
                /* 733 */
                /* 734 */
                /* 735 */
                /* 736 */
                /* 737 */
                /* 738 */
                /* 739 */
                /* 740 */
                /* 741 */
                /* 742 */
                /* 743 */
                /* 744 */
                /* 745 */
                /* 746 */
                /* 747 */
                /* 748 */
                /* 749 */
                /* 750 */
                /* 751 */
                /* 752 */
                /* 753 */
                /* 754 */
                /* 755 */
                /* 756 */
                /* 757 */
                /* 758 */
                /* 759 */
                /* 760 */
                /* 761 */
                /* 762 */
                /* 763 */
                /* 764 */
                /* 765 */
                /* 766 */
                /* 767 */
                /* 768 */
                /* 769 */
                /* 770 */
                /* 771 */
                /* 772 */
                /* 773 */
                /* 774 */
                /* 775 */
                /* 776 */
                /* 777 */
                /* 778 */
                /* 779 */
                /* 780 */
                /* 781 */
                /* 782 */
                /* 783 */
                /* 784 */
                /* 785 */
                /* 786 */
                /* 787 */
                /* 788 */
                /* 789 */
                /* 790 */
                /* 791 */
                /* 792 */
                /* 793 */
                /* 794 */
                /* 795 */
                /* 796 */
                /* 797 */
                /* 798 */
                /* 799 */
                /* 800 */
                /* 801 */
                /* 802 */
                /* 803 */
                /* 804 */
                /* 805 */
                /* 806 */
                /* 807 */
                /* 808 */
                /* 809 */
                /* 810 */
                /* 811 */
                /* 812 */
                /* 813 */
                /* 814 */
                /* 815 */
                /* 816 */
                /* 817 */
                /* 818 */
                /* 819 */
                /* 820 */
                /* 821 */
                /* 822 */
                /* 823 */
                /* 824 */
                /* 825 */
                /* 826 */
                /* 827 */
                /* 828 */
                /* 829 */
                /* 830 */
                /* 831 */
                /* 832 */
                /* 833 */
                /* 834 */
                /* 835 */
                /* 836 */
                /* 837 */
                /* 838 */
                /* 839 */
                /* 840 */
                /* 841 */
                /* 842 */
                /* 843 */
                /* 844 */
                /* 845 */
                /* 846 */
                /* 847 */
                /* 848 */
                /* 849 */
                /* 850 */
                /* 851 */
                /* 852 */
                /* 853 */
                /* 854 */
                /* 855 */
                /* 856 */
                /* 857 */
                /* 858 */
                /* 859 */
                /* 860 */
                /* 861 */
                /* 862 */
                /* 863 */
                /* 864 */
                /* 865 */
                /* 866 */
                /* 867 */
                /* 868 */
                /* 869 */
                /* 870 */
                /* 871 */
                /* 872 */
                /* 873 */
                /* 874 */
                /* 875 */
                /* 876 */
                /* 877 */
                /* 878 */
                /* 879 */
                /* 880 */
                /* 881 */
                /* 882 */
                /* 883 */
                /* 884 */
                /* 885 */
                /* 886 */
                /* 887 */
                /* 888 */
                /* 889 */
                /* 890 */
                /* 891 */
                /* 892 */
                /* 893 */
                /* 894 */
                /* 895 */
                /* 896 */
                /* 897 */
                /* 898 */
                /* 899 */
                /* 900 */
                /* 901 */
                /* 902 */
                /* 903 */
                /* 904 */
                /* 905 */
                /* 906 */
                /* 907 */
                /* 908 */
                /* 909 */
                /* 910 */
                /* 911 */
                /* 912 */
                /* 913 */
                /* 914 */
                /* 915 */
                /* 916 */
                /* 917 */
                /* 918 */
                /* 919 */
                /* 920 */
                /* 921 */
                /* 922 */
                /* 923 */
                /* 924 */
                /* 925 */
                /* 926 */
                /* 927 */
                /* 928 */
                /* 929 */
                /* 930 */
                /* 931 */
                /* 932 */
                /* 933 */
                /* 934 */
                /* 935 */
                /* 936 */
                /* 937 */
                /* 938 */
                /* 939 */
                /* 940 */
                /* 941 */
                /* 942 */
                /* 943 */
                /* 944 */
                /* 945 */
                /* 946 */
                /* 947 */
                /* 948 */
                /* 949 */
                /* 950 */
                /* 951 */
                /* 952 */
                /* 953 */
                /* 954 */
                /* 955 */
                /* 956 */
                /* 957 */
                /* 958 */
                /* 959 */
                /* 960 */
                /* 961 */
                /* 962 */
                /* 963 */
                /* 964 */
                /* 965 */
                /* 966 */
                /* 967 */
                /* 968 */
                /* 969 */
                /* 970 */
                /* 971 */
                /* 972 */
                /* 973 */
                /* 974 */
                /* 975 */
                /* 976 */
                /* 977 */
                /* 978 */
                /* 979 */
                /* 980 */
                /* 981 */
                /* 982 */
                /* 983 */
                /* 984 */
                /* 985 */
                /* 986 */
                /* 987 */
                /* 988 */
                /* 989 */
                /* 990 */
                /* 991 */
                /* 992 */
                /* 993 */
                /* 994 */
                /* 995 */
                /* 996 */
                /* 997 */
                /* 998 */
                /* 999 */
                /* 1000 */
                /* 1001 */
                /* 1002 */
                /* 1003 */
                /* 1004 */
                /* 1005 */
                /* 1006 */
                /* 1007 */
                /* 1008 */
                /* 1009 */
                /* 10010 */
                /* 10011 */
                /* 10012 */
                /* 10013 */
                /* 10014 */
                /* 10015 */
                /* 10016 */
                /* 10017 */
                /* 10018 */
                /* 10019 */
                /* 10020 */
                /* 10021 */
                /* 10022 */
                /* 10023 */
                /* 10024 */
                /* 10025 */
                /* 10026 */
                /* 10027 */
                /* 10028 */
                /* 10029 */
                /* 10030 */
                /* 10031 */
                /* 10032 */
                /* 10033 */
                /* 10034 */
                /* 10035 */
                /* 10036 */
                /* 10037 */
                /* 10038 */
                /* 10039 */
                /* 10040 */
                /* 10041 */
                /* 10042 */
                /* 10043 */
                /* 10044 */
                /* 10045 */
                /* 10046 */
                /* 10047 */
                /* 10048 */
                /* 10049 */
                /* 10050 */
                /* 10051 */
                /* 10052 */
                /* 10053 */
                /* 10054 */
                /* 10055 */
                /* 10056 */
                /* 10057 */
                /* 10058 */
                /* 10059 */
                /* 10060 */
                /* 10061 */
                /* 10062 */
                /* 10063 */
                /* 10064 */
                /* 10065 */
                /* 10066 */
                /* 10067 */
                /* 10068 */
                /* 10069 */
                /* 10070 */
                /* 10071 */
                /* 10072 */
                /* 10073 */
                /* 10074 */
                /* 10075 */
                /* 10076 */
                /* 10077 */
                /* 10078 */
                /* 10079 */
                /* 10080 */
                /* 10081 */
                /* 10082 */
                /* 10083 */
                /* 10084 */
                /* 10085 */
                /* 10086 */
                /* 10087 */
                /* 10088 */
                /* 10089 */
                /* 10090 */
                /* 10091 */
                /* 10092 */
                /* 10093 */
                /* 10094 */
                /* 10095 */
                /* 10096 */
                /* 10097 */
                /* 10098 */
                /* 10099 */
                /* 100100 */
                /* 100101 */
                /* 100102 */
                /* 100103 */
                /* 100104 */
                /* 100105 */
                /* 100106 */
                /* 100107 */
                /* 100108 */
                /* 100109 */
                /* 100110 */
                /* 100111 */
                /* 100112 */
                /* 100113 */
                /* 100114 */
                /* 100115 */
                /* 100116 */
                /* 100117 */
                /* 100118 */
                /* 100119 */
                /* 100120 */
                /* 100121 */
                /* 100122 */
                /* 100123 */
                /* 100124 */
                /* 100125 */
                /* 100126 */
                /* 100127 */
                /* 100128 */
                /* 100129 */
                /* 100130 */
                /* 100131 */
                /* 100132 */
                /* 100133 */
                /* 100134 */
                /* 100135 */
                /* 100136 */
                /* 100137 */
                /* 100138 */
                /* 100139 */
                /* 100140 */
                /* 100141 */
                /* 100142 */
                /* 100143 */
                /* 100144 */
                /* 100145 */
                /* 100146 */
                /* 100147 */
                /* 100148 */
                /* 100149 */
                /* 100150 */
                /* 100151 */
                /* 100152 */
                /* 100153 */
                /* 100154 */
                /* 100155 */
                /* 100156 */
                /* 100157 */
                /* 100158 */
                /* 100159 */
                /* 100160 */
                /* 100161 */
                /* 100162 */
                /* 100163 */
                /* 100164 */
                /* 100165 */
                /* 100166 */
                /* 100167 */
                /* 100168 */
                /* 100169 */
                /* 100170 */
                /* 100171 */
                /* 100172 */
                /* 100173 */
                /* 100174 */
                /* 100175 */
                /* 100176 */
                /* 100177 */
                /* 100178 */
                /* 100179 */
                /* 100180 */
                /* 100181 */
                /* 100182 */
                /* 100183 */
                /* 100184 */
                /* 100185 */
                /* 100186 */
                /* 100187 */
                /* 100188 */
                /* 100189 */
                /* 100190 */
                /* 100191 */
                /* 100192 */
                /* 100193 */
                /* 100194 */
                /* 100195 */
                /* 100196 */
                /* 100197 */
                /* 100198 */
                /* 100199 */
                /* 100200 */
                /* 100201 */
                /* 100202 */
                /* 100203 */
                /* 100204 */
                /* 100205 */
                /* 100206 */
                /* 100207 */
                /* 100208 */
                /* 100209 */
                /* 100210 */
                /* 100211 */
                /* 100212 */
                /* 100213 */
                /* 100214 */
                /* 100215 */
                /* 100216 */
                /* 100217 */
                /* 100218 */
                /* 100219 */
                /* 100220 */
                /* 100221 */
                /* 100222 */
                /* 100223 */
                /* 100224 */
                /* 100225 */
                /* 100226 */
                /* 100227 */
                /* 100228 */
                /* 100229 */
                /* 100230 */
                /* 100231 */
                /* 100232 */
                /* 100233 */
                /* 100234 */
                /* 100235 */
                /* 100236 */
                /* 100237 */
                /* 100238 */
                /* 100239 */
                /* 100240 */
                /* 100241 */
                /* 100242 */
                /* 100243 */
                /* 100244 */
                /* 100245 */
                /* 100246 */
                /* 100247 */
                /* 100248 */
                /* 100249 */
                /* 100250 */
                /* 100251 */
                /* 100252 */
                /* 100253 */
                /* 100254 */
                /* 100255 */
                /* 100256 */
                /* 100257 */
                /* 100258 */
                /* 100259 */
                /* 100260 */
                /* 100261 */
                /* 100262 */
                /* 100263 */
                /* 100264 */
                /* 100265 */
                /* 100266 */
                /* 100267 */
                /* 100268 */
                /* 100269 */
                /* 100270 */
                /* 100271 */
                /* 100272 */
                /* 100273 */
                /* 100274 */
                /* 100275 */
                /* 100276 */
                /* 100277 */
                /* 100278 */
                /* 100279 */
                /* 100280 */
                /* 100281 */
                /* 100282 */
                /* 100283 */
                /* 100284 */
                /* 100285 */
                /* 100286 */
                /* 100287 */
                /* 100288 */
                /* 100289 */
                /* 100290 */
                /* 100291 */
                /* 100292 */
                /* 100293 */
                /* 100294 */
                /* 100295 */
                /* 100296 */
                /* 100297 */
                /* 100298 */
                /* 100299 */
                /* 100300 */
                /* 100301 */
                /* 100302 */
                /* 100303 */
                /* 100304 */
                /* 100305 */
                /* 100306 */
                /* 100307 */
                /* 100308 */
                /* 100309 */
                /* 100310 */
                /* 100311 */
                /* 100312 */
                /* 100313 */
                /* 100314 */
                /* 100315 */
                /* 100316 */
                /* 100317 */
                /* 100318 */
                /* 100319 */
                /* 100320 */
                /* 100321 */
                /* 100322 */
                /* 100323 */
                /* 100324 */
                /* 100325 */
                /* 100326 */
                /* 100327 */
                /* 100328 */
                /* 100329 */
                /* 100330 */
                /* 100331 */
                /* 100332 */
                /* 100333 */
                /* 100334 */
                /* 100335 */
                /* 100336 */
                /* 100337 */
                /* 100338 */
                /* 100339 */
                /* 100340 */
                /* 100341 */
                /* 100342 */
               
```

```

/* avans la caracterul urmator din expresie */
c = getchar();
} /* sfisit while 1 */
/* sfisit expresie: se trce operatorii din stiva in tefpp, daca exista */
while((j = pop()) != '\0') tefpp[i++] = j;

/* se afisaza forma poloneza postfixata a expresiei curente */
putchar('\n');
for(j=0; j<i; j++) putchar(tefpp[j]);
putchar('\n');

/* la EOF se termina executia; altfel se citeste primul caracter al expresiei urmatoare */
if(c == EOF) break;
c = getchar();
c = sca(c); /* avans peste caractere albe */

} /* sfisit for ncimbrat */
} /* sfisit main */

int sca(int x) /* salt peste caractere albe */
{
    while(x == ' ' || x == '\t' || x == '\n') x = getchar();
    return x;
}

```

8. POINTERI

Un *pointer* este o variabilă care are ca valori adrese. Pointerii se utilizează pentru a face referire la date cunoscute prin adresele lor. Astfel, dacă *p* este o variabilă de tip pointer care are ca valoare adresa lui *x*, atunci

**p*

reprezintă chiar valoarea lui *x*.

Fie de exemplu:

int x,y;

atunci dacă *p* are ca valoare adresa lui *x*, atribuirea:

y=*x*+100

este identică cu:

y=**p*+100

În mod analog, atribuirea:

x=3

este identică cu:

**p*=3

În construcția **p* utilizată mai sus, caracterul *** se consideră ca fiind un operator unar care furnizează valoarea din zona de memorie a cărei adresă este conținută în *p*.

Operatorul unar *** are aceeași prioritate ca și ceilalți operatori unari din limbajul C și se asociază de la dreapta spre stînga (vezi 3.2.16.).

Dacă *p* conține adresa zonei de memorie alocată lui *x*, vom spune că *p* *pointează spre x*.

De asemenea, dacă *p* are ca valoare adresa de început a unei zone de memorie care conține o dată de tipul *tip*, atunci vom spune că *p* *pointează spre tip*.

Menționăm că în legătură cu noțiunea de *pointer*, în limba română se utilizează și alte denumiri, ca de exemplu:

- referință;
- localizator;
- reper;
- indicator de adresă etc.

În cartea de față păstrăm denumirea englezescă.

Pentru a atribui o adresă unei variabile de tip *pointer* se poate folosi operatorul unar *&* (vezi 3.2.11.). Astfel, dacă dorim ca *p* să poarteze spre *x* (sa aibă ca valoare adresa lui *x*), atunci putem utiliza atribuirea:

$p = \&x;$

Operatorul unar & este numit operator *adresă* sau de *referențiere*. Operatorul unar * îl vom numi operator de *indirectare* sau de *dereferețiere*.

Ultima denumire a operatorului unar * decurge din efectul invers al acestuia față de operatorul unar &. Într-adevar, expresia:

$*\&x$

are aceeași valoare ca și operandul x .

Exemplu:

Fie declarația:

`int x;`

Variabilei x i se aloca o zonă de memorie de 16 biți:



Fie 1000 adresa zonei alocate variabilei x .

Dacă p este o variabilă pointer, atunci ei î se atribuie, de asemenea, o zonă de memorie de 16 biți (o să vedem ulterior că uneori această zonă va fi de 32 biți):



Folosind instrucțunea de atribuire:

$p = \&x;$

lui p î se atribuie valoarea 1000:



Instrucțunea:

$x = 10;$

atribuie lui x valoarea 10:



Același efect se obține folosind instrucțunea:

$*p = 10;$

În concluzie, instrucțunea:

$x = 10;$

este echivalentă cu secvența:

$p = \&x;$

$*p = x;$

sau

$p = \&x, *p = x;$

Noțiunea de pointer joacă un rol important deoarece permite calcule cu adrese. Acestea sunt utile mai ales în scrierea programelor de sistem. Astfel de calcule sunt proprii limbajelor de asamblare. Introducerea lor în limbajul C, oferă programatorilor posibilitatea de a scrie programe mai optime, decât cele realizate fără a beneficia de facilitățile oferite de pointeri.

Un beneficiu obținut simplu pe baza utilizării pointerilor este înlocuirea expresiilor cu indici prin expresii cu pointeri. Ca efect, înmulțirile utilizate la evaluarea variabilelor cu indici se schimbă cu adunări și deplasări. Un alt beneficiu obținut cu ajutorul pointerilor este posibilitatea alocării dinamice a memoriei prin alte mijloace decât cel utilizat la alocarea datelor automate.

Folosirea funcțiilor ca parametri este și un beneficiu al utilizării pointerilor.

În capitolul de față se descriu aspectele de bază cu privire la utilizarea pointerilor în scrierea programelor C.

8.1. Declarația de pointer și tipul pointer

Un *pointer* se declară ca orice variabilă, cu singura deosebire că numele este precedat de caracterul *. Astfel, dacă dorim să declarăm variabila p utilizată mai sus pentru a păstra adresa lui x , vom folosi declarația:

`int *p;`

Tipul *int* stabilește faptul că p conține adrese de zone de memorie în care se păstrează date de tip *int*. Declarația de mai sus se poate interpreta astfel:

$*p$ reprezintă conținutul zonei de memorie spre care pointează p , iar acest conținut are tipul *int*.

În general, un pointer se declară prin:

`tip *nume;`

ceea ce înseamnă că *nume* este un pointer care pointează spre o zonă de memorie

ce conține o dată de tipul *tip*.

Comparind declarația de mai sus cu cea obișnuită:

tip nume;

putem considera că:

*tip **

dintr-o declarație de pointeri reprezintă

tip

dintr-o declarație obișnuită. De aceea, construcția:

*tip **

se spune că reprezintă un tip nou, tipul *pointer*. Acest tip se spune că este tipul *pointer spre tip*.

Dacă avem declarațiile:

```
int x;  
int *p;  
float y;
```

atunci atribuirea

p = &x

este corectă, în timp ce

p = &y

nu este corectă, deoarece *p* poate conține numai adrese de zone de memorie în care se păstrează date de tip *int*.

Dacă există declarația:

*float *q;*

atunci se poate folosi atribuirea

q = &y

Exemplu:

```
int x,y;  
int *p;
```

1. *y = x+100;*
este echivalentă cu secvența:

```
p = &x;  
y = *p+100;
```

2. *x = y;*
este echivalentă cu secvența:

```
p = &x;  
*p = y;
```

3. *x++;*

este echivalentă cu secvența:

```
p = &x;  
(*p)++;
```

Există cazuri în care dorim ca un pointer să fie utilizat cu mai multe tipuri de date. În acest caz, la declararea lui nu putem specifica un tip, ca în exemplele de mai sus. Aceasta se realizează folosind cuvântul *void*:

*void *nume;*

Exemplu:

```
int x;  
float y;  
char c;  
void *p;  
...  
p = &x;  
...  
p = &y;  
...  
p = &c;  
...
```

Deoarece *p* a fost declarat cu ajutorul cuvântului cheie *void*, lui *p* îl se pot atribui adrese de zone de memorie care pot conține date de tipuri diferite:

int, float, char etc.

Cind se folosesc pointeri de tip *void*, este necesar să se facă conversii explicite prin expresii de tip *cast*, pentru a preciza tipul datei spre care pointează un astfel de pointer. Într-adevăr, dacă se utilizează *p* declarat ca mai sus, atunci o atribuire de forma:

**p = 10*

nu este corectă, deoarece nu este definit tipul datei spre care pointează *p*.

Pentru a putea realiza o astfel de atribuire, va trebui să convertim valoarea lui *p* spre tipul "pointer spre tipul *int*". Tipul *pointer spre int* se exprimă prin construcția:

*int **

Amintim că valoarea unui operand se poate converti spre tipul *tip* folosind operatorul unar (*tip*):

(tip) operand

De exemplu, pentru a converti valoarea lui *y*, din exemplul de mai sus, spre *long*, vom scrie:

(long) y

În cazul de față, valoarea lui *p* trebuie convertită spre tipul *int ** deci vom folosi expresia cast:

(int *) p

În felul acesta atribuirea de mai sus devine:

*(int *)p = 10

În mod analog, o expresie de forma:

*p+1.5

este eronată. Este necesar să se convertească explicit valoarea lui *p* spre tipul de date conținut în zona de memorie a cărui adresa este valoarea lui *p*. Rezultă că expresiile:

*(int *)p+1.5
*(float *)p+1.5
*(char *)p+1.5

pot fi utilizate în locul expresiei de mai sus. Expresia:

*(int *)p+1.5

se calculează astfel:

- Adresa care este valoarea lui *p* se interpretează ca fiind adresa zonei de memorie care conține o dată de tip *int*.
- Valoarea de la adresa definită de expresia:
(int *)p
se convertește din *int* spre *double* în conformitate cu regula conversiilor implicate.
- Valoarea obținută la punctul b se adună cu 1,5 și suma este de tip *double*.

Conversia tipului pointer:

void *

spre un tip pointer concret (*int *, float ** etc.) este totdeauna posibilă și nu înseamnă altceva decât precizarea tipului de pointer pe care îl are valoarea pointerului la care se aplică conversia respectivă. Deci, conversia realizată prin expresia *cast*:

(int *)p

precizează faptul că *p* are ca valoare o adresă a unei zone de memorie în care se păstrează întregii de tip *int*.

Utilizarea tipului:

void *

asigură o flexibilitate mare în utilizarea pointerelor. Cu toate acestea, se recomandă să nu se utilizeze în mod abuziv, deoarece aceasta poate fi și o sursă de erori. Într-adevăr, programatorul trebuie să știe, în fiecare moment, ce fel de tip de pointer este valoarea atribuită variabilei pointer de tip *void **.

De exemplu, dacă se utilizează expresia:

(int *)p

într-un moment în care *p* are ca valoare adresa unei zone de memorie care conține o dată flotantă, rezultatul va fi imprevizibil.

8.2. Realizarea apelului prin referință utilizând parametri de tip pointer

Am văzut că în limbajul C apelul este prin valoare. Aceasta înseamnă că la un apel, parametrilor formali li se atribuie valorile parametrilor efectivi care le corespund (vezi 4.15.).

În cazul în care un parametru efectiv este un nume de tablou, apelul prin valoare devine prin referință, parametrului formal corespunzător lui i se atribuie ca valoare adresa primului element al tabloului. Deci, dacă se face apelul:

```
int tab[...];  
...  
f(tab);  
...
```

și face antetul:

```
void f(int v[])
```

atunci la apel *v* are aceeași valoare ca *tab*. Aceasta înseamnă că *tab[0]* și *v[0]* exprimă același lucru, adică valoarea lui *tab[0]*. În general, *tab[i]* și *v[i]* au aceeași valoare și anume valoarea elementului de indice *i* al tabloului *tab*.

Folosind pointeri și în alte cazuri, putem să transformăm apelul prin valoare în apel prin referință. Astfel, dacă *x* este o variabilă simplă, atunci noi putem să transferăm, la un apel, în locul valorii lui *x*, adresa lui *x*:

```
int x;  
...  
h(x); /* se transferă valoarea lui x */  
...  
g(&x); /* se transferă adresa lui x */  
...
```

În acest caz cele două funcții vor avea antete diferite și anume:

```

void h(int i)
{
    void g(int *pi)

```

În cazul funcției *g* parametrul formal *pi* este pointer spre date de tip *int*.

La apel, lui *pi* îi se atribuie ca valoare adresa lui *x*. De aceea, funcția *g* are posibilitatea să modifice valoarea lui *x*. De exemplu, dacă în corpul funcției *g* se utilizează atribuirea:

```

...
*pi=100;
...
```

atunci valoarea 100 se atribuie variabilei *x*, a cărei adresă s-a atribuit la apel lui *pi*.

În felul acesta, apelul prin valoare se poate folosi pentru a realiza apelul prin referință.

În principiu, în limbajul C, apelul prin referință se poate realiza dacă parametrul efectiv are ca valoare o adresă, iar parametrul formal corespunzător este un *pointer*.

Ulterior o să vedem că în limbajul C++ a fost introdus apelul prin referință și el există împreună cu cel prin valoare.

Exerciții:

8.1 Să se scrie o funcție care dintr-o dată calendaristică definită prin numărul zilei din an și anul respectiv, determină luna și ziua din luna respectivă.

Această funcție a fost definită în exercițiul 6.7. Ea a avut trei parametri:

<i>zz</i>	- Ziua din an.
<i>an</i>	- Anul.
<i>tzll</i>	- Tablou de tip <i>int</i> de 2 elemente.

Funcția atribuie ziua determinată, elementului *tzll[0]*, iar luna elementului *tzll[1]*.

În funcția de mai jos se înlocuiește parametrul *tzll* cu doi parametri de tip pointer.

FUNCȚIA BVIII1

```

void pluna_si_ziua( int zz, int an,int *zi,int *luna)
/* determină luna și ziua din luna;
   zz - ziua din an;
   an - anul;
   zi - pointer a carui valoare este adresa zonei în care se păstrează ziua determinată de funcție
   luna - pointer a carui valoare este adresa zonei în care se păstrează luna determinată de funcție
*/

```

```

int bisect = an %4 == 0 && an%100 || an%400 == 0;
int i;
extern int nrzile[];

for(i = 1; zz > nrzile[i]+ (i==2 && bisect); i++)
    zz -= (nrzile[i] + (i==2 && bisect));

*zi = zz; /* păstrează valoarea lui zz în zona de memorie alocată parametrului efectiv corespunzător parametrului formal zi */

*luna = i; /* păstrează valoarea lui i în zona de memorie alocată parametrului efectiv corespunzător parametrului formal luna */

}

```

8.2 Să se scrie o funcție care afișează caracterele unui tablou și citește un întreg de tip *int*.

Funcția are doi parametri:

- | | |
|-------------|---|
| <i>text</i> | - Tablou unidimensional de tip caracter și care are aceeași utilizare ca parametrul <i>text</i> din funcția 6.3. |
| <i>x</i> | - Pointer spre întregi de tip <i>int</i> ; are ca valoare adresa zonei de memorie în care se păstrează valoarea intregului citit. |

Această funcție este asemănătoare cu funcția definită în exercițiul 6.3. Diferența constă în aceea că, funcția definită în exercițiul 6.3. atribuie numărul citit variabilei globale *v_int*, în loc să-l transfere funcției care face apelul, ca în cazul de față.

Funcția de față returnează aceleași valori ca și cea amintită mai sus, adică 0 la înțilnirea sfîrșitului de fișier și 1 altfel.

FUNCȚIA BVIII2

```

int pcit_int(char text[], int *x)
/* - citește un întreg și-l păstrează în zona de memorie a carei adresa este valoarea lui x;
   - returnează:
     0 - la întâlnirea sfîrșitului de fisier;
     1 - altfel */
{
    char t[255];

    for ( ; ; ) {
        printf(text);
        if (gets(t) == NULL) return 0;
        if (sscanf(t, "%d", x) == 1) return 1;
    }
}

```

Observație:

Deoarece *x* are ca valoare chiar adresa zonei de memorie în care se păstrează

intregul convertit din ASCII în *int* prin funcția *sscanf*, în apelul acestei funcții se folosește ca parametru efectiv chiar *x*. El nu mai trebuie să fie precedat de operatorul adresa (*&*).

- 8.3 Să se scrie o funcție care citește un întreg de tip *int* care aparține unui interval dat.

Ea are parametri:

<i>text</i>	- Tablou unidimensional de tip <i>char</i> care are aceeași utilizare ca în funcția definită în exercițiul 6.4. (funcția <i>cit_int_lim</i>).
<i>inf</i>	- Întreg de tip <i>int</i> care are aceeași utilizare ca în funcția <i>cit_int_lim</i> .
<i>sup</i>	- Întreg de tip <i>int</i> care are aceeași utilizare ca în funcția <i>cit_int_lim</i> .
<i>pint</i>	- Pointer spre intregi de tip <i>int</i> ; are ca valoare adresa zonei de memorie în care se păstrează numărul citit.

Ca și funcția *cit_int_lim*, funcția de față returnează:

<i>0</i>	- La înțlnirea sfîrșitului de fișier.
<i>1</i>	- Altfel.

FUNCȚIA BVIII3

```
int pcit_int(char [], int *); /* prototip */
int pcit_int_lim(char text[], int inf,int sup, int *pint)
/* - citește un întreg de tip int ce aparține intervalului [inf,sup] și-l păstrează în zona de memorie a cărei adresa este valoarea parametrului pint;
   - returnează:
     0 - la înțlnirea sfîrșitului de fișier;
     1 - altfel. */
{
    for( ; ; ) {
        if(pcit_int(text, pint) == 0) return 0; /* s-a întlnit EOF */
        if(*pint >= inf && *pint <= sup) return 1;
        printf("intregul tastat nu aparține intervalului:\n");
        printf("[%d,%d]\n", inf,sup);
        printf("se reia citirea\n");
    }
}
```

- 8.4 Să se scrie o funcție care citește o dată calendaristică compusă din *zi*, *luna* și *an*.

Funcția validează data calendaristică respectivă. Ea are trei parametri de tip pointer:

<i>pzi</i>	- Are ca valoare adresa zonei de memorie în care se păstrează
------------	---

numărul zilei.

- Are ca valoare adresa zonei de memorie în care se păstrează numărul lunii.
- Are ca valoare adresa zonei de memorie în care se păstrează anul.

Funcția returnează:

<i>0</i>	- Dacă s-a întlnit sfîrșitul de fișier.
<i>1</i>	- Altfel.

Pentru validarea datei calendaristice se apelează funcția *v_calend* definită în exercițiul 6.5.

Funcția de față este analogă cu cea definită în exercițiul 6.8. Diferența constă în utilizarea parametrilor și a funcției care citește cele 3 valori din compunerea datei calendaristice. Astfel, funcția de față utilizează trei parametri de tip pointer spre *int*, pentru transferul celor trei valori citite, spre deosebire de funcția *cit_data_calend* definită în exercițiul 6.8., care utilizează în același scop un tablou de tip *int*. De asemenea, funcția de față apelează funcția *pcit_int_lim* pentru a citi cele trei valori ale datei calendaristice, spre deosebire de funcția *cit_data_calend* care utilizează funcția *cit_int_lim*.

FUNCȚIA BVIII4

```
int pcit_int_lim( char [], int,int, int *); /* prototip */

int pcit_data_calend( int *pzi,int *pluna,int *pan)
/* - citește o data calendaristica, o validează și o păstrează în zonele de memorie a caror adrese sunt definite de căi trei parametri formalii de tip pointer;
   - funcția returnează:
     0 - la înțlnirea sfîrșitului de fisier;
     1 - altfel. */
{
    static char ziua[] = "ziua: ";
    static char luna[] = "luna: ";
    static char an[] = "anul: ";
    static char er[] = "s-a tastat EOF";

    for( ; ; ) {
        /* se citește ziua */
        if(pcit_int_lim(ziua,1,31,pzi) == 0) {
            printf("%s\n",er);
            return 0;
        }

        /* se citește luna */
        if(pcit_int_lim(luna,1,12,pluna) == 0) {
            printf("%s\n",er);
            return 0;
        }

        /* se citește an */
        if(pcit_int_lim(an,1,99,pan) == 0) {
            printf("%s\n",er);
            return 0;
        }
    }
}
```

```

    }

    /* se citeste anul */
    if(pcit_int_lim(an,1600,4900,pan) == 0 ) {
        printf("%s\n",er);
        return 0;
    }

    /* validare data calendaristica */
    if(v_calend(*pzi, *pluna, *pan)) return 1;
    printf("data calendaristica este eronata\n");
    printf("se reia citirea datei calendaristice\n");
}
}

```

8.5 Să se scrie un program care citește o dată calendaristică și afișează data calendaristică pentru ziua următoare.

Acest program este analog cu cel definit în exercițiul 6.9. Programul de față utilizează funcții ce au ca parametri pointeri, spre deosebire de cel definit în exercițiul 6.9., care apelează funcții ce utilizează variabila globală *v_int* și parametrii de tip tablou.

PROGRAMUL BVIII5

```

#include <stdio.h>
#include <stdlib.h>
#include "bviii2.cpp" /* functia pcit_int */
#include "bviii3.cpp" /* functia pcit_int_lim */
#include "bvi5.cpp" /* functia v_calend */
#include "bvi6.cpp" /* functia zi_din_an */
#include "bviii1.cpp" /* functia pluna_si_ziua */
#include "bviii4.cpp" /* functia pcit_data_calend */

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

main() /* citeste o data calendaristica, o valideaza si in caz ca este
corecta, afiseaza data calendaristica a zilei urmatoare */
{
    int zz,ll,aa;

    /* citeste si valideaza data calendaristica */
    if(pcit_data_calend(&zz,&ll,&aa) == 0) exit(1);

    if(zz == 31 && ll == 12) {
        /* 31 decembrie; ziua urmatoare este 1 ianuarie din anul urmator */
        zz = 1;
        ll = 1; /* ianuarie */
        aa++; /* anul urmator */
    }
    else /* se determina ziua urmatoare */
        pluna_si_ziua(zi_din_an(zz,ll,aa)+1,aa,&zz,&ll);
}

```

```

/* afiseaza data calendaristica a zilei urmatoare */
printf("ziua:%d\tluna:%d\tanul:%d\n",zz,ll,aa);
}

```

Observație:

Parametrii efectivi *&zz*, *&ll*, *&aa* de la apelul funcției *pcit_data_calend* atribuie parametrilor formali corespunzători, adretele zonelor de memorie alocate variabilelor *zz*, *ll*, și respectiv *aa*.

La revenirea din funcția *pcit_data_calend* zonele de memorie alocate acestor variabile vor conține valorile corespunzătoare zilei, lunii și anului care au fost citite prin apelul respectiv (dacă nu s-a tastat sfîrșitul de fișier).

8.6 Să se scrie o funcție care citește:

- valoarea variabilei *m* de tip *int*;
- valoarea variabilei *n* de tip *int*;
- *m*n* numere care reprezintă elementele unei matrice de ordinul *m*n*.

Funcția are următorii parametri:

<i>dmat</i>	- Tablou unidimensional de tip <i>double</i> în care se păstrează elementele matricei prin liniarizare.
<i>max</i>	- Maximul produsului <i>m*n</i> admis.
<i>nrlin</i>	- Pointer a căruia valoare este adresa zonei de memorie în care se păstrează valoarea lui <i>m</i> .
<i>nrcol</i>	- Pointer a căruia valoare este adresa zonei de memorie în care se păstrează valoarea lui <i>n</i> .

Această funcție este asemănătoare cu funcția *gdcimat* definită în exercițiul 5.1. Diferența dintre ele constă în faptul că funcția *gdcimat* nu utilizează parametri de tip pointer pentru a transfera valorile lui *m* și *n* în afara funcției, ci două variabile globale.

FUNCȚIA BVIII6

```

int ndcit(int, double []); /* prototip */

int pdcitmat(double dmat[],int max,int *nrlin,int *nrcol)
/* - citeste pe:
   m-numar de linii;
   n-numar de coloane;
   - m*n numere de tip double pe care le pastreaza in matricea dmat prin liniarizare;
   - returneaza valoarea m*n;
   m - se pastreaza in zona de memorie definita de parametru nrlin;
   n - se pastreaza in zona de memorie definita de parametru nrcol. */
{
    int i;
    char t[255];
    char er[]="s-a tastat EOF\n";

```

```

do { /* se citesc valorile lui m si n */
    do { /* citeste pe m */
        printf("numarul de liniile= ");
        if(gets(t) == NULL) {
            printf(er);
            exit(1);
        }
        if(sscanf(t,"%d", &nrlin)==1 && *nrlin > 0 &&
           *nrlin <= max)
            break;
        printf("nu s-a tastat un intreg in\"
               intervalul [1,%d]\n", max);
    } while(1);

    do { /* citeste pe n */
        printf("numarul de coloane= ");
        if(gets(t) == NULL) {
            printf(er);
            exit(1);
        }
        if(sscanf(t,"%d", &nrcol)==1 && *nrcol > 0 &&
           *nrcol <= max)
            break;
        printf("nu s-a tastat un intreg in\"
               intervalul [1,%d]\n", max);
    } while (1);

    i = *nrlin * *nrcol;
    if(i <=max) break;
    printf("produsul m*n=%d depaseste pe max=%d\n", i, max );
    printf("se reiau citirile lui m si n\n");
} while(1);

/* se citesc elementele matricei tastate pe linii */
if(ndcit(i,dmat) != i) {
    printf("nu s-au tastat %d elemente\n", i);
    exit(1);
}
return i;
}

```

Observații:

1. Funcția *ndcit* este definită în exercițiul 4.37.
2. Citirea numerelor *m* și *n* se poate realiza mai simplu folosind funcția *pcit_int_lim* definită în exercițiul 8.3.
- 8.7 Să se scrie un program care citește elementele de tip *double* ale două matrice *a* și *b*, calculează produsul lor și-l afișează.

Dacă

$$c=a*b$$

atunci

$$c[i,j]=a[i,0]*b[0,j]+a[i,1]*b[1,j]+\dots+a[i,n-1]*b[n-1,j]$$

pentru

$$i=0,1,\dots,m-1; j=0,1,\dots,s-1$$

unde:

- a* - Este o matrice de ordinul *m*n*.
- b* - Este o matrice de ordinul *n*s*.

Matricea produs *c* este de ordinul *m*s*.

PROGRAMUL BVIII7

```

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include "biv37.cpp" /* functia ndcit */
#include "bviii6.cpp" /* functia pdcitmat */

#define MAX 900

main() /* citeste elementele a doua matrice, calculeaza si afiseaza
          matricea produs, cite 4 elemente pe o linie */
{
    int i,j,k;
    int m,n,p,s,r,q;
    int mn,np;
    double a[MAX],b[MAX],c[MAX];

    /* citeste matricea a */
    mn = pdcitmat(a,MAX, &m, &n);
    /* citeste matricea b */
    np = pdcitmat(b,MAX,&p,&s);
    if(n != p) {
        printf("numarul coloanelor matricei a = %d \n",n);
        printf("difera de numarul liniilor ");
        printf("matricei b = %d \n",p);
        exit(1);
    }

    /* se realizeaza produsul c = a*b */
    for(i=0; i < m; i++) {
        q = i*n;
        for(j=0; j < s; j++) {
            r = i*s + j;
            c[r] = 0.0;
            for(k=0; k < n; k++) c[r] += a[q+k]* b[k*s+j];
        }
    }
}

```

```

/* afiseaza elementele matricii produs */
printf("\n\n\tmatricea produs\n");
k=1;
for(i=0;i<m;i++) {
    p = i*s;
    for(j=0;j<s;j++) {
        printf("c[%d,%d]=%8g ",i,j,c[p+j]);
        if(j%4 == 3) {
            /* afiseaza 4 elemente pe un rind */
            printf("\n");
            k++; /* numara randurile */
        }
    }
    if(k==23) {
        printf("actionati o tasta pentru a continua\n");
        getch();
        k=1;
    }
}
printf("\n");
k++;
}

```

8.3. Legătura dintre pointeri și tablouri

Numele unui tablou este un *pointer* deoarece el are ca valoare adresa primului său element. Totuși există o diferență între numele unui tablou și o variabilă de tip pointer. Unei variabile de tip pointer îi se atribuie valori la execuție, în timp ce acesta nu este posibil să se realizeze pentru numele unui tablou. Acesta tot timpul are ca valoare adresa primului său element. De aceea, se obișnuiește să se spună că numele unui tablou este un *pointer constant*.

Exemplu:

```

int t[10];
int *p;
int x;
...
p=t; /* în urma acestei atribuiri p are același valoare ca și t, adică adresa lui t[0]*/
x=t[0];

```

și

```
x=*p;
```

au același efect: atribuie lui x valoarea elementului t[0].

Un parametru formal ce corespunde unui parametru efectiv care este un nume de tablou unidimensional, poate fi declarat fie ca *tablou*, fie ca *pointer* spre tipul tabloului.

Fie declarația:

```
int tab[10];
```

și apelul:

```
f(tab);
```

Funția f poate avea unul din urmatoarele antete:

```
void f(int t[])
```

sau

```
void f(int *t)
```

Intr-adevăr, declarația:

```
int t[]
```

defineste pe t ca nume de tablou, iar parametrului t îi se atribuie la apel valoarea lui tab, adică adresa lui tab[0]. În felul acesta, construcțiile tab[i] și t[i] reprezintă unul și același element al i+1 - lea al tabloului tab.

Parametrul formal t declarat în acest fel este deci un pointer (are ca valoare o adresă) spre int. Spre deosebire de tab, t este chiar un pointer variabil. El este alocat pe stivă și lui îi se atribuie o valoare prin apelul funcției. Ulterior o să vedem că noi putem să modificăm valoarea lui t. Nu același lucru se poate spune despre tab. Aceasta este un pointer constant, a carui valoare nu poate fi schimbată la execuție.

Din cele de mai sus rezultă că parametrul t este un pointer variabil spre int și deci el poate fi declarat printr-o declarație de forma:

```
int *t
```

În concluzie, dacă un parametru formal corespunde unui parametru efectiv care este nume de tablou unidimensional, atunci el poate fi declarat fie ca tablou:

tip nume_parametru_formal[]

fie ca pointer:

*tip *nume_parametru_formal*

Cele două declarații sunt echivalente și de aceea, ambele declarații permit ca în corpul funcției să se utilizeze construcția:

nume_parametru_formal[indice]

8.4. Operații cu pointeri

Asupra pointerilor se pot face diferite operații pe care le precizăm în continuare.

8.4.1. Operații de incrementare și decrementare a pointerilor

Operatorii `++` și `--` se pot aplica la operanzi de tip pointer. El se execută altfel decit asupra datelor care nu sunt pointeri.

Operatorul de incrementare (`++`) aplicat unui operand de tip pointer spre tipul *t*, mărește adresa care este valoarea operandului, cu numărul de octeți necesari pentru a păstra o dată de tip *t*.

Operatorul de decrementare are un efect similar, diferența constind în aceea că în acest caz valoarea operandului se micșorează cu numărul de octeți necesari pentru a păstra o dată de tipul spre care pointează operandul.

De exemplu, dacă avem declarația:

```
int *p;
```

Atunci expresiile:

`++p`

și

`p++`

măresc valoarea lui *p* cu 2, deoarece o dată de tip *int* se păstrează pe 2 octeți.

În mod analog, expresiile:

`--p`

și

`p--`

micșorează valoarea lui *p* cu 2.

Aceste operații sunt utile cînd se au în vedere prelucrările de date de tip tablou.

Exemplu:

```
double tab[10];
double *p; int i;
```

Fie:

`p = &tab[i];`

unde $0 < i < n-1$. Instrucțiunile expresie:

`p++;` și `++p;`

măresc valoarea lui *p* cu 8, deci *p* va avea ca valoare adresa următoare elementului `tab[i]`, adică adresa elementului `tab[i+1]`.

În mod analog, instrucțiunile expresie:

`p--;` și `--p;`

micșorează valoarea lui *p* cu 8, deci *p* va avea ca valoare adresa elementului

precedent lui `tab[i]`, adică adresa elementului `tab[i-1]`.

8.4.2. Adunarea și scăderea unui întreg dintr-un pointer

Dacă *p* este un pointer spre tipul *t* și *n* un întreg, atunci se pot utiliza expresiile:

`p+n`

și

`p-n`

Expresia:

`p+n`

are ca valoare, valoarea lui *p* mărită cu produsul *r*n*, unde prin *r* am notat numărul de octeți necesari pentru a păstra în memorie o dată de tipul *t*.

În mod analog, valoarea expresiei

`p-n`

este valoarea lui *p* micșorată cu produsul *r*n*.

Fie *tab* un tablou unidimensional de tip *t*. Atunci *tab* este un pointer (constant) și deci o expresie de forma:

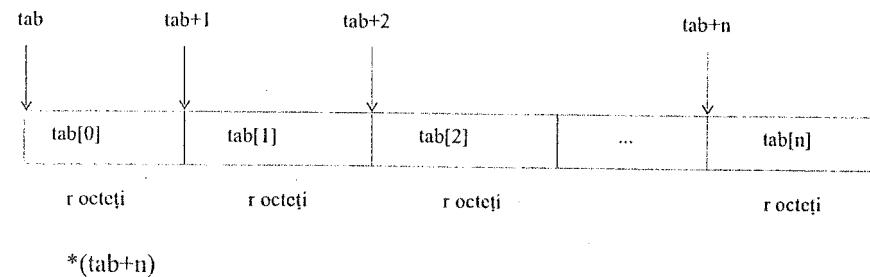
`tab+n`

este corectă. Conform celor spuse mai sus, rezultă că această expresie este chiar adresa elementului `tab[n]`. Într-adevăr, *tab* este adresa lui `tab[0]`. Atunci `tab+1` are ca valoare adresa lui `tab[0]` mărită cu *r*, *r* fiind numărul de octeți alocați pentru o dată de tip *t*, adică numărul de octeți ocupati de elementul `tab[0]`.

Deci `tab+1` are ca valoare adresa elementului `tab[1]`.

În general, `tab+n` va avea ca valoare adresa elementului `tab[n]`.

Deoarece `tab+n` este adresa elementului `tab[n]`, rezultă că



are ca valoare chiar valoarea elementului `tab[n]`. Deci, o atribuire de forma:

`x = tab[n]`

este echivalentă cu:

`x = *(tab+n)`

În general, *variabilele cu indici* se pot înlocui prin *expresii cu pointeri*. Astfel de înlocuiră conduce la optimizări înlocuindu-se operațiile de înmulțire care intervin la evaluarea variabilelor cu indici, prin adunări. De aceea, se recomandă utilizarea expresiilor cu pointeri în locul variabilelor cu indici.

8.4.3. Compararea a doi pointeri

Doi pointeri care pointează spre elementele aceluiași tablou pot fi comparați folosind operatorii de relație și de egalitate.

Astfel, dacă p pointează spre elementul $t[i]$ al tabloului t (adică are ca valoare adresa elementului $t[i]$) și q spre elementul $t[j]$ al aceluiași tablou, atunci expresiile:

$p < q$, $p \leq q$, $p \geq q$, $p > q$, $p == q$ și $p != q$

sunt legale. De exemplu, expresia:

$p < q$

are valoarea *adevărat* (1), dacă $i < j$ și *fals* (0) în caz contrar.

În mod analog se evaluatează și celelalte expresii de mai sus.

Operatorii de egalitate ($==$ și $!=$) pot fi folosiți pentru a compara pointerii cu o constantă specială *NULL*. Aceasta este definită în fișierul *stdio.h* astfel:

```
#define NULL 0
```

Ea reprezintă așa numitul *pointer nul*.

Dacă p este un pointer spre orice tip, atunci se pot face comparații de forma:

$p == NULL$

și

$p != NULL$

În limbajul C++ se recomandă să nu se utilizeze constanta *NULL*, ci chiar valoarea ei zero:

$p == 0$

și

$p != 0$

sau echivalentele lor:

$!p$

și

p

Într-adevăr, $p == 0$ are valoarea adevarat atunci și numai atunci cind $!p$ are valoarea adevarat.

Aceleași lucru se poate spune și despre celelalte două expresii:

$p != 0$

are valoarea fals atunci și numai atunci cind p are valoarea zero, adică cind p este fals.

În general, dacă un pointer are valoarea zero (pointerul nul), înseamnă că el nu definește o adresă. Așa cum o să vedem mai târziu, astfel de teste sunt uneori necesare pentru a pune în evidență anumite erori.

Tinând seama de faptul că *NULL* nu se mai utilizează în programarea curentă în limbajul C++, nu o vom utiliza nici la scrierea programelor în C. În prezent am utilizat constanta *NULL* la apelul funcției *gets*. Aceasta returnează pointerul spre zona în care se păstrează sirul de caractere citit. Deoarece la întâlnirea sfîrșitului de fișier, conținutul acestei zone nu este definit, funcția *gets* returnează zero în acest caz.

8.4.4. Diferența a doi pointeri

Doi pointeri care pointează spre elementele aceluiași tablou pot fi scăzuți. Fie p un pointer spre elementul $t[i]$ al tabloului t și q un pointer spre elementul $t[i+n]$ al aceluiași tablou. Atunci diferența:

$q - t$

are valoarea n .

Exerciții:

- 8.8 Să se scrie o funcție care citește cel mult n elemente de tip *double* și le păstrează în zona de memorie a cărei adresă de început este valoarea parametrului formal p al funcției. Funcția returnează numarul numerelor citite.

Funcția de față este analogă cu funcția *ndcit* definită în exercițiul 4.37. În acest caz se utilizează expresii cu pointeri în locul indiciilor.

FUNCȚIA BVIII8

```
int pdcit(int n, double *p)
/* - citește cel mult n numere și le păstrează în zona spre care pointează p;
   - returnează numarul numerelor citite. */
{
    double d;
    char t[255];
    int i = 0;
```

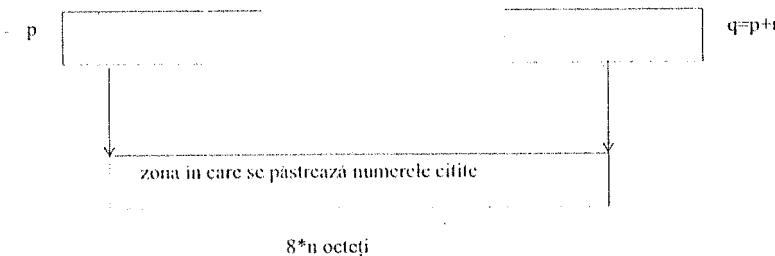
```

double *q = p+n;
while( p < q ) {
    printf("elementul (%d)= %.1f\n");
    if(gets(t) == NULL) return i;
    if(sscanf(t,"%lf", &d) != 1) break;
    *p++ = d;
    i++;
}
return i;
}

```

Observații:

- Parametrul *p* are ca valoare la apel, adresa de inceput a zonei in care se păstrează numerele citite. Dimensiunea acestei zone este de $8*n$ octeți, pentru a putea păstra cel mult *n* numere de tip *double*.



- Instrucțiunea:
 **p++=d;*
memorează numărul citit în zona spre care pointează *p*.

Totodată pointerul *p* se incrementează, deci valoarea lui se mărește cu 8. În felul acesta, la iterarea următoare, *p* pointează spre zona în care urmează să fie memorat elementul următor.

După păstrarea a *n* numere, *p* devine egal cu valoarea lui *q*. Deci ciclul *while* în acest moment trebuie să se termine, dacă nu cumva s-a terminat înainte, deoarece s-au tastat mai puțin de *n* numere.

- Să se scrie o funcție care citește componentele unui vector precedate de numărul lor. Funcția returnează numărul componentelor vectorului respectiv.

Numărul componentelor se citește apelând funcția *pcit_int_lim* definită în exercițiul 8.3.

Componentele vectorului se citesc apelând funcția *pndcit* definită în exercițiul 8.8.

FUNCȚIA BVIII9

```

int pdvcit(int nmax, double dvector[])
/* - citește un întreg n și cele n componente ale vectorului dvector;
   - returnează valoarea lui n. */
{
    int i,n;
    char t[255];

    if(pcit_int_lim("numarul componentelor=",1,
                     nmax,&n) == 0 ) {
        printf("s-a tastat EOF\n");
        exit(1);
    }
    if((i=pndcit(n,dvector)) != n) {
        printf("nu s-a reusit citirea celor n=%d \
               componente\n", n );
        printf("s-au citit numai %d componente\n",i);
    }
    return i;
}

```

Observație:

Funcția *pcit_int_lim* a fost apelată folosind șirul de caractere:
"numarul componentelor="

ca primul parametru efectiv al apelului. Acest parametru este păstrat de compilator într-o zonă specială rezervată șirului de caractere, iar la apel, se atribuie parametrului formal corespunzător, adresa de inceput a zonei în care se păstrează șirul respectiv.

- Să se scrie un program care citește componentele vectorilor *x* și *y*, calculează și afișează produsul lor scalar.

Componentele celor doi vectori sunt precedate de numărul lor.

Programul este similar cu cel definit în exercițiul 4.39. În cazul de față se folosesc expresii cu pointeri în locul variabilelor cu indici.

PROGRAMUL BVIII10

```

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

#include "bviii2.cpp" /* pcit_int */
#include "bviii3.cpp" /* pcit_int_lim */
#include "bviii8.cpp" /* pndcit */
#include "bviii9.cpp" /* pdvcit */

#define MAX 1000

```

```

main() /* - citeste componentele vectorilor x si y precedate de numarul lor;
           - calculeaza si afiseaza produsul lor scalar. */
{
    int i,n;
    double *p,*q;
    double x[MAX],y[MAX],s;

    /* citeste pe n si componentele vectorului x */
    n = pdvcit(MAX,x);

    /* citeste componentele vectorului y */
    if(pndcit(n,y) != n) {
        printf("nu sunt n=%d componente pentru y\n",n);
        exit(1);
    }

    /* calculeaza produsul scalar */
    s = 0.0;
    for(p=x,q=y, i=0; i < n; i++,p++,q++) s += *p * *q;
    printf("(x,y)= %g\n", s);
}

```

8.5. Modificatorul const

Am văzut în capitolul 1 că o constantă se poate defini prin caracterele care o compun.

Într-adevăr, caracterele din compunerea unei constante definesc atât tipul cît și valoarea acesteia.

Exemple:

1234	constantă de tip <i>int</i>
1.5	constantă de tip <i>double</i>
'a'	constantă caracter
"a"	constantă sir

Tot în capitolul 1 s-au definit *constantele simbolice*. O astfel de constantă se definește cu ajutorul construcției `#define` tratată de preprocesor. Ea are următorul format:

`#define nume succesiune_de_caractere`

Prinț-o astfel de construcție se atribuie un nume unei constante.

La preprocesare, *nume* se înlocuiește peste tot prin *succesiune_de_caractere*, exceptind cazurile cînd *nume* se află într-un comentariu sau sir de caractere.

La compilare, *nume* definit prinț-o astfel de construcție `#define` nu există în afara sirurilor de caractere și a comentariilor. De aceea, valoarea lui nici nu poate fi modificată la execuția programului.

În limbajele C și C++ există posibilitatea de a defini date constante folosind modificatorul *const* în declarații. Prinț-o astfel de declarație, unui nume i se

poate atribui o valoare inițială care nu poate fi modificată, în mod obișnuit, printr-o expresie de atribuire, ca în cazul variabilelor. De aceea, o dată declarata cu ajutorul modificatorului *const* se consideră că este o *constantă*.

Formatele posibile ale unei declarații cu modificatorul *const* sunt:

```

tip const nume = valoare;
tip const *nume = valoare;
const tip nume = valoare;
const tip *nume;
const nume = valoare;

```

O declarație de forma celor de mai sus se spune că este o *declarație de constantă*.

O declarație de constantă de formă:

(1) `tip const nume = valoare;`

este asemănătoare cu o declarație de variabilă obișnuită de formă:

(2) `tip nume = valoare;`

Diferența dintre cele două declarații constă în aceea că valoarea lui *nume* atribuită prin declarația (1) nu poate fi schimbată folosind o expresie de atribuire de formă:

nume = *expresie*

în timp ce aceasta este posibil pentru *nume* declarat prin declarația (2).

Exemple:

1. `int const i = 10;`
i este o constantă care are valoarea 10. O expresie de atribuire de formă
`i = 3`
este eronată.
Constanta *i* declarată ca mai sus diferează față de o constantă definită prin `#define`, deoarece ea nu este prelucrată de preprocesor. Numele ei există la compilare.

O expresie de formă:

`y = i+7`

este corectă și va atribui lui *y* valoarea 17.

2. `double const pi = 3.14159265;`
declară numele *pi* ca fiind o constantă de tip *double* și care are valoarea 3.14159265.

O atribuire de formă:

`pi = 3.1415`

generează o eroare la compilare.

pi poate fi utilizat în expresii de formă:

`pi*r*r`

```

sin(pi/2+x)
cos(x-pi)

3. char *compt_s = "aii";

```

Aici *tip* este *char **, deci *s* este un *pointer constant* spre zona în care se păstrează sirul de caractere format din literele *s*, *i*, *r* și caracterul *NUL*. În acest caz valoarea lui *s* nu poate fi schimbată. El fiind pointer, are ca valoare o adresă a unei zone de memorie de dimensiune egală cu 4 octeți. În aceasta zona se păstrează sirul de caractere indicat mai sus. Conținutul acestei zone poate fi modificat. Astfel, în timp ce o atribuire de forma *s = t*; unde *t* este un pointer spre caractere, nu este acceptată de compilator, atribuirile:

```

*s = '1';
*(s+1) = '2';

```

sunt corecte. Cu ajutorul lor se schimbă caracterul *s* cu 1 și *i* cu 2 în zona spre care pointează *s*.

Declarația:

(3) *tip const *nume = valoare;*

definește pe *nume* ca un pointer spre o zonă constantă. În acest caz, valoarea pointerului *nume* se poate schimba. De exemplu, dacă se consideră declarația:

```
char const *s = "sir";
```

atunci atribuirea:

```
s = t;
```

unde *t* este un pointer spre *char* este corecta. În schimb atribuirile:

```

*s = 'a'
*(s+1) = 'b'

```

sunt eronate, deoarece *s* pointează spre o zonă în care se păstrează o dată constantă. Cu toate acestea, constanta respectivă poate fi modificată folosind un pointer diferit de *s*:

```

char *p;
p = (char *)s; /* p, ca și s, pointează spre zona în care se păstrează sirul de caractere "sir" */
*p = 'a';
*(p+1) = 'b';

```

Acstea atribuiriri sunt corecte, deoarece *p* nu mai este un pointer spre o zonă constantă.

Declarația:

(4) *const tip nume = valoare;*

este identică cu declarația (1) dacă *tip* nu este un tip pointer.

Dacă *tip* este un tip pointer, adică declarația (4) este de forma:

(5) *const tip *nume = valoare;*

atunci ea este identică cu declarația (3) de mai sus. De obicei, se utilizează formatul (5), în locul formatului (3) pentru a declara un pointer spre o zonă constantă (a cărei valoare nu poate fi modificată direct, adică prin atribuire în care se folosește *nume*):

```

*nume = ...
*(nume+k)=...

```

Declarația

(6) *const tip *nume*

se utilizează pentru a declara un parametru formal.

Fie funcția *f* de antet:

*tip f(tip *nume)*

La apelul funcției *f*, parametrul formal *nume* își se atribuie ca valoare o adresă. Am văzut că acest fapt dă posibilitatea funcției *f* să modifice *data* păstrată în zona de memorie spre care pointează *nume* folosind o atribuire de forma:

**nume = valoare*

Există cazuri în care funcția apelată în acest fel nu are voie să modifice *data* din zona de memorie a cărei adresă este atribuită unui parametru formal. Ea trebuie numai să aibă acces la *data* respectivă. Pentru a proteja *data* față de eventualele atribuiriri neautorizate, vom declara parametrul formal respectiv ca un pointer spre o dată constantă. În acest caz, antetul funcției *f* devine:

*tip f(const tip *nume)*

În acest caz o atribuire de forma:

**nume = valoare*

este interzisă. De aceea, declarația:

*const tip *nume*

se recomandă a fi utilizată pentru orice parametru formal care la apel are ca valoare adresa unei zone de memorie al cărui conținut nu poate fi modificat de funcția apelată.

8.6. Funcții standard utilizate la prelucrarea șirurilor de caractere

Biblioteca standard conține o serie de funcții care permit operații cu șiruri de caractere. Majoritatea acestor funcții au prototipul în fișierul *string.h*. Mai jos prezentăm funcțiile mai importante din această clasă.

Un șir de caractere se păstrează într-o zonă de memorie organizată ca tablou unidimensional de tip *char*. Fiecare caracter se păstrează pe cîte un octet prin codul său numeric.

Cel mai frecvent cod utilizat în acest scop este codul ASCII.

După ultimul caracter al șirului se păstrează caracterul *NUL* ('\0').

Pentru a opera cu un șir de caractere se poate utiliza *numele tabloului* ale căruia elemente au ca valori codurile caracterelor șirului respectiv.

Spunem despre acest *nume* că este un pointer constant spre șirul respectiv.

Evident, se pot utiliza și pointeri variabili spre un șir de caractere.

Exemplu:

```
char tab[] = "Acesta este un sir";
```

Șirul de caractere "Acesta este un sir" se păstrează în zona de memorie alocată lui *tab*.

tab are ca valoare adresa de început a zonei de memorie în care se păstrează caracterele șirului.

tab - Adresa caracterului *A*.

tab+1 - Adresa caracterului *c*.

tab+2 - Adresa caracterului *e*.

etc.

tab[0] - Codul ASCII al caracterului *A*.

tab[1] - Codul ASCII al caracterului *c*.

etc.

**tab* - Codul ASCII al caracterului *A*.

**(tab+1)* - Codul ASCII al caracterului *c*.

etc.

Un efect similar se obține cu ajutorul declarației:

```
char *const p = "Acesta este un sir";
```

Șirul de caractere se păstrează într-o zonă de memorie rezervată pentru a păstra șiruri de caractere.

Adresa de început a zonei în care se păstrează șirul de față se atribuie pointerului *p*. Acesta, ca și *tab*, este un pointer constant.

<i>p</i>	- Adresa caracterului <i>A</i> .
<i>p+1</i>	- Adresa caracterului <i>c</i> .
<i>p+2</i>	- Adresa caracterului <i>e</i> .
etc.	
<i>p[0] sau *(p)</i>	- Codul ASCII al caracterului <i>A</i> .
<i>p[1] sau *(p+1)</i>	- Codul ASCII al caracterului <i>c</i> .
etc.	

În legătură cu șirurile de caractere se au în vedere operații de următorul fel:

- calculul lungimii unui șir de caractere;
- copierea șirurilor de caractere;
- concatenarea șirurilor de caractere;
- compararea șirurilor de caractere.

Funcțiile standard prin care se realizează aceste operații au fiecare un nume care începe cu prefixul *str* (prescurtare de la *string*).

8.6.1. Lungimea unui șir de caractere

Lungimea unui șir de caractere se definește prin numărul de caractere proprii care intră în compunerea șirului respectiv. Caracterul *NUL* este un caracter impropriu și el nu este considerat la determinarea lungimii unui șir de caractere. Prezența lui este însă necesară, deoarece la determinarea lungimii unui șir se numără caracterele acestuia pînă la întlnirea caracterului *NUL*.

Funcția pentru determinarea lungimii unui șir de caractere are prototipul:

```
unsigned strlen(const char *s);
```

Exemplu:

```
1. char *const p = "Acesta este un sir";
   unsigned n;
```

```
   ...
   n = strlen(p);
```

Lui *n* îi se atribuie valoarea 18 (numărul caracterelor proprii din compunerea șirului spre care pointează *p*).

```
2. char tab[] = "Acesta este un sir";
   int n;
   n = strlen(tab);
```

Variabila *n* primește aceeași valoare ca în exemplul precedent.

```
3. int n;
   n = strlen("Acesta este un sir");
```

Lui *n* îi se atribuie aceeași valoare ca în exemplele precedente.

Observație:

Parametrul formal al funcției *strlen* este un pointer spre o dată constantă deoarece funcția *strlen* nu are voie să modifice caracterele șirului pentru care determină lungimea.

8.6.2. Copierea unui șir de caractere

Adesea este nevoie să se copieze un șir de caractere din zona de memorie în care se află, într-o altă zonă.

În acest scop se poate folosi funcția de prototip:

```
char *strcpy(char *dest, const char *sursa);
```

Funcția copiază șirul de caractere spre care pointează *sursa* în zona de memorie a cărei adresa de început este valoarea lui *dest*.

Funcția copiază atât caracterele proprii șirului, cit și caracterul *NUL* de la sfîrșitul șirului respectiv.

Se presupune că zona de memorie în care se face copierea este destul de mare pentru a putea păstra caracterele copiate. În caz contrar se alterează datele pastrate imediat după zona rezervată la adresa definită de parametrul *dest*.

La revenire, funcția returnează adresa de început a zonei în care s-a transferat șirul, adică chiar valoarea lui *dest*. Această valoare este pointer spre caractere deci tipul returnat de funcție este:

```
char *
```

De aceea, prototipul funcției *strcpy* începe prin construcția *char ** (tipul valorii returnate de funcție este pointer spre *char*).

Se observă că parametrul *sursa*, care definește zona în care se află șirul ce se copiază, este declarat prin modificatorul *const*. Aceasta deoarece funcția *strcpy* nu are voie să modifice șirul care se copiază. În schimb, parametrul *dest* nu este declarat cu modificatorul *const* deoarece funcția *strcpy* modifică zona spre care pointează *dest* (în ea se copiază caracterele șirului).

Exemple:

1.

```
char tab[] = "Acest sir se copiaza";
char t[sizeof tab]; /* are același număr de elemente ca și tab */
...
strcpy(t,tab); /* șirul pastrat în tab se copiază în zona alocată lui */
```
2.

```
char t[100];
strcpy(t,"Acest sir se copiaza");
```
3.

```
char *p = "Acest sir se copiaza";
char t[100];
char *q;
```

```
q = strcpy(t,p);
```

Șirul păstrat în zona spre care pointează *p* se transferă în zona spre care pointează *t*. Valoarea lui *t* se atribuie lui *q*.

Pentru a copia cel mult *n* caractere ale unui șir dintr-o zonă de memorie în alta, se va folosi funcția de prototip:

```
char *strncpy(char *dest, const char *sursa, unsigned n);
```

Dacă *n > lungimea șirului* spre care pointează *sursa*, atunci toate caracterele șirului respectiv se transferă în zona spre care pointează *dest*. Altfel se copiază numai primele *n* caractere ale șirului. În rest, funcția *strncpy* are același efect ca și *strcpy*.

Exemplu:

```
char *p = "Acest sir se copiaza truncheat";
char t[10];
strncpy(t,p,sizeof t);
```

8.6.3. Concatenarea șirurilor de caractere

Bibliotecile limbajelor C și C++ conțin o funcție care permite concatenarea unui șir de caractere la sfârșitul unui alt șir de caractere. Una dintre ele are prototipul:

```
char *streat(char *dest,const char *sursa);
```

Această funcție copiază șirul de caractere din zona spre care pointează *sursa*, în zona de memorie care urmează imediat după ultimul caracter propriu al șirului spre care pointează *dest*. Se presupune că zona spre care pointează *dest* este suficientă pentru a păstra caracterele proprii celor două șiruri care se concatenează, plus caracterul *NUL* care termină șirul rezultat în urma concatenării.

Funcția returnează valoarea lui *dest*.

Exemplu:

```
char tab1[100] = "Limbajul C++";
char tab2[] = "este c incrementat";
strcat(tab1, " "); /* concatenează caracterul spatiu după cel de al doilea caracter
+ pastrat în tabloul tab1 la initializare */
strcat(tab1,tab2); /* concatenează textul pastrat în tab2 la initializare, după
spatiul concatenat prin instrucțiunea precedenta */
```

Observație:

Funcția *strcat*, la fel ca funcția *strcpy*, nu trebuie să modifice șirul de caractere spre care pointează *sursa*.

O altă funcție de bibliotecă utilizată la concatenarea de șiruri este funcția

strncat.

Ea are prototipul:

```
char *strncat(char *dest,const char *sursa,unsigned n);
```

În acest caz se concatenează, la sfîrșitul șirului spre care pointează *dest*, cel mult *n* caractere ale șirului spre care pointează *sursa*.

Dacă *n* > lungimea șirului spre care pointează *sursa*, atunci se concatenează intregul șir, altfel numai primele *n* caractere ale acestuia.

Exemplu:

```
char tab1[100] = "Limbajul E este mai bun decit ";
char tab2[] = "limbagul C++ care este un superset a lui C";
strncat(tab1,tab2,12);
```

După revenirea din funcție, tabloul *tab1* conține succesiunea de caractere:
Limbajul E este mai bun decit limbagul C++

8.6.4. Compararea șirurilor de caractere

Șirurile de caractere se pot compara folosind codurile ASCII ale caracterelor din compunerea lor.

Fie *s1* și *s2* două tablouri unidimensionale de tip caracter folosite pentru a păstra, fiecare, cîte un șir de caractere.

Șirurile păstrate în aceste tablouri sunt *egale* dacă au lungimi egale și *s1[i]=s2[i]* pentru toate valorile lui *i*.

Șirul păstrat în tabloul *s1* este *mai mic* decît cel păstrat în *s2*, dacă există un indice *i*, așa incit:

s1[i] < s2[i]

și

s1[i] = s2[j] pentru *j=0,1,...,i-1*.

Șirul păstrat în tabloul *s1* este *mai mare* decît cel păstrat în *s2*, dacă există un indice *i*, așa incit:

s1[i] > s2[i]

și

s1[j] = s2[j] pentru *j=0,1,...,i-1*.

Compararea șirurilor de caractere se poate realiza folosind funcții standard de felul celor de mai jos.

O funcție utilizată frecvent în compararea șirurilor este funcția de prototip:

```
int strcmp(const char *s1,const char *s2);
```

Notăm cu *sir1* șirul de caractere spre care pointează *s1* și cu *sir2* șirul de caractere spre care pointează *s2* (spunem că un pointer pointează spre un șir dacă

valoarea lui este adresa de început a zonei de memorie în care se păstrează șirul respectiv).

Funcția *strcmp* returnează:

- o valoare negativă dacă *sir1 < sir2*;
- zero dacă *sir1 = sir2*;
- o valoare pozitivă dacă *sir1 > sir2*.

O altă funcție pentru compararea a două șiruri este funcția de prototip:

```
int stricmp(const char *s1,const char *s2,unsigned n);
```

Această funcție compară cele două șiruri spre care pointează *s1* și *s2* utilizând cel mult primele *n* caractere din fiecare șir. În cazul în care minimul dintre lungimile celor două șiruri este mai mic decît *n*, funcția *strncmp* realizează aceeași comparație ca și funcția *strcmp*.

Adesea, la compararea șirurilor de caractere dorim să nu se facă distincție între literele mici și mari. Acest lucru este posibil dacă se folosește funcția de prototip:

```
int stricmp(const char *s1,const char *s2);
```

Această funcție returnează aceleași valori ca și funcția *strcmp*, cu deosebirea că la compararea literelor nu se face distincție între literele mari și mici.

Exemplu:

```
char *sir1 = "ABC";
char *sir2 = "abc";
int i;
```

Apelul:

i = strcmp(sir1,sir2);

returnează o valoare negativă, deoarece literele mari au coduri ASCII mai mici decît literele mici (*A* are codul 65, iar *a* are codul 97), deci

"ABC" < "abc"

Apelul:

i = stricmp(sir1,sir2);

returnează zero, deoarece ignorindu-se diferența dintre literele mari și mici, cele două șiruri devin egale.

Pentru a limita compararea a două șiruri de caractere la primele cel mult *n* caractere ale lor, la comparare ignorindu-se diferența dintre literele mici și mari, se va folosi funcția de prototip:

```
int strincmp(const char *s1, const char *s2, unsigned n);
```

Exerciții:

- 8.11 Să se scrie un program care citește o succesiune de cuvinte și-l afișează pe cel mai lung dintre ele.

Prin *cuvint* înțelegem o succesiune de caractere diferite de caracterele albe.

Vom presupune că un cuvint nu are mai mult de 100 de caractere. Cuvintele sunt separate prin caractere albe. La sfîrșit se tastează sfîrșitul de fișier pentru a termina succesiunea de cuvinte.

PROGRAMUL BVIII11

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

#define MAX 100

main() /* citește o succesiune de cuvinte și-l afișează pe cel mai lung dintre ele */
{
    int max=0,i;
    char cuvint[MAX+1];
    char cuvint_max[MAX+1];

    while(scanf("%100s",cuvint) != EOF)
        if(max < (i=strlen(cuvint))) {
            max = i;
            strcpy(cuvint_max,cuvint);
        }
    if(max) printf("%s\n", cuvint_max);
}
```

Observații:

1. Cuvintul citit se păstrează în tabloul *cuvint* de MAX+1 elemente. La citire se utilizează specificatorul de format:

%100s

Care permite să se citească cel mult 100 de caractere diferite de cele albe. Funcția *scanf* păstrează caracterul NUL după ultimul caracter citit. Deci, un cuvint de 100 de caractere ocupă 101 octeți.

2. După citirea unui cuvint se apeleză funcția *strlen* pentru a determina numărul caracterelor citite prin *scanf* și păstrate în tabloul *cuvint*.

În acest caz s-a utilizat expresia:

i = strlen(cuvint)

Apoi se compară lungimea cuvintului citit cu *max*.

Variabila *max* are ca valoare lungimea maximă a cuvintelor citite înaintea celui curent.

Initial *max* = 0, deoarece nu există nici un cuvint citit.

Dacă *max* este mai mic decât lungimea cuvintului citit curent, atunci lui *max* îi se atribuie această valoare, iar cuvintul citit este transferat în zona de memorie alocată tabloului *cuvint_max*. În acest scop se apelează funcția *strcpy*:

```
strcpy(cuvint_max,cuvint);
```

- 8.12 Să se scrie un program care citește două cuvinte și le afișează în ordine crescătoare.

Cuvintul se definește ca în exercițiul precedent, adică este o succesiune de cel mult 100 de caractere care nu sunt albe. Cuvintele sunt separate prin caractere albe.

PROGRAMUL BVIII12

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

#define MAX 100

main() /* citește două cuvinte și le afișează în ordine crescătoare */
{
    char cuv1[MAX+1];
    char cuv2[MAX+1];

    if(strcmp(cuv1,cuv2) != 1) {
        printf("nu s-a tastat un cuvint\n");
        exit(1);
    }
    if(strcmp(cuv1,cuv2) != -1) {
        printf("nu s-a tastat un cuvint\n");
        exit(1);
    }
    if(strcmp(cuv1,cuv2) < 0) {
        /* primul cuvint tastat este mai mic decât cel de-al doilea */
        printf("%s\n",cuv1);
        printf("%s\n",cuv2);
    }
    else {
        printf("%s\n",cuv2);
        printf("%s\n",cuv1);
    }
}
```

- 8.13 Să se scrie un program care citește o succesiune de cuvinte și-l afișează pe cel mai mare.

Prin cuvint înțelegem o succesiune de cel mult 100 de caractere care nu sunt albe. Cuvintele sunt separate prin caractere albe. Succesiunea de cuvinte se termină cu sfîrșitul de fișier.

PROGRAMUL BVIII13

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

#define MAX 100

main() /* citeste o succesiune de cuvinte si-l afiseaza pe cel mai mare */
{
    char cuvcrt[MAX+1];
    char cuvmax[MAX+1];

    cuvmax[0] = '\0'; /* cuvmax se initializeaza cu cuvintul vid */
    while(scanf("%100s", cuvcrt) != EOF)
        if(strcmp(cuvcrt, cuvmax) > 0)
            /* cuvintul curent este mai mare decit cel păstrat in cuvmax */
            strcpy(cuvmax, cuvcrt);
    printf("cel mai mare cuvint este\n");
    printf("%s\n", cuvmax);
}
```

8.14 Să se scrie un program care citește o succesiune de cuvinte, le sortează în ordine crescătoare și apoi le afișează în ordinea respectivă.

Prin *cuvint* înțelegem un șir de litere mici sau mari. La compararea cuvintelor nu se face deosebirea între literele mici și mari. În felul acesta, cuvintele se vor afișa în ordine alfabetică. Vom presupune că lungimea unui cuvint nu depășește 30 de caractere și că sunt cel mult 500 de cuvinte la intrare.

În prima parte se citesc cuvintele și se păstrează în tabloul *tcuvinte*. Acesta este de tip *char*. Fiecare cuvint se termină prin caracterul *NUL*.

La întărirea sfîrșitului de fișier se trece la sortarea cuvintelor păstrate în tabloul *tcuvinte*.

Sortarea cuvintelor se face folosind metoda bulelor (vezi exercițiul 4.40).

Conform acestei metode, se parcurg elementele tabloului comparindu-se de fiecare dată două elemente vecine. Dacă ele nu sunt în ordinea cerută (crescătoare), atunci ele se permutează. Permutarea elementelor invecinate este simplă în cazul în care elementele respective sunt numere. În cazul de față elementele sunt cuvinte și permutarea lor este mai complicată. De aceea, vom folosi pointeri spre începutul fiecărui cuvint. Acești pointeri îi păstrăm ca elemente ale unui tablou pe care îl numim *tpointer*.

La terminarea citirii cuvintelor, elementele lui *tpointer* vor pointa spre începuturile cuvintelor păstrate în tabloul *tcuvinte*. Astfel, *tpointer[0]* pointează spre primul cuvint citit și păstrat în *tcuvinte* (are ca valoare adresa de început a zonei de memorie în care se păstrează primul caracter), *tpointer[1]* pointează spre al doilea cuvint și aşa mai departe.

Exemplu:

Presupunem că se tastează textul:

Programarea orientata spre obiecte este un stil modern de programare.

După citirea textului respectiv, tabelele *tpointer* și *tcuvinte* au următoarele conținute:

tpointer[0] = adresa lui *tcuvinte[0]*;

Elementele *tcuvinte[0]* -*tcuvinte[11]* conțin caracterele cuvintului *Programarea*, inclusiv caracterul *NUL* de la sfîrșitul cuvintului.

tpointer[1] = adresa lui *tcuvinte[12]*;

Elementele *tcuvinte[12]* -*tcuvinte[21]* conțin caracterele corespunzătoare cuvintului *orientata*.

tpointer[2] = adresa lui *tcuvinte[22]*;

Elementele *tcuvinte[22]* -*tcuvinte[26]* conțin caracterele corespunzătoare cuvintului *spre* și aşa mai departe.

Ultimul element al tabloului *tpointer* la care i s-a atribuit valoare este:

tpointer[9] = adresa lui *tcuvinte[58]*

Elementele *tcuvinte[58]* -*tcuvinte[68]* conțin caracterele corespunzătoare cuvintului *programare*.

Pentru a compara două cuvinte invecinate păstrate în tabloul *tcuvinte*, utilizăm pointeri spre ele, pointeri care sunt memorati în tabloul *tpointer*.

Astfel, pentru a compara cuvintul al *k*-lea din tabloul *tcuvinte*, cu următorul lui, vom apela funcția *strcmp* cu parametri *tpointer[k]* și *tpointer[k+1]*:

```
strcmp(tpointer[k], tpointer[k+1])
```

Dacă la un astfel de apel funcția *strcmp* returnează o valoare pozitivă, înseamnă că *tpointer[k]* pointează spre un cuvint care este mai mare (urmărează în ordine alfabetică) decât cuvintul spre care pointează *tpointer[k+1]*. În acest caz ar urma să se permute cuvintele respective, deoarece nu sunt în ordinea cerută.

Existența pointerilor spre cuvintele respective face ca să nu mai fie necesară permutea efectivă a cuvintelor, ci numai a valorilor pointerilor spre ele. Deci, în acest caz se vor permuta valorile pointerilor *tpointer[k]* și *tpointer[k+1]*:

```
if(strcmp(tpointer[k], tpointer[k+1]) > 0)
{
    char *t;
    t = tpointer[k];
    tpointer[k] = tpointer[k+1];
    tpointer[k+1] = t;
}
```

Se observă utilizarea funcției *strcmp*, pentru a realiza comparații între cuvinte la care se ignoră diferența dintre literele mici și mari.

PROGRAMUL BVIII14

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

#define MAXLC 30
#define MAXNC 500

main() /* sorteaza un sir de cuvinte in ordine crescatoare (alfabetica) */
{
    static char tcuvinte[(MAXLC+1)*MAXNC];
    static char *tpointer[MAXNC];
    int c,i,j,k,ind;
    char *t;

    i = 0; /* indice in tpointer */
    j = 0; /* indice in tcuvinte */
    c = getchar();

    /* citirea cuvintelor */
    while(c != EOF) { /* 1 */
        /* avans peste caracterele care nu sunt litere */
        while( !(c >= 'A' && c <= 'Z' || c >= 'a' && c <= 'z')) {
            /* c nu contine o litera */
            c = getchar();
            if( c == EOF ) break;
        }

        if( c == EOF ) break;

        /* c contine o litera; este litera de inceput a cuvantului curent; acest cuvant se pastreaza
           incepind cu elementul tcuvinte[j]; adresa de inceput a cuvantului, adica adresa
           elementului tcuvinte[j], se pastreaza in tabloul tpointer; aceasta adresa este
           tcuvinte + j si se atribuie elementului tpointer[i] */
        tpointer[i++] = tcuvinte + j;

        /* se citesc caracterele cuvantului curent si se pastreaza in tabloul
           tcuvinte incepind cu elementul tcuvinte[j] */
        while(c >= 'A' && c <= 'Z' || c >= 'a' && c <= 'z') {
            tcuvinte[j++] = c;
            c = getchar();
        }

        /* se pastreaza caracterul NUL, dupa cel curent */
        tcuvinte[j++] = '\0';
    } /* sfarsit while 1 */

    /* s-a terminat citirea cuvintelor */
    /* s-au citit i cuvinte */

    if(i) { /* 1 */
        /* exista cel putin un cuvant citit */
        /* se face sortare */
    }
}
```

```
ind = 1;
while( ind ) {
    ind = 0;
    for( k=0; k < i-1; k++ )

        /* se analizeaza ordinea cuvintelor vecine */
        if(strcmp(tpointer[k], tpointer[k+1]) > 0) {
            /* permutarea pointerilor */
            t = tpointer[k];
            tpointer[k] = tpointer[k+1];
            tpointer[k+1] = t;
            ind = 1;
        } /* sfarsit if */
} /* sfarsit while */

/* s-a terminat sortarea; se afiseaza cuvintele */
for(j = 0; j < i; j++ ) {
    printf("%s\n", tpointer[j]);
    if((j+1)%23 == 0) {
        printf("actionati o tasta pentru a continua\n");
        getch();
    }
} /* sfarsit for */
} /* sfarsit if 1 */
}
```

8.7. Expresie lvalue

În paragraful 3.2.7, s-a definit expresia de atribuire prin următorul format:

(1) $v = \text{expresie}$

unde:

v

- Se consideră că este o variabilă simplă, o variabilă cu indici (permite accesul sau modificarea valorii unui element de tablou) sau definește un element de structură.

Definiția expresiei de atribuire poate fi completată în momentul de față adăugind formatul:

(2) $*ep = \text{expresie}$

unde:

ep

- Este o expresie care definește un pointer nenul și care nu este un pointer spre o zonă constantă. Astfel de expresii s-au utilizat deja în paragrafele precedente.

Exemplu:

```
int n;
double x;
void *p;
```

Expresiile de atribuire de forma:

$n = 123$, $x = 3.14159$

corespund formatului (1), indicat mai sus pentru expresiile de atribuire.

Fie:

$p = \&n;$

atunci am văzut că atribuirea

$\ast(\text{int } *)p = 123$

realizează același lucru ca și expresia:

$n = 123.$

Expresia:

$(\text{int } *)p$

defineste un pointer spre o zonă care nu este constantă și deci atribuirea de mai sus corespunde formatului (2) al expresiilor de atribuire. În mod analog, dacă:

$p = \&x;$

atunci atribuirea:

$\ast(\text{double } *)p = 3.14159$

realizează același lucru ca și expresia:

$x = 3.14159$

Expresiile utilizabile în partea stîngă a unei expresii de atribuire se numesc expresii *lvalue*.

Litera *l* provine de la cuvintul englezesc *left*.

Noțiunea de expresie *lvalue* aparține autorilor limbajului C (vezi [2]).

Ulterior s-a introdus și noțiunea de expresie *rvalue*, care este o expresie care se poate utiliza în partea dreaptă a unei expresii de atribuire, dar nu și în partea stîngă (vezi [8]).

Din cele de mai sus rezultă că o expresie *lvalue* poate fi:

- un nume de variabilă simplă;
- o variabilă cu indici;
- o construcție ce permite accesul sau modificarea valorii unui element de structură;
- o construcție de forma:

$\ast e p$

unde:

$e p$ - Este o expresie care definește un pointer nenufără spre o dată care nu este o constantă.

Fie declarațiile:

```
const int *a = 100;  
int b;
```

În acest caz, *a* este un pointer spre o zonă constantă și de aceea, expresia:

$\ast a = 123$

nu este corectă. În acest caz $\ast a$ nu este o expresie *lvalue*. Ea este o expresie *rvalue*, deoarece se poate utiliza în partea dreaptă a expresiilor de atribuire.

Într-adevăr, expresia:

$b = \ast a$

este corectă și atribuie variabilei *b* valoarea 100.

8.8. Alocarea dinamică a memoriei

Limbajul C permite utilizatorului să aloce date atât pe stivă (date automatice), cât și în zone de memorie care nu aparțin stivei (date globale sau statice).

Alocarea datelor pe stivă se face la execuție și ea nu este permanentă. Astfel, dacă declarația:

tip nume;

se utilizează în corpul unei funcții, atunci variabila *nume* se aloca pe stivă la fiecare apel al funcției respective. La revenirea din funcție, stiva se "curată" (se reduce la starea ayuntă înaintea apelelului) și prin aceasta variabila *nume* nu mai este alocată (devine nedefinită). O alocare de acest fel a memoriei se spune că este *dinamică*.

Pentru datele globale sau statice, memoria este alocată în fazele precedente execuției și alocarea rămâne valabilă pînă la terminarea execuției programului. De aceea, pentru datele de acest fel se spune că alocarea este *statică* (nu este dinamică).

Limbajele C și C++ oferă utilizatorului posibilitatea de a aloca dinamic memorie și în alt mod decît cel indicat mai sus pentru datele automatice. Aceasta se realizează într-o zonă de memorie specială, distinctă de stivă. Această zonă de memorie se numește *memorie heap* (memorie grămadă, mormăne, de acumulare etc.). Ea poate fi gestionată prin funcții standard în ambele limbaje.

Mai jos indicăm cîteva funcții care pot fi utilizate la alocarea dinamică a datelor în memoria *heap*. Aceste funcții sunt comune ambelor limbaje. Mai tîrziu o să vedem că în limbajul C++ există chiar operatori pentru alocări dinamice în memoria *heap*.

Funcțiile standard pentru gestiunea memoriei *help* au prototipurile în fișierul *alloc.h*.

*Alocarea unei zone de memorie în memoria *heap* se poate realiza cu ajutorul mai multor funcții. O funcție utilizată frecvent este funcția *malloc*. Aceasta are prototipul:*

```
void *malloc(unsigned n);
```

Funcția aloca în memoria *heap* o zonă contigua de *n* octeți. Ea returnează adresa de început a zonei alocate. Aceasta adresa reprezintă un pointer de tip *void* (*void **). Prin intermediul acestuia se pot păstra date în zona de memorie alocată în acest fel. Pentru a păstra o dată de un *tip* dat într-o zonă de memorie alocată prin *malloc* este necesar să convertim adresa returnată de funcție spre tipul datei respective.

Exemplu:

Se cere să se aloce în memoria *heap* o zonă de memorie pentru a păstra *n* valori de tip *int*. În acest scop declarăm un pointer spre tipul *int*:

```
int *p;
```

Apoi, apelăm funcția *malloc* cu ajutorul expresiei de atribuire:

```
p = (int *)malloc(n*sizeof(int))
```

Se observă că valoarea returnată de funcția *malloc* a fost convertită spre tipul *int **, adică pointer spre *int*.

În continuare putem păstra și utiliza date de tip *int*, folosind pointerul *p*. De exemplu, expresia:

```
*p = 123
```

păstrează întregul 123 în primii doi octeți ai zonei alocate prin expresia de mai sus.

Expresia:

```
y = *p
```

atribuie lui *y* valoarea păstrată mai sus în memoria *heap*.

În general, expresia:

```
*(p+i) = k
```

atribuie valoarea lui *k* în zonă de memorie de adresa definită prin expresia pointer:

```
p+i
```

Același expresie se poate utiliza pentru a face acces la datea respectivă.

Observații:

1. Funcția *malloc* are ca parametru un întreg fără semn, adică acesta trebuie să aparțină intervalului [0,65535].
2. În cazul în care în memoria *heap* nu se poate aloca o zonă de memorie contigua de atâtia octeți cît este valoarea parametrului de la apel, se va returna *pointerul nul*, adică valoarea zero.

De aceea, după apelul funcției *malloc* vom testa valoarea returnată pentru a ne

asigura că aceasta nu este zero.

Zonile alocate prin funcția *malloc* pot fi *eliberate*, pentru a putea fi eventual realocate, folosind funcția standard *free*. Aceasta are prototipul:

```
void free(void *p);
```

Prin apelul ei, se eliberează zona de memorie din memoria *heap*, spre care pointează *p*. Menționăm că valoarea lui *p* trebuie să fie obținută printr-un apel al unei funcții standard de alocare, cum este de exemplu funcția *malloc*.

Se recomandă ca această funcție să fie apelată de îndată ce datele dintr-o zonă de memorie *heap* nu mai sunt necesare.

În felul acesta zona respectivă poate fi ulterior realocată. De asemenea, eliberarea sistematică a zonelor din memoria *heap* poate preveni fărîmîțarea excesivă a memoriei *heap*.

O altă funcție standard utilă pentru a aloca zone de memorie în memoria *heap* este funcția *calloc*. Ea are prototipul:

```
void *calloc(unsigned nrelem,unsigned dimelem);
```

Funcția aloca o zonă de memorie de *nrelem * dimelem* octeți.

Că și funcția *malloc*, funcția *calloc* returnează adresa de început a zonei de memorie alocată, adresa care reprezintă un pointer spre *void*.

În cazul în care nu se poate aloca *nrelem * dimelem* octeți, funcția returnează valoarea zero.

Elementele zonei de memorie alocată prin *calloc* sunt inițializate cu valoarea zero.

Zona de memorie alocată cu ajutorul funcției *calloc* se eliberează folosind funcția *free* indicată mai sus.

Pentru programe relativ mici, pointerii se păstrează pe 16 biți. Aceasta înseamnă că ei pot păstra adrese de pînă la 64k. Despre un astfel de pointer se spune că este de tip *near*.

Programele mai complexe, care necesită adrese de peste 64k, utilizează pointeri alocati pe 32 de biți. Despre un astfel de pointer se spune că este de tip *far*.

Biblioteca standard conține funcții pentru a aloca în memoria *heap* zone de memorie a căror adresa se reprezintă în memoria *heap* pe mai mult de 16 biți. De asemenea, dimensiunea unei astfel de zone de memorie poate depăși 64k. Astfel, pentru a aloca în memoria *heap* zone de memorie de adrese mai mari ca 64k, putem folosi funcțiile *farmalloc* și *farcalloc*, funcții analoge cu funcțiile *malloc* și respectiv *calloc*, indicate mai sus.

Funcția *farmalloc* are prototipul:

```
void far *farmalloc(unsigned long n);
```

Ea se utilizează la fel ca și funcția *malloc*. În acest caz funcția returnează un pointer de tip *far* (se alocă pe 32 de biți). De asemenea, la apelul funcției *farmalloc*, parametrul *n* poate depăși 64k = 65536.

Zona alocată cu ajutorul funcției *farmalloc* se eliberează apelind funcția *farfree*, care este similară cu funcția *free*. Ea are prototipul:

```
void farfree(void far *p);
```

Exerciții:

8.15 Să se scrie o funcție care păstrează un sir de caractere într-o zonă de memorie alocată în memoria *heap*.

Funcția returnează adresa de început a zonei în care se păstrează sirul de caractere sau zero (pointerul nul), în cazul în care nu se poate rezerva zona respectivă în memoria *heap*.

FUNCTIA BVIII15

```
char *memsir(char *s)
/* păstrează în memoria heap sirul de caractere spre care pointează s */
{
    char *p;

    if((p = (char *)malloc(strlen(s) + 1)) != 0) {
        /* p are ca valoare adresa de început a zonei de memorie rezervată în
           memoria heap și care are strlen(s) + 1 octeți */
        strcpy(p, s); /* copiază sirul spre care pointează s, în zona spre care pointează p */
        return p;
    }
    else
        return 0; /* pointerul nul; nu s-a putut aloca zona necesară
                  pentru a păstra sirul spre care pointează s */
}
```

Observații:

1. Funcția de față returnează un pointer spre caractere, deci *tip* din antetul ei este:

```
char *
```

2. Condiția din paranteza lui *if* se poate scrie și fără partea

```
!= 0
```

adică

```
if(p=(char *)malloc(strlen(s)+1))
```

Apelul:

```
strlen(s)
```

determină numărul de octeți necesari pentru a păstra caracterele proprii ale sirului spre care pointează *s*.

Numărul returnat de funcția *strlen* se mărește cu 1 pentru a aloca un octet pentru caracterul *NUL* care trebuie să termine orice sir de caractere.

3. Apelul:

```
strcpy(p,s);
```

copiază caracterele sirului, inclusiv caracterul *NUL*, spre care pointează *s*, în zona a cărei adresa de început este valoarea lui *p*.

8.16 Să se scrie un program care citește o succesiune de cuvinte și-l afișează pe cel mai mare.

Un astfel de program este descris în exercițiul 8.13. În programul respectiv, cuvintul maxim se păstrează în tabloul *cuvmax* alocat pe stivă. În programul de față se va păstra cuvântul maxim în memoria *heap*.

PROGRAMUL BVIII16

```
#include <stdio.h>
#include <string.h>
#include <alloc.h>
#include "bviii15.cpp"

#define MAX 100

main() /* citește o succesiune de cuvinte și-l afișează pe cel mai mare */
{
    char cuv crt[MAX+1];
    char *cuvmax = 0; /* pointează spre cuvântul maxim; initial
                        are ca valoare pointerul nul */

    while(scanf("%100s", cuv crt) != EOF)
        if(cuvmax==0) /* prima citire */
            cuvmax = memsir(cuv crt);
        else
            if(strcmp(cuv crt, cuvmax) > 0) {
                /* cuvântul citit este mai mare decât cel din memoria heap și spre care
                   pointează cuvmax */
                /* se eliberează zona din memoria heap în care se păstrează cuvântul
                   maxim înaintat la citirile anterioare */
                free(cuvmax);

                /* se aloca zona în memoria heap și se păstrează în ea cuvântul curent
                   citit */
                cuvmax = memsir(cuv crt);
            }
    printf("cel mai mare cuvânt este\n");
    printf("%s\n", cuvmax);
    free(cuvmax);
}
```

Observație:

În acest program nu s-a testat imposibilitatea păstrării sirului maxim în memoria *heap* deoarece acesta are cel mult 101 caractere. Oricănd memoria *heap* este alocată un spațiu mai mare.

8.17 Sa se scrie o funcție care sortează în ordine crescătoare n șiruri de caractere.

Funcția are doi parametri:

- $tpointer$
- Tablou unidimensional de tip pointer spre *char*.
 - Fiecare element al tabloului este un pointer spre unul din cele n șiruri de caractere.
 - $tpointer[i]$ este pointer spre al $i+1$ -lea șir de caractere.
- n
- Întreg de tip *int* care are ca valoare numărul șirurilor de caractere care se sortează.

Funcția utilizează metoda bulelor. Pentru a ordona șirurile în ordine crescătoare se permute pointerii în locul șirurilor, ca în exercițiul 8.14.

La compararea șirurilor de caractere se ignoră diferența dintre literele mici și mari.

FUNCȚIA BVIII17

```
void ordsircresc(char *tpointer[], int n)
/* sorteaza in ordine crescatoare cele n siruri de caractere spre care pointeaza elementele
   tpointer[0],tpointer[1],...,tpointer[n-1]*/
{
    int ind,i;
    char *t;

    ind = 1;
    while( ind ) {
        ind = 0;
        for(i=0; i< n-1; i++)
            if(strcmp(tpointer[i],tpointer[i+1]) > 0){
                t = tpointer[i];
                tpointer[i] = tpointer[i+1];
                tpointer[i+1] = t;
                ind = 1;
            }
    }
}
```

8.18 Să se scrie un program care citește o succesiune de cuvinte și le afișeză în ordine crescătoare. Se ignoră diferența dintre literele mici și mari.

Prin cuvint se înțelege o succesiune de caractere care nu sunt albe. Cuvintele sunt separate prin caractere albe. Succesiunea de cuvinte se termină cu sfîrșitul de fișier.

Cuvintele citite se păstrează în memoria *heap*. Programul utilizează un tablou de pointeri ale cărui elemente pointează fiecare spre cîte un cuvînt.

Se presupune că sunt cel mult 500 de cuvinte.

PROGRAMUL BVIII18

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

#include "bviii15.cpp"
#include "bviii17.cpp"

#define MAXCUV 500

main() /* citeste o succesiune de cuvinte si le afiseaza in ordine crescatoarc */
{
    char *tp[MAXCUV];
    int i,j;
    char t[255];

    /* se citesc cuvintele si se pastreaza in memoria heap */
    for(i=0; scanf("%s",t) == 1; i++)
        if((tp[i] = memsir(t)) == 0 ) {
            printf("memorie insuficienta\n");
            exit(1);
        }

    /* se sorteaza cuvintele in ordine crescatoarc */
    if( i ) ordsircresc(tp,i);

    /* listarea cuvintelor dupa sortare */
    for(j=0; j < i; j++) {
        printf("%s\n", tp[j] );
        if((j+1)%23 == 0 ) {
            printf("actionati o tasta pentru a continua\n");
            getch();
        }
    } /* sfirsit for listare */
}
```

Observație:

Acest program are un avantaj față de cel din exercițiul 8.14, deoarece în cazul de față se alocă pentru fiecare cuvînt o zonă de memorie de lungime egală cu numărul de caractere ale cuvintului respectiv, plus 1 (pentru caracterul *NUL*). În programul din exercițiul 8.14. se alocă o zonă fixă de memorie de $(30+1)*500$ octeti, care este de obicei prea mare, deoarece cuvintele au în medie mai puțin de 30 de caractere.

În general, alocarea dinamică în memoria *heap* permite alocări optime, față de cele făcute în mod obișnuit, folosind tablouri automatice, globale sau statice.

8.9. Utilizarea tablourilor de pointeri la prelucrări de date de tip șir de caractere

Tablourile de pointeri pot fi utilizate pentru a prelucra succesiuni de șiruri de caractere. Utilizările acestor tablouri s-au văzut deja în exercițiile din paragraful precedent (vezi 8.18. și 8.14.).

Într-un program se întâlnesc adesea șiruri de caractere care se tratează într-un mod unitar. În acest caz este util să se definiască un taboulu de pointeri, fiecare element al său fiind un pointer spre un astfel de șir de caractere.

Astfel de tablouri pot fi utilizate pentru a păstra sau a avea acces la diferite denumiri, cuvinte cheie, mesaje de eroare etc.

În [2] se definesc astfel de tablouri pentru a păstra și utiliza denumirile lunilor calendaristice și cuvintele cheie ale limbajului C.

De exemplu, pentru a păstra denumirile lunilor calendaristice se poate utiliza un taboulu de pointeri cu 13 elemente:

Numim *tpdl* acest taboulu. Elementul *tpdl[1]* este pointer spre șirul de caractere "ianuarie", *tpdl[2]* este pointer spre șirul de caractere "februarie" etc.

Ultimul element, *tpdl[12]* este pointer spre șirul de caractere "decembrie".

Primul element al tabloului, nu pointează spre denumirea nici unei luni. De aceea, acest element poate avea ca valoare pointerul nul. O altă posibilitate este ca *tpdl[0]* să pointeze spre un text de eroare, de exemplu "luna ilegală".

Dacă *p* este un pointer spre un șir de caractere, atunci poate fi definit sau declarat ca pointer spre un șir de caractere printr-o construcție de forma:

```
char *p = sir;
```

Șirul de caractere *sir* se păstrează într-o zonă de memorie, iar lui *p* îi se atribuie adresa de început a zonei respective.

În mod analog se pot inițializa elementele tabloului *tpdl*, folosind o definiție sau o declarație de forma:

```
char *tpdl[] = {  
    "luna ilegală",  
    "ianuarie",  
    "februarie",  
    "martie",  
    "aprilie",  
    "mai",  
    "iunie",  
    "iulie",  
    "august",  
    "septembrie",  
    "octombrie",  
    "noiembrie",  
    "decembrie"  
};
```

Prin această construcție, elementul *tpdl[i]*, pentru $1 \leq i \leq 12$, este pointer spre

șirul de caractere care denumește luna calendaristică a i -a din an. De exemplu, apelul:

```
printf("%s\n", tpdl[1]);
```

afisează textul

ianuarie

Exerciții:

- 8.19 Să se scrie o funcție care are ca parametru un întreg *n* și returnează un pointer spre denumirea lunii calendaristice a n -a pentru $1 \leq n \leq 12$ sau spre textul "luna ilegală" pentru "*n*" în afara acestui interval.

Fie *denluna* numele acestei funcții. Ea returnează un pointer spre *char*, deci returnează o dată de tip *char **.

Deci antetul ei este:

```
char *denluna(int n)
```

FUNCȚIA BVIII19

```
char *denluna(int n)  
/* returneaza pointerul spre denumirea lunii */  
{  
    static char *tpdl[] = {  
        "luna ilegală",  
        "ianuarie",  
        "februarie",  
        "martie",  
        "aprilie",  
        "mai",  
        "iunie",  
        "iulie",  
        "august",  
        "septembrie",  
        "octombrie",  
        "noiembrie",  
        "decembrie"  
    };  
  
    return n < 1 || n > 12 ? tpdl[0] : tpdl[n];  
}
```

- 8.20 Să se scrie un program care citește o dată calendaristică scrisă sub formă:

zzllaaaa

o validează și o afișează sub formă:

zz luna aaaa

unde:

zz - Reprezintă ziua pe două cifre.

ll - Reprezintă numărul lunii pe două cifre.

- Reprezintă anul pe 4 cifre.
- Reprezintă denumirea lunii calendaristice.

Data citită se validează utilizând funcția *v_calend* definită în exercițiul 6.5.

PROGRAMUL BVIII20

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

#include "bvi5.cpp"
#include "bvi119.cpp"

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

main() /* - citește o dată calendaristică sub forma:
           zzllaaa
           - o validează și o afișează sub forma:
             zz luna anaa */
{
    int zz,ll,aaaa;

    if(scanf("%2d %2d %4d", &zz, &ll, &aaaa) != 3 ) {
        printf("nu s-a tastat un întreg de 8 cifre\n");
        exit(1);
    }
    if(v_calend(zz,ll,aaaa) == 0 ){
        printf("data calendaristica eronata\n");
        exit(1);
    }
    printf("%02d %s %4d\n", zz, denluna(ll), aaaa);
}

```

8.21 Să se scrie o funcție care din denumirea lunii calendaristice determină numărul ei.

Această funcție este inversa funcției *denluna* definită în exercițiul 8.19.

Funcția are ca parametru un pointer spre denumirea lunii calendaristice. Folosind acest pointer, denumirea lunii se caută cu ajutorul funcției *strcmp* pentru a stabili coincidența ei cu unul din sirurile de caractere spre care pointează elementele tabloului *tpdl* (vezi exercițiul 9.19.). În cazul în care denumirea este eronată funcția returnează valoarea zero.

FUNCȚIA BVIII21

```
int nrluna(char *denl)
/* returneaza numarul lunii din denumirea ci */
{
    static char *tpdl[] = {
        "", "ianuarie",
        "februarie",
        "martie",
        "aprilie",
        "mai",
        "iunie",
        "iulie",
        "august",
        "septembrie",
        "octombrie",
        "noiembrie",
        "decembrie"
    };
    int i;

    for( i=1; i <= 12; i++)
        if(strcmp(denl,tpdl[i]) == 0 )
            return i;
    return 0;
}
```

```
"februarie",
"martie",
"aprilie",
"mai",
"iunie",
"iulie",
"august",
"septembrie",
"octombrie",
"noiembrie",
"decembrie"
};

int i;

for( i=1; i <= 12; i++)
    if(strcmp(denl,tpdl[i]) == 0 )
        return i;
return 0;
}
```

8.22 Să se scrie un program care citește o dată calendaristică tastată sub forma:

zz luna aaaa
 și o afișează sub forma:
 zz/l/l/aaaa
 dacă este validă.

În aceste formate s-au utilizat notațiile:

- | | |
|------|-----------------------------------|
| zz | - Ziua pe 1 - 2 cifre. |
| ll | - Luna pe 1 - 2 cifre. |
| aaaa | - Anul pe 4 cifre. |
| luna | - Denumirea lunii calendaristice. |

PROGRAMUL BVIII22

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "bvi5.cpp"
#include "bvi119.cpp"

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

main() /*- citește o data calendaristica sub forma:
           zz luna aaaa
           - o validează și o afișează sub forma:
             zz/l/l/aaaa */
{
    int zz,ll,aaaa;
    char dl[11]; /* pastreaza denumirea lunii calendaristice */

    if(scanf("%2d %10s %4d", &zz, dl,&aaaa) != 3 ) {

```

```

printf("data calendaristica eronata\n");
exit(1);

/* determina numarul lunii calendaristice */
if((l1 = nrLuna(dl)) == 0) {
    printf("luna eronata\n");
    exit(1);
}
if(v_calend(zz, l1, aaaa) == 0){
    printf("data calendaristica eronata\n");
    exit(1);
}
printf("%d/%d/%d\n", zz, l1, aaaa);
}

```

8.10. Tratarea parametrilor din linia de comandă

În linia de comandă folosită la apelul execuției unui program se pot utiliza diferiți parametri (argumente).

În cazul utilizării mediului integrat de dezvoltare Turbo C, acești parametri se definesc utilizând submeniul *Arguments* al meniului *Options*. Dacă se utilizează mediul integrat de dezvoltare Borland C++, atunci submeniul *Arguments* se selectează din meniul *Run*.

În toate cazurile argumentele sunt succesiuni de caractere separate prin spații. Ele se tastează în linia de comandă prin care se activează imaginea executabilă (fișierul cu extensia .exe) a programului.

Dacă se utilizează mediile integrate de dezvoltare Turbo C și Borland C++, atunci se procedează astfel:

mediul Turbo C

Se selectează meniu

Option:
<Alt>-O

Se selectează submeniu:

Arguments:
A

mediul Borland C++

Se selectează meniu

Run:
<Alt>-R

Se selectează submeniu:

Arguments:
A

După selectarea submeniului *Arguments* se afișează o fereastră în care se pot taste argumentele separate prin spațiu. După ultimul argument se actionează tasta Enter. Apoi se poate continua cu lansarea automată a fazelor: compilare, link-editare și execuție. Se tastează:

<Ctrl>-F9

Pentru a utiliza aceste argumente se vor folosi parametrii *argc* și *argv* ai funcției principale. În acest caz funcția principală are antetul

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

Parametrul *argc* are ca valoare numărul argumentelor din linia de comandă mărit cu unu. Parametrul *argv* este un tablou de pointeri spre zonele în care s-au păstrat argumentele definite ca mai sus.

Argumentele se păstrează sub formă de siruri de caractere. Pointerul *argv[0]* este întotdeauna pointerul spre calea fișierului cu imaginea executabilă a programului. În continuare *argv[1]* este pointerul spre primul argument tastat, *argv[2]* este pointerul spre al doilea argument și aşa mai departe. În general *argv[i]* este pointer spre al *i*-lea argument.

Exemplu:

Presupunem că la lansarea programului s-au furnizat argumentele:

1 OCTOMBRIE 1993

În cazul acesta parametrii funcției principale au valorile:

- | | |
|-----------------|---|
| <i>argc</i> = 4 | - (3 argumente plus specificatorul fișierului cu imaginea executabilă a programului). |
| <i>argv[0]</i> | - Pointer spre specificatorul fișierului cu imaginea executabilă a programului (cale, numele și extensia .EXE). |
| <i>argv[1]</i> | - Pointer spre sirul "1". |
| <i>argv[2]</i> | - Pointer spre sirul "OCTOMBRIE". |
| <i>argv[3]</i> | - Pointer spre sirul "1993". |

Exerciții:

8.23 Să se scrie un program care afișează parametri din linia de comandă.

PROGRAMUL BVIII23

```

#include <stdio.h>
main( int argc, char *argv[] )
/* afiseaza parametri din linia de comanda */
{
    int i;
    for( i=0; i < argc; i++)
        printf("%s\n", argv[i]);
}

```

8.24 Să se scrie un program care afișează data calendaristică preluată din linia de comandă sub forma:

zz//aaa

unde:

zz

- Ziua pe 1 - 2 cifre.

//

- Numărul lunii calendaristice pe 1 - 2 cifre.

Programul apelează funcția `v_calend` pentru a valida data calendaristică preluată din linia de comandă. Întrucât ea se tastează sub forma indicată mai sus, acesta reprezintă un singur argument.

`argv[1]` este pointerul spre zona în care se păstrează data calendaristică respectivă.

PROGRAMUL BVIII24

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

#include "bvi5.cpp"

int mizi[12] = { 0, 31, 28, 31, 10, 31, 10, 31, 31, 30, 31, 30, 31 };

main(int argc, char *argv[])
/* valideaza si afisaza data calendaristica preluata din linia de comanda */
{
    int zz, l1, aaaa;
    char a, b;

    if(argc < 2) {
        printf("lipseste data calendaristica in linia \
               de comanda\n");
        exit(1);
    }
    if(argc > 2) {
        printf("mai mult de un argument in linia de comanda\n");
        exit(1);
    }
    if(sscanf(argv[1], "%2d %c %2d %c %4d", &zz, &a,
              &l1, &b, &aaaa) != 5) {
        printf("argumentul nu are formatul zz/l1/aaaa\n");
        exit(1);
    }
    if( a != '/' ) {
        printf("dupa zi nu urmeaza caracterul /\n");
        exit(1);
    }
    if(b != '/') {
        printf("dupa luna nu urmeaza caracterul /\n");
        exit(1);
    }
    if(v_calend(zz, l1, aaaa) == 0) {
        printf("data calendaristica eronata\n");
        exit(1);
    }
    printf("%d/%d/%d\n", zz, l1, aaaa);
}
```

8.11. Pointeri spre funcții

Numele unei funcții este un pointer spre funcția respectiva. De aceea, numele unei funcții poate fi folosit ca parametru efectiv la apelul unei funcții. În felul acesta, o funcție poate transfera funcției apelate un pointer spre o funcție. Funcția apelată, la rândul ei, poate apela funcția care i-a fost transferată în acest fel.

Fie, de exemplu, `f` o funcție care are ca parametru o altă funcție. Apelul:

`f(g);`

unde `g` este numele unei funcții, transferă funcției `f` pointerul spre funcția `g`.

Înainte de apel, ambele funcții (`f` și `g`) trebuie să fie definite sau să li se indice prototipurile.

Fie `g` de prototip:

`tipg g(listag);`

unde prin `tipg` s-a notat tipul valorii returnate de `g` sau `void` în cazul în care funcția `g` nu returnează o valoare, iar `listag` este lista cu tipurile parametrilor funcției `g` sau cuvântul `void` dacă `g` nu are parametri.

Să definim prototipul funcției `f`. Așa cum s-a indicat mai sus, ea are un parametru care este un pointer spre o funcție care are un prototip de forma prototipului lui `g` (numai numele poate差别).

O construcție de forma:

`tip *nume`

definește pe `nume` ca pointer spre tipul `tip`.

O construcție de forma:

`tip *nume(lista)`

definește pe `nume` ca funcție care returnează un pointer spre `tip`.

Prezența parantezelor conduce la faptul că `nume` este o funcție și nu un pointer. Aceasta deoarece parantezele rotunde reprezintă operatorii mai prioritari decit operatorul unar `*`. Pentru ca `nume` să fie pointer, este nevoie ca operatorul unar `*` să se aplique prioritar față de parantezele care indică prezența unei funcții. Aceasta se obține folosind o construcție de forma:

`tip (*nume)(lista)`

În cazul de față `nume` este un pointer spre o funcție. Valoarea returnată de această funcție are tipul `tip`, iar `lista` definește tipurile parametrilor acestei funcții.

În cazul funcției `f` de mai sus, parametrul ei formal se definește astfel:

`tipg (*nume_par_formal)(listag)`

Amintind faptul că într-un prototip se pot omite numele parametrilor formali, rezultă că prototipul funcției `f` poate fi scris astfel:

`tipf(tipg (*)(listag));`

unde *tipf* este tipul valorii returnate de funcția *f* sau *void* în cazul în care *f* nu returnează o valoare.

Antetul funcției *f* va fi:

*tipf f(tipg (*p)(listag))*

- unde *listag* definește tipurile parametrilor funcțiilor care se transferă la apelurile lui *f* prin pointerii spre ele.

Exemple:

1. Funcția *g* are prototipul:

int g(void);

Funcția *f* care are la apel ca parametru efectiv pe *g* și returnează o valoare de tip *double*, are prototipul:

double f(int (*)(void));

2. Funcția *h* are prototipul:

double h(int,float);

Funcția *f* care are la apel ca parametru efectiv pe *h* și care returnează o valoare de tip *int*, are prototipul:

int f(double (*)(int,float));

Fie funcția *f* care are antetul:

*tipf f(tip (*p)(lista))*

Deci, parametrul *ci p* este un pointer spre o funcție care returnează o valoare de tipul *tip*, iar tipurile parametrilor acestei funcții sunt definite de *lista*.

La un apel de forma:

f(g);

lui *p* îi se atribuie ca valoare pointerul spre funcția *g*. Se pune problema de a apela funcția *g* în corpul funcției *f*, folosind parametrul formal *p*. În acest caz, apelul obișnuit al lui *g*:

g(...);

sau

x=g(...);

se schimbă înlocuind numele *g* (necunoscut în momentul definirii funcției *f*) prin **p*.

Deci în corpul funcției *f*, vom utiliza apeluri de forma:

*(*p)(...);*

sau

*x=(*p)(...);*

pentru a apela funcția care s-a transferat prin parametru.

Exerciții:

8.25 Să se scrie o funcție care calculează și returnează valoarea aproximativă a integraliei definite dintr-o funcție *f(x)*, folosind metoda trapezului.

Să presupunem că integrala se calculează de la *a* la *b*. Atunci o valoare aproximativă a integralei se determină cu ajutorul formulei de mai jos:

$$In = h * ((f(a) + f(b)) / 2 + f(a + h) + f(a + 2 * h) + f(a + 3 * h) + \dots + f(a + (n - 1) * h))$$

unde *h* = (*b* - *a*) / *n*.

Funcția de față calculează partea dreaptă a acestei relații. Ea are următorii parametri:

- | | |
|----------|---|
| <i>a</i> | - Limita inferioară - număr de tip <i>double</i> . |
| <i>b</i> | - Limita superioară - număr de tip <i>double</i> . |
| <i>n</i> | - Numărul de subintervale în care s-a impărțit intervalul [<i>a</i> , <i>b</i>] - intreg de tip <i>int</i> . |
| <i>p</i> | - Pointerul spre funcția <i>f(x)</i> din care se calculează integrala - aceasta are un parametru de tip <i>double</i> și returnează o valoare de tip <i>double</i> (<i>double f(double x)</i>). |

FUNCȚIA BVIII25

```
double itrapez(double a, double b, int n, double (*p)(double))
/* - calculează prin metoda trapezului, integrala definită de la a la b din funcția spre care
   pointează p;
   - n - este numărul subintervalelor în care s-a impărțit intervalul [a,b]*/
{
    double h,s;
    int i;

    h = (b - a) / n;
    s = 0.0;
    for(i=1; i < n; i++) s += (*p)(a + i * h);
    s += ((*p)(a) + (*p)(b)) / 2;
    s *= h;
    return s;
}
```

8.26 Să se scrie un program care calculează integrala definită din funcția:

$$f(x) = \sin(x^*x)$$

din 0 la 1, folosind metoda trapezului. Se cere o eroare mai mică decât $1e-8$.

Pentru a obține precizia dorită vom proceda ca mai jos:

1. Se alege valoarea inițială pentru *n*, de exemplu 10.
2. Se calculează *In*.
3. Se calculează *I2n* (*n* este dublat).
4. Dacă $\text{abs}(I2n - In) < 1e-8$, atunci *I2n* este rezultatul și procesul de calcul se

intrerupe.
Altfel se dublează n , se pune $\ln = \ln_2 n$ și se continuă cu pasul 3 de mai sus.

PROGRAMUL BVIII26

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

#include "bviii25.cpp"

#define A 0.0
#define B 1.0
#define N 10
#define EPS 1e-8

double sinxp(double); /* prototip */
main() /* calculeaza integrala definită din funcția
           sinxp=sin(x*x)
           în intervalul [A,B] cu o eroare mai mică decit EPS */
{
    int n = N;
    double in,i2n,vabs;

    in = itraperz(A,B,n,sinxp);
    do {
        n = 2*n;
        i2n = itraperz(A,B,n,sinxp);
        if((vabs = in - i2n) < 0) vabs = -vabs;
        in = i2n;
    } while(vabs >= EPS);
    printf("integrala din sin(x*x)=%.10g\n",i2n);
    printf("numarul de subintervale n=%d\n", n);
} /* sfârșit main */

double sinxp(double x)
{
    return sin(x*x);
}
```

Observație:

Valoarea aproximativă a integralei din $\sin(x^*x)$ în intervalul $[0,1]$ este 0.3102683026. Ea se obține pentru $n=10240$ în aproximativ 40 de secunde, la un calculator fără coprocesor.

8.27 Să se scrie o funcție care determină rădăcina ecuației:

(1) $f(x)=0$
din intervalul $[a,b]$, cu o eroare mai mică decit $\text{EPS}=1e-8$, știind că ecuația are o singură radăcină în intervalul respectiv și că $f(x)$ este o funcție continuă pe același interval.

Din condițiile enunțate mai sus rezultă că:

$$f(a)*f(b) \leq 0$$

sau

$$f(a)*f(b) < 0$$

dacă nici una din limitele intervalului $[a,b]$ nu sunt rădăcini ale ecuației (1). O metodă simplă de calcul a rădăcinii ecuației (1) este prin metoda înjumătățirii. Ea constă în următoarele:

1. $c = (a+b)/2$.
2. Dacă $f(c)=0$, atunci c este soluția căutată și se intrerupe procesul de calcul.
3. Dacă $f(a)*f(c) < 0$, atunci se pune $b=c$, altfel $a=c$.
4. Dacă $|b-a| < \text{EPS}$, atunci procesul de calcul se intrerupe și rădăcina se ia media dintre a și b .

Altfel algoritmul se reia de la pasul 1.

FUNCȚIA BVIII27

```
#define EPS 1e-8

double radec(double a, double b,
              double (*p)(double))
/* - calculeaza si returnaza soluția ecuației f(x) = 0 din intervalul [a,b];
   - ecuația are o singură soluție în intervalul [a,b];
   - p - este pointerul spre funcția f(x) de prototip:
        double f(double). */
{
    double c,d;

    if((*p)(a) == 0) return a; /* a este radacina */
    if((*p)(b) == 0) return b; /* b este radacina */
    if((*p)(a) * (*p)(b) > 0) {
        printf("ecuația sau nu are radacina în [a,b]\n");
        printf("sau are radacini multiple în [a,b]\n");
        exit(1);
    }
    do {
        c = (a + b)/2;
        if((d = (*p)(c)) == 0) return c; /* c este radacina */
        if((*p)(a)*d < 0) b = c;
        else a = c;
    } while(|b-a| >= EPS);
    return (a+b)/2;
}
```

8.28 Să se scrie un program care calculează și afișează rădăcina ecuației:

$x-\sin(x+1)=0$
din intervalul $[0.5,1]$, cu o eroare mai mică decit $1e-8$.

PROGRAMUL BVIII28

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

#include "bvi27.cpp"

#define A 0.5
#define B 1.0

double f(double x) /* returnaza valoarea functiei x - sin(x+1) */
{
    return x - sin( x+1 );
}

main () /* calculeaza si afiseaza radacina ecuatiei:
           x - sin(x+1) = 0
           din interval [0.5,1], cu o eroare mai mica decit 1.e-8 */
{
    printf("radacina ecuatiei: \n");
    printf("x - sin(x+1) = 0 este: %.10g\n", radec(A,B,f));
}
```

8.12. Tablouri de pointeri

În paragraful 8.9, s-au utilizat tablouri de pointeri spre siruri de caractere. Astfel, s-a definit tabloul *tpdl*, care este un tablou de pointeri spre siruri de caractere care sunt denumirile lunilor calendaristice:

- *tpdl[1]* este pointer spre sirul de caractere "ianuarie";
- *tpdl[2]* este pointer spre sirul de caractere "februarie"
- și aşa mai departe.

Acest tablou a fost declarat astfel:

```
char *tpdl[] = {
    "luna ilegală",
    "ianuarie",
    "februarie",
    ...
    "decembrie"
};
```

Evident, dacă în declarația de mai sus lipsesc sirurile de caractere prin care se inițializează elementele tabloului *tpdl*, atunci este necesar să se indice, în declarația respectivă, numărul elementelor tabloului *tpdl*.

Astfel, în locul declarației de mai sus, putem utiliza o declarație de forma:

```
char *tpdl[13];
```

urmând ca în continuare să se inițializeze elementele

tpdl[0], tpdl[1], tpdl[2],...,tpdl[12]

cu pointeri spre sirurile de caractere "luna ilegală", "ianuarie", etc.

Numele *tpdl*, poate fi folosit ca parametru efectiv la apeluri de funcții. Fie, de exemplu, funcția *afis* care se apelează astfel:

```
afis(tpdl);
```

Parametrul formal al funcției *afis*, corespunzător parametrului efectiv *tpdl*, se declară ca fiind tablou de pointeri spre siruri de caractere, adică funcția *afis* are un antet de forma:

```
...afis(char *p[])
```

În paragraful 8.3, am văzut că numele unui tablou poate fi asimilat cu un pointer. Astfel, dacă *f* are antetul

```
...f(tip t[])
```

atunci parametrul *t* poate fi declarat ca un pointer spre *tip* și deci antetul lui *f* poate fi scris și sub forma:

```
...f(tip *t)
```

În mod similar, parametrul *p* fiind un tablou poate fi declarat înlocuind *p[]* cu **p*:

```
...afis(char **p)
```

Deoarece *p[i]* este un pointer spre un sir de caractere, caracterele unui astfel de sir pot fi accesate prin expresii de forma:

```
*(p[i]), *(p[i]+1), *(p[i]+2), ..., *(p[i]+j), ...
```

sau utilizând indicii:

```
p[i][0], p[i][1], p[i][2], ..., p[i][j], ...
```

Un exemplu de parametru formal care este un tablou de pointeri spre siruri de caractere, este parametrul *argv* al funcției *main* amintit în paragraful 8.10. Acest parametru a fost declarat astfel:

```
char *argv[]
```

Accesul la caracterele argumentului liniei de comandă spre care pointează *argv[i]* se poate realiza prin expresii de forma:

```
*(argv[i]+j), *(*(argv+i)+j) sau argv[i][j]
```

Tablourile de pointeri se pot utiliza în general, pentru orice tip. Astfel, o declarație de forma:

```
tip *nume[...];
```

este corectă și prin această declarație *nume* se definește ca fiind un tablou de

pointeri spre tipul *tip*.

Fie *f* o funcție care are la apel ca parametru efectiv pe *nume* declarat ca mai sus:

```
f(nume);
```

Funcția *f* va avea antetul:

```
...f(tip *p[])
```

care mai poate fi scris și sub forma:

```
...f(tip **p)
```

În corpul funcției *f*, pointerul *p* poate fi utilizat în expresii de forma:

```
*(*(p+i)+j), *(p[i]+j) sau p[i][j]
```

Declarația tabloului *nume*, de mai sus, poate fi înlocuită cu o declarație de forma:

```
tip *nume;
```

Această ultimă declarație este preferată adesea, deoarece ea nu necesită prezența expresiei din parantezele patrate, expresie care trebuie să fie o *expresie constantă*, adică valoarea ei trebuie să poată fi evaluată de compilator, la inițierea ei.

Un tablou declarat în acest fel poate fi utilizat ca un tablou bidimensional. Astfel, accesul la elementele lui se realizează prin construcții de forma:

```
*(*(nume+i)+j), *(nume[i]+j) sau nume[i][j]
```

Să observăm că deși *nume* declarat ca mai sus, poate fi utilizat folosind doi indici, el diferă de tablourile bidimensionale declarate în mod obișnuit prin declarații de forma:

```
tip nume[ec1][ec2];
```

unde *ec1* și *ec2* sunt expresii constante. Într-adevăr, în acest caz, compilatorul aloca o zonă de memorie de dimensiune egală cu:

```
ec1*ec2*sizeof(tip)
```

octeți.

La o declarație de forma:

```
tip **nume;
```

compilatorul aloca o zonă de memorie de dimensiune egală cu:

```
sizeof(tip *)
```

octeți. În acest caz, pentru a utiliza pointerul *nume* la fel ca atunci cind acesta este declarat ca un tablou bidimensional, utilizatorul trebuie să aloce memorie pentru

tabloul de pointeri *nume*, folosind funcția *malloc*.

De exemplu, dacă considerăm declarația:

```
double nume[10]{4};
```

atunci pentru *nume* se alocă:

```
10*4*sizeof(double)=40*8=320
```

octeți.

Dacă considerăm declarația:

```
double **nume;
```

atunci pentru *nume* se alocă:

```
sizeof(double *)
```

octeți. Această expresie are valoarea 2 dacă pointerii se alocă pe 16 biți sau 4 dacă se alocă pe 32 de biți. Amintim că alocarea memoriei pentru pointeri este în funcție de modelul de memorie utilizat.

Pentru a aloca memorie *heap* în cazul numelui declarat prin:

```
tip **nume;
```

înținem seama de faptul că *nume* declarat în acest fel este un tablou de pointeri. Dacă acest tablou are *m* elemente, atunci pentru el se aloca o zonă de memorie de

```
m*sizeof(tip *)
```

octeți. Aceasta se realizează astfel:

```
nume=(tip **)malloc(m*sizeof(tip *));
```

În continuare, este necesar să se aloce memorie *heap* spre care să pointeze elementele tabloului *nume*, adică:

```
nume[0], nume[1],...,nume[m-1].
```

Să presupunem că *nume[i]*, pentru *i*=0,1,...,m-1 pointează spre o zonă de memorie capabilă să păstreze *n* elemente de tipul *tip*. Atunci zonele de memorie spre care vor pointa elementele *nume[i]*, pentru *i*=0,1,...,m-1, se pot aloca utilizând secvența de mai jos:

```
for(i=0; i<m; i++) nume[i]=(tip *)malloc(n*sizeof(tip));
```

După aceste alocări *nume* poate fi utilizat folosind doi indici, la fel ca în cazul cind ar fi declarat prin:

```
tip nume[m][n];
```

cu condiția ca *m* și *n* să fie expresii constante.

Se observă că în cazul declarației

```
tip **nume;
```

nu mai sunt necesare constantele *m* și *n*. Alocarea făcindu-se la execuție, valorile

m și n pot varia de la o execuție a programului la alta.

Exerciții:

- 8.9 În exercițiul 8.6 se definește funcția *pdcitmat* pentru citirea elementelor unei matrice. Elementele matricei sunt păstrate într-un tablou unidimensional, prin liniarizare. Mai jos definim funcția *citmat* care citește elementele unei matrice și le păstrează într-o zonă de memorie alocată unui tablou de pointeri numit *mat*. În felul acesta elementele matricei pot fi accesate folosind doi indici.

Funcția *citmat* utilizează funcția *ndcit* pentru a citi elementele unei linii a matricei. De asemenea, pentru citirea numărului de linii și a numărului de coloane, se apelează funcția *pcit_int_lim*.

FUNCȚIA BVIII29

```
int pcit_int_lim(char text[], int inf, int sup, int *pint);
int ndcit(int n, double linie[]);

#define ER {
    printf("S-a tastat sfirsitul de fisier\n");
    return 0;
}

#define ERM {
    printf("Memorie insuficienta\n");
    exit(1);
}

double **citmat(int *nrlin, int *nrcol)
/*- citeste pc:
   m - numar de linii;
   n - numar de coloane;
   aloca memorie pentru tabloul de pointeri mat;
   citește m*n numere de tip double și le păstrează în zona alocată pentru mat;
   m se păstrează în zona de memorie spre care pointează nrlin;
   n se păstrează în zona de memorie spre care pointează nrcol;
   returnaza:
   0 - la eroare;
   mat - altfel
*/
{
    double **mat;
    int i, m, n;

    /* se citește numarul linijilor și numarul coloanelor */
    if (pcit_int_lim("Nr. linii=", 1, 5000, &m) == 0) ER
    if (pcit_int_lim("Nr. coloane=", 1, 5000, &n) == 0) ER
    *nrlin = m;
    *nrcol = n;
```

```
/* se aloca memorie heap pentru tabloul de pointeri mat */
if ((mat = (double **)malloc(m * sizeof(double *))) == 0) ERM

/* se aloca memorie pentru fiecare linie a matricei */
for (i = 0; i < m; i++)
    if ((mat[i] = (double *)malloc(n * sizeof(double))) == 0) ERM

/* se citește elementele matricei și se păstrează în zonele rezervate lor */
for (i = 0; i < m; i++) {
    printf("Linia %d - ", i + 1);
    if (ndcit(n, mat[i]) != n) {
        printf("Nu s-au tastat %d elemente\n", n);
        return 0;
    }
}
return mat;
}
```

- 8.30 Programul definit mai jos citește elementele a două matrice de tip *double*, calculează și afișează produsul lor.

Programul de față este similar cu cel definit în exercițiul 8.7. Deosebirea dintre cele două programe constă în accea că, în timp ce în programul din exercițiul 8.7 matricele se păstrează prin liniarizare, în cazul de față, se utilizează tablouri de pointeri. În acest scop, pentru citirea celor două matrice se utilizează funcția *citmat* definită în exercițiul 8.29.

PROGRAMUL BVIII30

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <alloc.h>
#include "bviii2.cpp" /* functia pcit_int */
#include "bviii3.cpp" /* functia pcit_int_lim */
#include "biv37.cpp" /* functia ndcit */
#include "bviii29.cpp" /* functia pdcitmat */

main()
/* citeste elementele a două matrice, calculeaza și afiseaza matricea produs,
   cate 4 elemente pe o linie
*/
{
    int i, j, k;
    int m, n, p, s, q;
    int mn, np;
    double **a, **b, **c;

    /* citeste matrica a */
    printf("Matricea a\n");
    a = citmat(&m, &n);
```

```

/* citeste matricea b */
printf("Matricea b\n");
b=editmat(&p,&s);
if(n!=p) {
    printf("numarul coloanelor matricei a = %d \n",n);
    printf("difera de numarul liniilor");
    printf("matricei b = %d \n",p);
    exit(1);
}

/* se aloca memorie pentru tabloul de pointeri c */
if((c=(double **)malloc(m*sizeof(double *)))==0) ERM;

for(i=0;i<m;i++)
    if((c[i]=(double *)malloc(s*sizeof(double)))==0) ERM;

/* se realizeaza produsul c = a*b */
for(i=0; i < m; i++)
    for(j=0; j < s; j++) {
        c[i][j] = 0.0;
        for(k=0; k < n; k++)
            c[i][j] += a[i][k]* b[k][j];
    }

/* afiseaza elementele matricei produs */
k=1;
printf("\n\n\tmatricea produs\n");
for(i=0;i<m;i++){
    for(j=0;j<s;j++) {
        printf("c[%d,%d]=%8g ",i,j,c[i][j]);
        if(j%4 == 3) {
            /* afiseaza 4 elemente pe un rind */
            printf("\n");
            k++; /* numara randurile */
        }
    }
    if(k==23) {
        printf("actionati o tasta pentru a continua\n");
        getch();
        k=1;
    }
}
printf("\n");
k++;
}
}

```

9. RECURSIVITATE

Spunem despre o funcție că este *recursivă* dacă ca se autoapeleză. Ea se poate autoapela fie *direct*, fie *indirect* prin apelul altor funcții.

În limbajele C și C++ se pot defini funcții recursive fără a fi nevoie de a specifica acest lucru.

La apelurile recursive ale unei funcții aceasta este reapelată înainte de a se reveni din ea. La fiecare reapel a funcției, *parametrii* și *variabilele automate* ale ei se alocă pe stivă într-o zonă independentă. De aceea, aceste date au valori distincte la fiecare reapelare. Nu același lucru se întâmplă cu *variabilele statice*. Ele ocupă tot timpul aceeași zonă de memorie și deci ele își păstrează valoarea la un reapel.

Orice apel al unei funcții conduce la o revenire din funcția respectivă la punctul următor celui din care s-a facut apelul.

Amintim că la revenirea dintr-o funcție se procedează la curățirea stivei, adică stiva se refacă la starea ei dinaintea apelului. De aceea, orice reapel al unui apel recursiv va conduce și el la curățirea stivei, la o revenire din funcție și deci parametrii și variabilele locale vor reveni la valorile lor dinaintea reapelului respectiv. Numai variabilele statice rămân neafectate la o astfel de revenire.

Funcțiile recursive se utilizează la definirea proceselor recursive de calcul.

Un proces de calcul se spune că este *recursiv*, dacă el are o parte care se definește prin el însuși. Un exemplu simplu de astfel de funcție este funcția de calcul al *factorialului*. Într-adevăr funcția *fact(n)* de calcul al factorialului se poate defini astfel:

fact(n)=1, dacă n=0;

altfel

*fact(n)=n*fact(n-1).*

Alternativa:

*fact(n)=n*fact(n-1)*

definește funcția *fact(n)* prin ea însăși.

Un proces recursiv trebuie totdeauna să conțină o parte care nu se definește prin el însuși. În exemplul de mai sus, alternativa:

fact(n)=1, dacă n=0

este partea care este definită direct.

Definiția de mai sus se transcrie imediat în limbajul C:

```

double fact(int n) /* calculează n! */
{
    if(n==0)
        return 1.0;
}

```

```

    else
        return n*fact(n-1);
}

```

Să urmărим execuția acestei funcții pentru $n=3$.

La apelul din funcția principală se construiește o zonă pe stivă în care se păstrează adresa de revenire în funcția principală, după apel, precum și valoarea lui n , ca în figura de mai jos.

Stiva programului

rev	adr1
n	3
a	

Adresa de revenire în funcția principală care a făcut apelul

Deoarece $n > 0$, se execută alternativa *else*. Aceasta conține expresia:

(1) $n * fact(n-1)$

care autoapelează (direct) funcția *fact*. Reapelarea funcției se realizează cu valoarea $n-1 = 3-1 = 2$. Prin aceasta se construiește o zonă nouă pe stivă, care conține revenirea la evaluarea expresiei de mai sus, precum și valoarea nouă a parametrului n , adică valoarea 2. Se obține starea din figura de mai jos.

Stiva programului

rev	adr2
n	2
rev	adr1
n	3
b	

Adresa de revenire la evaluarea expresiei (1)

La noua reapelare a funcției *fact*, $n=2 > 0$, deci din nou se execută alternativa *else*.

Evaluarea expresiei (1) conduce la o reapelare cu valoarea $n-1 = 2-1 = 1$. Se obține configurația:

Stiva programului

rev	adr2
n	1
rev	adr2
n	2
rev	adr1
n	3
c	

Adresa de revenire la evaluarea expresiei (1)

Din nou $n = 1 > 0$ și deci se face o nouă reapelare și se obține configurația:

Stiva programului

rev	adr2
n	0
rev	adr2
n	1
rev	adr2
n	2
rev	adr1
n	3
d	

Adresa de revenire la evaluarea expresiei (1)

În acest moment $n=0$, deci se revine din funcție cu valoarea 1. Se curăță stiva și se revine la configurația din figura c. Adresa de revenire permite continuarea evaluării expresiei (1). În acest moment n are valoarea 1 și cum s-a revenit tot cu valoarea 1, se realizează produsul

$$1 * 1 = 1$$

După evaluarea expresiei (1) se realizează o nouă revenire tot cu valoarea 1. După curățirea stivei se ajunge la configurația din figura b. În acest moment n are valoarea 2. Evaluind expresia (1), se realizează produsul

$$2 * 1 = 2$$

Apoi se revine din nou din funcție cu valoarea 2. După curățirea stivei se

ajunge la configurația din figura a.

În acest moment n are valoarea 3. Se calculează expresia (1) și se obține:

$$3 \cdot 2 = 6$$

Se revine din funcție cu valoarea 6 (3!) și conform adresei de revenire, se revine în funcția principală din care s-a apelat funcția *fact*.

În general, o funcție recursivă se poate realiza și nerecursiv, adică fără să se autoapeleze. De obicei, recursivitatea nu conduce la economie de memorie și nici la execuția mai rapidă a programelor. Ea permite însă o descriere mai compactă și mai clara a funcțiilor care exprimă procese de calcul recursive. Cu toate acestea, există cazuri cînd funcțiile recursive, deși au exprimări clare, ele nu sunt recomandate deoarece implică consum mare de timp și memorie, în comparație cu varianta nerecursivă a același proces de calcul. De exemplu, algoritmul de generare a permutărilor de n obiecte poate fi descris atât recursiv, cât și nerecursiv. O varianta recursivă simplă generează permutările de n obiecte inserind un obiect nou (n) în toate pozițiile posibile în permutările de $n-1$ obiecte. Acest algoritm, deși este mai simplu decît varianta nerecursivă, el nu este practic, deoarece generația permutărilor de n obiecte presupune existența deja în memorie a permutărilor de $n-1$ obiecte, ori $n!$ crește foarte rapid, ceea ce conduce la un necesar mare de memorie chiar și pentru n mic. Se pot indica și alte funcții recursive inefficiente față de variantele lor nerecursive. De exemplu, în multe lucrări se prezintă funcție recursivă pentru calculul *numerelor lui Fibonacci* și se arată ineficiența ei față de varianta nerecursivă.

De aceea, rezolvările cu ajutorul funcțiilor recursive urmează să se adopte numai după un studiu atent al implicațiilor pe care le au acestea în privința necesarului de memorie și al timpului de execuție.

Există cîteva tipuri de probleme care, de obicei, se rezolvă cu ajutorul funcțiilor recursive. Astfel, se recomandă să se utilizeze funcții recursive pentru probleme a căror metode de rezolvare se pot defini recursiv.

În aceasta clasa se includ metode de divizare, metode de căutare cu revenire (backtracking) etc.

În toate cazurile trebuie însă să se aibă în vedere necesarul de memorie și a timpului de execuție în comparație cu variantele nerecursive.

Exerciții:

- 9.1 Să se scrie un program care citește pe n de tip *int*, calculează și afișează $n!$ folosind funcția recursivă *fact* definită mai sus.

PROGRAMUL BIX1

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

#define MAX 170
```

```
double fact(int n) /* calculeaza pe n! */
{
    if(n == 0) return 1.0;
    else return n * fact(n-1);
}

main () /* citeste pe n, calculeaza si afiseaza pe n! */
{
    int n;
    char t[255];

    for( ; ; ) {
        printf("valoarea lui n: ");
        if(gets(t) == 0) {
            printf("s-a tastat EOF\n");
            exit(1);
        }
        if(sscanf(t, "%d", &n) == 1 && n >= 0 && n <= MAX) break;
        printf("se cere un intreg in intervalul [0,%d]\n", MAX);
    }
    printf("n= %d\tn!= %g\n", n, fact(n));
}
```

- 9.2 Să se scrie o funcție recursivă care calculează numărul aranjamentelor de n obiecte luate cîte k ($n \geq k > 0$).

Dacă notăm cu $A(n,k)$ numărul aranjamentelor de n obiecte luate cîte k , atunci:

$$\begin{aligned} A(n,k) &= n(n-1)(n-2)\dots(n-k+1) = n(n-1)(n-2)\dots((n-1)-(k-1)+1) = \\ &= nA(n-1,k-1); \\ A(n,1) &= n. \end{aligned}$$

Relația de mai sus, pentru calculul numărului aranjamentelor, este recursivă. Ea se transcrie imediat în limbajul C folosind o funcție recursivă.

FUNCȚIA BIX2

```
double aranjamente(int n, int k)
/* returneaza numarul aranjamentelor de n obiecte luate cîte k (n>=k>0) */
{
    if( k == 1 ) return (double) n;
    else return n*aranjamente(n-1,k-1);
}
```

- 9.3 Să se scrie un program care citește doi intregi n și k ($1 \leq n \leq 170, 1 \leq k \leq n$), calculează și afișează numărul aranjamentelor de n obiecte luate cîte k .

Valorile lui n și k se citesc folosind funcția *pcit_int_lim* definită în exercițiul BVIII3.CPP. Această funcție, la rîndul ei, apelează funcția *pcit_int* definită în exercițiul BVIII2.CPP.

PROGRAMUL BIX3

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

#include "bviii2.cpp"
#include "bviii3.cpp"
#include "bix2.cpp"
#define MAX 170

main() /* - citeste intregii n si k (1 <= n <= 170; 1 <= k <= n),
           - calculeaza si afiseaza numarul aranjamentelor de n obiecte luate cu k */
{
    int n,k;
    char er[]="S-a tastat EOF\n";
    if(pcit_int_lim("n= ",1,170,&n) == 0 ) {
        printf(er);
        exit(1);
    }
    if(pcit_int_lim("k= ",1,n,&k) == 0 ) {
        printf(er);
        exit(1);
    }
    printf("n=%d\tn=%d\tA(n,k)=%g\n",n,k,aranjamente(n,k));
}
```

- 9.4 Să se scrie o funcție care are ca parametri doi intregi m și n de tip *long* și care calculează și returnează cel mai mare divizor comun al lor.

Notăm cu:

(m,n)

cel mai mare divizor comun al numerelor m și n . Calculul celui mai mare divizor comun a două numere se poate realiza recursiv astfel:

$(m,n) = m$	dacă $n = 0$;
$(m,n) = n$	dacă $m = 0$;
$(m,n) = (n,m \% n)$	dacă atât m , cât și n sunt diferenți de zero și $m > n$ (prin $m \% n$ s-a notat restul împărțirii lui m la n).

FUNCTIA BIX4

```
long cmmdc(long m, long n)
/* calculeaza si returneaza pe cel mai mare divizor comun al numerelor m si n */
{
    if( m == 0 ) return n;
    else
        if( n == 0 ) return m;
        else
            if( m > n ) return cmmdc( n, m%n );
            else
                return cmmdc( m, n%m );
```

- 9.5 Să se scrie un program care citește doi întregi pozitivi m și n , de tip *long*, calculează și afișează pe cel mai mare divizor comun al lor.

PROGRAMUL BIX5

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

#include "bix4.cpp"

main() /* citeste pe m si n de tip long, calculeaza si afiseaza (m,n) */
{
    long m,n;

    printf("valorile lui m si n: ");
    if(scanf("%ld %ld", &m, &n) != 2 || m <= 0 || n <= 0 ) (
        printf("nu s-au tastat doi intregi pozitivi\n");
        exit(1);
    )
    printf("m=%ld\tn=%ld\t(m,n)=%ld\n",m,n,cmmdc(m,n))
}
```

- 9.6 Să se scrie un program pentru rezolvarea problemei *turnurile din Hanoi*. Această problemă se enunță ca mai jos.

Se consideră trei tije A, B, C și n discuri de diametre diferite. Tijele sunt fixate vertical pe o placă. Discurile au fiecare cîte un orificiu în centru și ele pot fi aranjate pe fiecare din cele trei tije. Inițial toate discurile sunt puse pe tija A și ordonate în așa fel încît orice disc este așezat peste un disc de diametru mai mare decit al lui. Dacă numerotăm discurile în ordinea crescătoare a diametrelor lor, adică discul 1 are diametrul cel mai mic, apoi discul 2 are diametrul următor și aşa mai departe, discul n avind diametrul cel mai mare, atunci discul 1 se află în virf, sub el discul 2 etc., iar la bază se află discul n .

Se cere să se mute discurile de pe tija A pe tija B, folosind tija C ca tijă de manevră. La fiecare mutare se mută un singur disc și anume, unul aflat în virful discurilor de pe o tijă se pune în virful discurilor de pe o altă tijă sau pe placă dacă tija respectivă nu conține discuri. De asemenea, nu se poate muta un disc de diametru mai mare peste unul de un diametru mai mic. Deci, la o mutare, peste discul k se poate așeza numai unul din discurile $1, 2, \dots, k-1$. În final discurile trebuie să fie așezate pe tija B în aceeași ordine în care s-au aflat pe tija A, adică în virf să fie discul 1, sub el discul 2 și aşa mai departe, discul n aflându-se la bază.

Problema turnurilor din Hanoi pentru n discuri se rezolvă imediat dacă știm să o rezolvăm pentru $n-1$ discuri. Într-adevăr, în ipoteza că știm să rezolvăm problema pentru $n-1$ discuri, procedăm astfel:

1. Se deplasează discurile $1, 2, \dots, n-1$ de pe tija A pe tija C.
2. Discul n se mută de pe tija A pe tija B.

3. Cele $n-1$ discuri de pe tija C se deplasează pe tija B.

În felul acesta, pe tija B se află toate cele n discuri și în ordinea cerută, adică $1, 2, \dots, n$.

Deci rezolvarea problemei turnurilor din Hanoi cu n discuri s-a redus la rezolvarea acelorași probleme, dar cu $n-1$ discuri. În mod analog, problema turnurilor din Hanoi cu $n-1$ discuri se poate rezolva dacă știm să rezolvăm problema respectivă pentru $n-2$ discuri. În felul acesta, din aproape în aproape, ajungem la concluzia că problema turnurilor din Hanoi pentru n discuri se poate rezolva, dacă știm să o rezolvăm pentru $n=1$. Ori în acest caz problema este imediată: se mută discul de pe tija pe care se află (tija sursă), pe tija pe care dorim să se afle discul (tija destinație).

Cele trei tije se află în una din următoarele stări posibile:

- | | |
|------------------------|---|
| <i>tija sursă</i> | - Conține discurile care trebuie transferate pe alta tija. |
| <i>tija destinație</i> | - Tija pe care trebuie să se afle discurile transferate de pe tija sursă. |
| <i>tija de manevră</i> | - Tija care se folosește pentru a stoca temporar discuri. |

Rolul tijelor se schimbă în diferite etape ale procesului de deplasare a discurilor și de aceea, rolul fiecărei tije se definește cu ajutorul parametrilor.

Mai jos definim o funcție recursivă pe care o numim *hanoi*. Ea are 4 parametri:

- | | |
|----------|-----------------------|
| <i>n</i> | - Numărul discurilor. |
| <i>a</i> | - Tija sursă. |
| <i>b</i> | - Tija destinație. |
| <i>c</i> | - Tija de manevră. |

Apelul:

```
hanoi(n,'A','B','C');
```

înseamnă că se rezolvă problema pentru n discuri. Acestea se află pe tija A, tija B este tija destinație și tija C este tija de manevră.

Funcția *hanoi* afișază fiecare mutare, indicind numărul discului, tija de pe care se mută discul respectiv, precum și tija pe care se mută el.

După afișarea unei mutări, se apelează funcția *getch* pentru a bloca execuția programului. În felul acesta se poate analiza mutarea afișată de calculator.

Menționăm că problema de față poate fi rezolvată și nerecursiv. Varianta nerecursivă este mai complexă decit cea recursivă.

PROGRAMUL BIX6

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

```
void hanoi(int n, int a, int b, int c)
/* funcție recursivă pentru rezolvarea problemei turnurilor din Hanoi,
   n - numărul discurilor;
   a - tija sursă;
   b - tija destinație;
   c - tija de manevră. */

{
    if(n == 1) { /*n=1*/
        printf("se mută discul 1 de pe tija: %c \
               pe tija: %c\n",a,b);
        getch();
        return;
    }

    /* n>1;
       se rezolvă problema pentru n-1 discuri;
       a - tija sursă; c - tija destinație; b - tija de manevră. */
    hanoi(n-1, a, c, b);

    /* discul n de pe tija a se mută pe tija b */
    printf("discul: %d de pe tija: %c se mută pe \
           tija: %c\n",n,a,b);
    getch();

    /* se rezolvă problema pentru n-1 discuri;
       c - tija sursă; b - tija destinație; a - tija de manevră. */
    hanoi(n-1, c, b, a);
}

/* sfîrșit hanoi */

main () /* citeste pe n și rezolvă problema turnurilor din Hanoi pentru n discuri */
{
    int n;
    char t[255];

    for( ; ; ) {
        printf("numărul discurilor= ");
        if(gets(t) == 0) {
            printf("s-a tastat EOF\n");
            exit(1);
        }
        if (sscanf(t,"%d",&n) == 1 && n > 0 )
            break;
        printf("nu s-a tastat un intreg pozitiv\n");
    }
    hanoi(n, 'A', 'B', 'C');
}
```

