Prefaţă

Această carte se adresează celor interesaţi în a pătrunde "spiritul" limbajului C. Mulţi dintre cei care au învăţat acest limbaj au ajuns să-l cunoască din cărţi care prezentau exemple de tipul numeric sau grafic, însă **Brian W. Kernighan** şi **Dennis M. Ritchie**, atunci când au inventat limbajul C, aveau în minte un singur lucru: punerea la punct a unui limbaj care să permită scrierea unui sistem de operare cu efort minim. Toată istoria limbajului C şi multe dintre ciudăţeniile lui sunt strâns legate de sistemul de operare UNIX. Portarea lui sub sistemul de operare DOS, după umila mea părere, unul dintre cele mai slabe sisteme de operare scrise pe această planetă, a încurcat lucrurile şi mai mult.

Cartea prezintă numai C standard. Majoritatea firmelor producătoare de compilatoare aderă la acest standard dar asigură, în plus, o mulţime de facilităţi care pot fi cunoscute numai prin citirea documentaţiei de firmă a compilatorului. Lucrarea s-a născut ca un protest împotriva celor care au folosit C fără a citi "THE C PROGRAMMING LANGUAGE, SECOND EDITION" a lui Brian W. Kernighan şi Dennis M. Ritchie respectiv standardul ANSI ajungând să-l aplice în domenii în care alte limbaje de programare îi sunt net superioare. Pentru cei interesaţi în calculul numeric ANEXA 1 prezintă avantajele actuale ale bătrânului FORTRAN faţă de C. Sintaxa limbajului C nu este una academică, iar C nu este nici măcar un limbaj de programare de nivel înalt. Celor care doresc să înveţe un limbaj în care reguli stricte duc la formarea unei gândiri ordonate le recomand limbajele Pascal şi Ada.

De ce, totuşi, această carte? Numai din experienţa trecerii prin "labirintul" acestui limbaj putem înţelege capcanele programării şi "imperfecţiunile maşinii" pe care rulează programele noastre. Pe de altă parte, toate limbajele actuale de nivel înalt, produse de proiectare asistată sau sisteme de gestionare a bazelor de date au interfeţe cu C pentru a oferi posibilitatea scrierii unor mici rutine la nivel de "asamblare" necesare optimizării unor aplicaţii mai delicate.

Publicarea acestei cărţi se datorează prietenilor şi cunoscuţilor care în fazele de concepţie, corectură şi publicare a manuscrisului m-au sprijinit financiar şi sufleteşte. Printre cei care au avut contribuţii esenţiale îi amintesc pe: Horea Călin HOREA, Tony WILLOWS, Ioana Carmen ŢICO, Ovidiu TĂTARU, Mihai DAMIAN. Coperţile sunt lucrările artistului plastic clujean Marin LESCHIAN. "POARTA" şi "CRISALIDA" prezintă într-o formă grafică progresul, conştientizarea prezentului, trecerea de la concret la abstract, trasformarea celui care îndrăzneşte să caute eternitatea, dincolo de frumuseţea trecătoare a clipei.

Cluj, 21. I. 2001

Conf. dr. ing. ANTAL Tiberiu Alexandru
Universitatea Tenhică din Cluj-Napoca
Facultatea Construcţii de Maşini
Catedra de Mecanică şi Programare
Curs de limbaj C

ANTAL Tiberiu Alexandru
tel.: 0040-264-530918
e-mail: antaltiberiu@pcnet.ro

Conf. dr. ing. ANTAL Tiberiu Alexandru
Universitatea Tenhică din Cluj-Napoca
Facultatea Construcţii de Maşini
Catedra de Mecanică şi Programare
Curs de limbaj C

Copyright 2001. Toate drepturile sunt rezervate autorului.

Multiplicarea acestui document în scop comercial este interzisă.

Sudenţii participanţi la orice formă de învăţământ superior bugetar pot multiplica acest document pentru uzul personal.

Sudenţii participanţi la orice formă de învăţământ cu plată sau alte persoane doritoare pot multiplica documentul numai contra cost şi cu acordul scris al autorului. În acest scop mă puteţi contactă la:

ANTAL Tiberiu Alexandru
tel.: 0040-264-530918
e-mail: antaltiberiu@pcnet.ro

Conf. dr. ing. ANTAL Tiberiu Alexandru
Universitatea Tenhică din Cluj-Napoca
Facultatea Construcţii de Maşini
Catedra de Mecanică şi Programare
Părinţilor mei.
Curs de limbaj C

Conf. dr. ing. ANTAL Tiberiu Alexandru
Universitatea Tenhică din Cluj-Napoca
Facultatea Construcţii de Maşini
Catedra de Mecanică şi Programare
Curs de limbaj C

Copyright 2001. Toate drepturile sunt rezervate autorului.

Multiplicarea acestui document în scop comercial este interzisă.

Sudenţii participanţi la orice formă de învăţământ superior bugetar pot multiplica acest document pentru uzul personal.

Sudenţii participanţi la orice formă de învăţământ cu plată sau alte persoane doritoare pot multiplica documentul numai contra cost şi cu acordul scris al autorului. În acest scop mă puteţi contactă la:

ANTAL Tiberiu Alexandru
tel.: 0040-264-530918
e-mail: antaltiberiu@pcnet.ro

Universitatea Tenhică din Cluj-Napoca
Facultatea Construcţii de Maşini
Catedra de Mecanică şi Programare
Curs de limbaj C

ANTAL Tiberiu Alexandru
tel.: 0040-264-530918
e-mail: antaltiberiu@pcnet.ro

Conf. dr. ing. ANTAL Tiberiu Alexandru
Universitatea Tenhică din Cluj-Napoca
Facultatea Construcţii de Maşini
Catedra de Mecanică şi Programare
Curs de limbaj C

Copyright 2001. Toate drepturile sunt rezervate autorului.

Multiplicarea acestui document în scop comercial este interzisă.

Sudenţii participanţi la orice formă de învăţământ superior bugetar pot multiplica acest document pentru uzul personal.

Sudenţii participanţi la orice formă de învăţământ cu plată sau alte persoane doritoare pot multiplica documentul numai contra cost şi cu acordul scris al autorului. În acest scop mă puteţi contactă la:

ANTAL Tiberiu Alexandru
tel.: 0040-264-530918
e-mail: antaltiberiu@pcnet.ro

Cuprins

ANTAL Tiberiu Alexandru
tel.: 0040-264-530918
e-mail: antaltiberiu@pcnet.ro

Conf. dr. ing. ANTAL Tiberiu Alexandru
Universitatea Tenhică din Cluj-Napoca
Facultatea Construcţii de Maşini
Catedra de Mecanică şi Programare
Curs de limbaj C

ANTAL Tiberiu Alexandru
tel.: 0040-264-530918
e-mail: antaltiberiu@pcnet.ro

Universitatea Tenhică din Cluj-Napoca
Facultatea Construcţii de Maşini
Catedra de Mecanică şi Programare
Curs de limbaj C

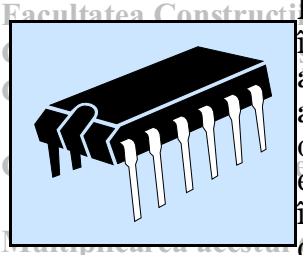
"C este un limbaj ambiguu şi uşor subcultural."
Horea Călin HOREA

1 Introducere în funcţionarea calculatoarelor

Când veţi începe a citi aceste rânduri trebuie să ştiţi că este vorba despre o călătorie în lumea mentalului, iar dispoziţia voastră trebuie să fie corespunzătoare. Pentru a învăţa un limbaj de programare sau a scrie programe într-un limbaj de programare trebuie să aveţi măcar habar de cum funcţionează calculatorul pe care-l veţi utiliza. Ceea ce urmează este o sumară introducere în lumea funcţionării sistemelor de calcul. Ar fi bine să nu vă cramponaţi în detalii şi să vă concentraţi asupra ideilor mari. Foarte probabil că nu veţi înţelege tot, dar asta nu e important; treceţi mai departe, iar dacă veţi simţi nevoia să vă delectaţi mai pe larg cu acest subiect, există cărţi suficiente despre hardware şi sisteme de operare.

1.1 Ciclul "fetch şi execute"

Un calculator este un sistem complex format din multe componente. Inima, sau mai bine zis, creierul calculatorului este **U**nitatea **C**entrală de **P**relucrare sau **UCP** (**C**entral **P**rocessing **U**nit sau **CPU**), singura care poate executa calcule împreună cu instrucţiuni şi care controlează celelalte componente ale sistemului de calcul. **U**nitatea **A**ritmetică şi **L**ogică (**A**rithmetic and **L**ogic **U**nit sau **ALU**) este o parte a UCP care poate realiza operaţii de tipul aritmetic cu întregi (adunare, scădere, înmulţire etc.), operaţii pe biţi sau booleene. Operaţiile în virgulă flotantă sunt în general realizate de o componentă separată, care poate face parte din UCP sau nu, numită "unitate pentru calcule în virgulă flotantă" (**F**loating **P**oint **U**nit sau **FPU**). O altă parte a UCP este unitatea de gestionare a memoriei (**M**emory **M**anagement **U**nit sau **MMU**) folosită pentru calcularea adreselor. Evoluţia tehnologiei şi a conceptelor de proiectare a permis ca în calculatoarele moderne UCP să fie "îngrămădit", împreună cu FPU şi MMU, pe un singur circuit integrat. Acesta are sarcina de a executa programele şi se numeşte microprocesor.



UCP sub forma unui circuit integrat

Un **`program`** este o listă de instrucţiuni care sunt executate mecanic de UCP. Instrucţiunile înţelese de UCP sunt foarte simple, iar limbajul în care se scriu poartă denumirea de **`limbaj maşină`**. Fiecare calculator are propriul limbaj maşină şi poate executa programe numai dacă acestea au fost scrise în propriul lui limbaj maşină. Un program scris într-un limbaj maşină diferit se poate executa numai după traducerea lui în cel cunoscut de UCP pe care se va executa.

Când UCP execută un program, programul trebuie să fie stocat în **`memoria principală`** a sistemului de calcul, adesea numită şi **RAM** (**R**andom **A**ccess **M**emory). Pe lângă

instrucţiuni, programul va conţine şi date care vor fi prelucrate de programul în sine. Memoria principală este formată dintr-o secvenţă de **locaţii**. Aceste locaţii sunt numerotate, iar numărul de ordine al fiecărei locaţii este unic şi se numeşte **adresă**. Adresa este o modalitate de a stoca sau a extrage o informaţie particulară dintre miliardele de informaţii stocate în memorie. Când trebuie să acceseze o informaţie particulară, cum ar fi o instrucţiune sau o dată, UCP va transmite memoriei principlale adresa la care se află informaţia dorită sub forma unor semnale electrice. Memoria va răspunde prin trimiterea înapoi a datei stocate spre UCP la adresa primită anterior de la acesta. UCP poate stoca şi date în memoria principală prin specificarea simultană a adresei la care se va face stocarea şi a datei de stocat.

Deşi la nivelul limbajului maşină operaţiile care pot fi executate de UCP sunt elementare şi simple (adunare, scădere, comparaţie, etc.), intern UCP va funcţiona după o schemă foarte complexă pentru cazul execuţiei anumitor operaţii. Instrucţiunile sunt stocate în memoria principală sub forma unei secvenţe de instrucţiuni în limbajul maşinii de unde sunt în mod ciclic extrase şi executate. Acest proces repetat de citire şi execuţie a instrucţiunilor se numeşte "fetch-and-execute cycle" adică ciclu de extragere şi execuţie a instrucţiunilor. Cu o singură excepţie care va fi lămurită mai târziu, aceasta este tot ceea ce face UCP.

Detaliile ciclului de extracţie şi execuţie sunt de asemenea caracteristice fiecărui UCP în parte, dar există şi caracteristici comune. Orice UCP conţine **registri interni** care sunt zone de memorie de dimensiune mică, ce pot stoca un singur număr sau o instrucţiune maşină. O parte din regiştrii UCP au utilizări speciale. De exemplu, registrul numărător de program (**P**rogram **C**ounter sau **PC**) va stoca locul din memorie la care se află UCP în timpul execuţiei unui program. PC-ul va stoca adresa următoarei instrucţiuni din program care urmează a fi executată. La începutul fiecărui ciclu de extragere-execuţie UCP va folosi conţinutul registrului PC pentru a afla care este următoarea instrucţiune de extras. În timplu ciclului de extracţie şi execuţie, numărul stocat în PC este actualizat pentru a stoca adresa instrucţiunii care va fi executată în următorul ciclu. (În mare parte din cazuri aceasta se află chiar la adresa următoare).

UCP este format din milioane de tranzistori care sunt interconectaţi pe o singură aşchie de siliciu (numită "chip" în engleză). Tranzistorii sunt puşi în circuite care au două stări electrice distincte şi stabile de pornit şi oprit, între care pot comuta, motiv pentru care aceste stări se pot pune în corespondenţă cu valorile binare de 1 şi 0. Pe măsură ce UCP calculează, stările tranzistorilor se modifică, rezultând un spaţiu de stări determinat de modul de conectare a tranzistorilor şi de instrucţiunea din program care se execută.

Instrucţiunile limbajului maşină se prezintă sub forma unor numere binare (secvenţe de 1 şi 0). Fiecare instrucţiune are o secvenţă particulară de 0 şi 1. Datele pe care UCP le manipulează sunt şi ele codificate sub formă binară. UCP poate lucra numai cu numere binare. Astfel, un tranzistor deschis corespunde lui 1, iar unul închis lui 0. Instrucţiunile limbajului maşină reprezintă secvenţe de 0 şi 1 stocate la adrese consecutive în memorie. Când o instrucţiune se încarcă în UCP secvenţa binară din memorie se aduce în UCP, iar acesta reacţionează în mod mecanic la instrucţiune. Reacţia UCP este determinată de modul în care acesta este microprogramat, adică de modul în care sunt interconectaţi tranzistorii din interiorul lui. În concluzie, memoria principală stochează datele şi instrucţiunile sau, într-un cuvânt, programul codificat în binar. UCP aduce din memorie şi apoi execută fiecare instrucţiune a programului în mod ciclic. UCP poate executa numai un program perfect în toate detaliile şi neambiguu.

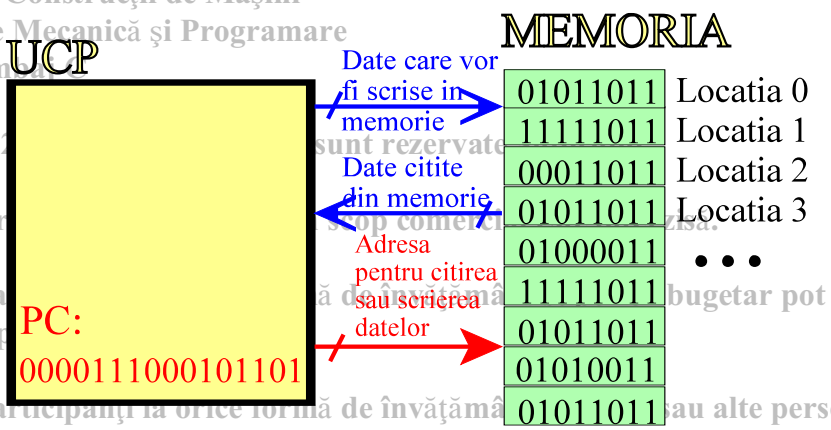


Figura 1 - Schema necesară înţelegerii ciclului de extragere şi execuţie a instrucţiunilor

În *Figura 1* se prezintă o schemă foarte simplă pentru înţelegerea funcţionării unui calculator. Memoria nu stochează la o adresă un singur bit, ci grupuri de câte 8 biţi care poartă numele de **octet** (**byte**). Din acest motiv, unei linii de date sau de adrese îi corespunde în realitate un grup de linii de date sau adrese. În figura de mai sus se observă o singură linie pentru adrese şi două pentru date. Aceste linii sunt însă intersectate de câte una oblică, ea simbolizând prezenţa în realitate a unui grup de linii care au aceeaşi semnificaţie.

1.2 Evenimente asincrone

Un sistem de calcul este format din mult mai multe componente decât UCP şi memoria principală, câteva din acestea sunt prezentate în continuare. UCP este fixat pe un circuit imprimat numit placă de bază, care în plus conţine magistrale, memorii RAM, ROM şi alte circuite integrate:

- **`hard disc-ul`** pentru stocarea programelor şi a fişierelor. Memoria principală stochează o cantitate mică de date şi numai atât timp cât sistemul de calcul este pus sub tensiune. Hard disc-ul este o memorie magnetică pentru stocarea permanentă (al cărei conţinut nu se pierde la scoaterea de sub tensiune a sistemului de calcul) a unor cantităţi mari de informaţii, dar programele trebuie încărcate de pe disc în memoria principală înainte de a putea fi executate;

- **`tastatura`** şi **`mouse-ul`** pentru ca utilizatorul să poată introduce date;

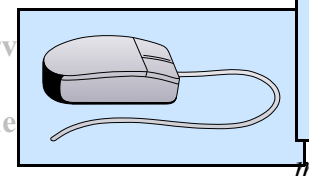
- **`monitor-ul`** şi **`imprimanta`** care se folosesc pentru afişarea ieşirilor produse de sistemul de calcul;

- **`placa de reţea`** care permite ca sistemul de calcul să comunice cu alte sisteme de calcul conectate şi ele la reţea;

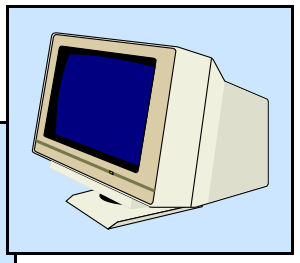
- **`scanner-ul`** care converteşte imagini în numere binare care vor putea fi stocate şi manipulate prin sistemul de calcul.

hard disc

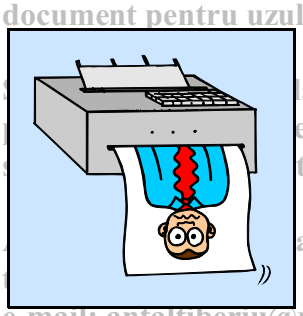
tastatură



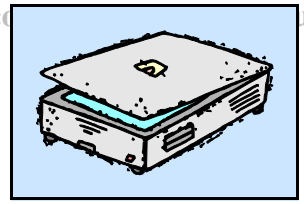
mouse



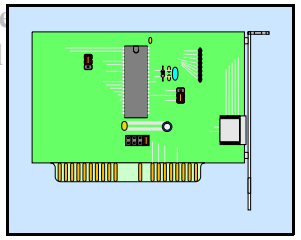
monitor



imprimantă

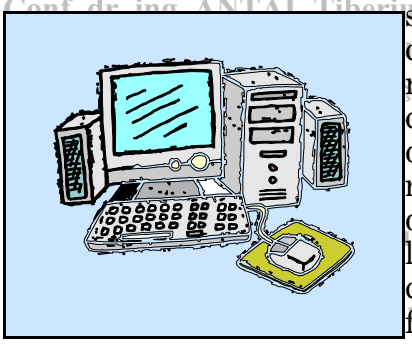


scanner



placă de reţea

Lista acestor componente şi dispozitive rămâne deschisă, sistemele de calcul fiind astfel construite încât să poată fi extinse prin adăugarea de noi dispozitive sub forma unor **plăci de extensie** (**expansion cards**) care vin introduse în **locaşurile de extensie** (**expansion slots**) ale plăcii de bază. UCP însă trebuie să comunice cu aceste dispozitive pentru a schimba informaţii cu ele, iar pentru că singurul limbaj cunoscut de acesta este cel maşină, fiecare dispozitiv are nevoie de un **driver de dispozitiv**. Acesta este un program care dacă este executat de UCP ştie să deruleze transferul de informaţii între UCP şi dispozitivul în cauză. Din acest motiv instalarea unui dispozitiv nou în sistemul de calcul se derulează în două etape: se conectează fizic noul dispozitiv la sistemul de calcul, apoi se instalează programul driver corespunzător acestuia. Există dispozitive atât de necesare funcţionării sistemului de calcul încât drivere-ele corespuzătoare lor sunt stocate în memorii **ROM** (**R**ead **O**nly **M**emory) care sunt furnizate implicit cu orice sistem de calcul. Acesta este cazul fericit în care nu avem nevoie de driver pentru a face ca dispozitivul să conlucreze cu sistemul de calcul. Pentru că aceste dispozitive vor schimba informaţii cu UCP şi cu memoria principală, ele trebuie să fie interconectate cumva cu acestea. Se numeşte **magistrală** (**bus** în engleză) un grup de linii electrice care transportă informaţii între dispozitivele interconectate ale sistemului de calcul. În funcţie de tipul de informaţii transportate avem **bus de date**, **bus de adrese** şi **bus de semnale de control**. Pe busul de adrese se vehiculează informaţia de selecţie. O adresă va direcţiona o dată particulară de pe busul de date către un anumit dispozitiv sau registru de dispozitiv. Pe busul de control se vehiculează semnalele necesare comunicaţiei între dispozitive. Un dispozitiv va folosi acest bus pentru a semnaliza altuia că datele aşteptate sunt disponibile şi valide pe busul de date. Cea mai simplă schemă de calculator este prezentată în *Figura 2*. Dispozitive cum sunt tastatura, mouse-ul şi placa de reţea pot produce date care trebuie prelucrate de UCP. De unde va şti UCP că datele sunt acolo? Cel mai simplu ar fi ca UCP să verifice în mod continuu prezenţa datelor, după care să le prelucreze. Această metodă



un sistem de calcul

de lucru se numeşte în limba engleză "**`polling`**". Este simplă dar ineficientă pentru că UCP pierde prea mult timp în aşteptarea datelor, motiv pentru care în practică este rar utilizată. Pentru evitarea acestei situaţii ineficiente de multe ori se vor utiliza **`întreruperi`**. O întrerupere este un semnal trimis către UCP de un dispozitiv diferit de UCP. UCP reacţionează la un semnal de întrerupere suspendându-şi activitatea curentă şi răspunzând întreruperii. După ce a terminat tratarea întreruperii, UCP va reveni la activitatea anterioară, continuând-o din punctul în care a întrerupt-o. De exemplu, la apăsarea unei taste se generează o întrerupere, UCP îşi întrerupe activitatea curentă, citeşte tasta apăsată, o prelucrează, apoi revine la operaţia pe care o executa înainte de apăsarea tastei.

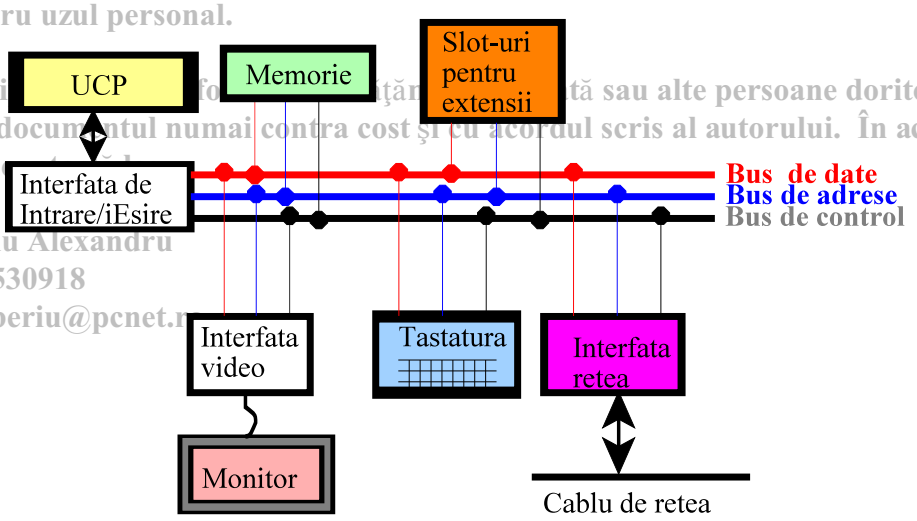


Figura 2 - Cea mai simplă schemă de calculator

Procesul descris mai sus este mecanic. Semnalul de întrerupere apare pe un pin al UCP care este construit aşa încât dacă apare acest semnal îşi salvează starea curentă pentru ca ulterior să poată reveni la ea şi să o continue. Salvarea constă în stocarea conţinutului unor registri mai importanţi, cum ar fi PC-ul. Apoi UCP sare la o adresă de memorie fixă şi începe să execute instrucţiunile de acolo. Aceste instrucţiuni alcătuiesc o rutină de tratare a întreruperii care face toate prelucrările necesare pentru a răspunde la întrerupere. De obicei, rutina de întrerupere este o parte a unui program driver a dispozitivului care a transmis întreruperea. La terminarea rutinei de tratare a întreruperii există o instrucţiune care forţează UCP să revină la operaţia pe care o efectua la momentul în care a sosit întreruperea prin refacerea stării lui.

Întreruperile permit UCP să trateze evenimente asincrone. În ciclul obişnuit de extracţie-execuţie totul se petrece într-o ordine strictă şi predeterminată. Tot ceea ce se petrece face parte dintr-un şablon al unui grup de evenimente sincronizate. Din cauza întreruperilor UCP lucrează eficient cu evenimente asincrone, adică evenimente care apar la momente imprevizibile.

Toate acestea sunt valabile numai dacă UCP are mai multe sarcini de executat. Calculatoarele moderne folosesc **`multitasking`** pentru a executa mai multe **`task`**-uri (sarcini) în acelaşi timp. Unele sisteme de calcul sunt atât de rapide încât pot fi folosite de mai mulţi utilizatori în acelaşi timp. Din moment ce UCP este atât de rapid, el poate să-şi comute atenţia de la un utilizator la altul, alocând pentru fiecare o fracţiune din timpul lui de lucru. Acest mod de aplicare al multitasking-ului se numeşte **`cu timesharing`** (partajare

în timp). Chiar şi banalele calculatoare personale folosite de un singur utilizator sunt capabile de multitasking. De exemplu, poţi scrie un document într-un editor de text în timp ce pe ecran, undeva într-un colţ, îţi apare ceasul care afişează ora exactă în mod continuu. Fiecare din aceste sarcini individuale executate de UCP se numesc **procese** (sau thread-uri; există diferenţe tehnice între proces şi thread dar pentru explicaţiile care urmează nu sunt importante). Un singur proces poate fi executat la un moment dat de UCP (există sisteme de calcul care au mai multe UCP, iar atunci se poate executa câte un proces pe fiecare UCP). UCP va continua să execute un proces până când:

- procesul, în mod voluntar, va preda controlul UCP altor procese pentru ca şi acestea să poate fi executate;
- procesul poate aştepta apariţia unui eveniment asincron. De exemplu, procesul poate cere date de la hard disc sau poate aştepta ca utilizatorul să apese o tastă. În timpul aşteptării procesul este blocat şi alte procese au şansa să fie executate. Când evenimentul aşteptat apare, o întrerupere va trezi procesul şi acesta îşi va continua execuţia;
- procesul şi-a folosit durata de timp alocată execuţiei şi va fi suspendat pentru ca şi alte procese să poate fi rulate. Nu toate calculatoarele pot să suspende astfel un proces; acelea care pot se spune că utilizează "preemtive multitasking". Pentru a realiza preemtive multitasking un sistem de calcul trebuie să aibă un circuit de ceas special care să genereze întreruperi la intervale regulate de timp. La apariţia unei întreruperi de timp, UCP are şansa să comute pe un alt proces indiferent dacă procesului actual "îi convine" aceasta sau nu.

Programul care realizează tratarea întreruperilor şi toată comunicaţia cu utilizatorul şi dispozitivele din sistemul de calcul se numeşte **sistem de operare**. Sistemul de operare este un program de bază fără de care un calculator nu poate funcţiona. Toate celelalte programe care se execută pe sistemul de calcul sunt dependente de sistemul de operare. Câteva din sistemele de operare mai cunoscute sunt OS/2 Warp, UNIX, DOS, Windows 98, Windows NT, Windows 2000 şi Macintosh OS.

1.3 Algoritm, date şi program

Algoritmul reprezintă o secvenţă detaliată de acţiuni executate pentru îndeplinirea unor sarcini. Teoretic, un algoritm trebuie să ajungă la rezultat după un număr finit de paşi. Denumirea de algoritm a fost dată în onoarea matematicianului iranian Al-Khawarizmi. Pentru un calculator, **datele** se prezintă sub forma unor numere, caractere, imagini etc. obţinute printr-o metodă oarecare de înregistrare, cu scopul de a fi introduse în sistemul de calcul, stocate şi prelucrate sau transmise la distanţă. Datele în sine nu au o semnificaţie. Numai în urma interpretării lor cu ajutorul unui sistem de prelucrare a datelor, acestea primesc o semnificaţie, devenind informaţie. Activitatea de programare presupune trecerea de la universul problemei reale la cel al sistemului de calcul, mult mai îngust. Universul real al unei probleme fiind de o complexitate prea mare este necesară trecerea prin limitare şi abstractizare la cel specific sistemului de prelucrare a datelor. Aşa a apărut noţiunea de model, o descriere a realităţii, simplificată prin ignorarea anumitor detalii. Prin folosirea unui model în locul problemei reale şi prin abstractizare se ajunge la simplificarea prelucrării realităţii. Rolul abstractizării este formularea unui algoritm de transformare a unor date. Algoritmul specifică acţiunile de prelucrare, iar datele descriu universul abstractizat. Pentru a putea fi prelucrate pe un sistem de calcul, algoritmul trebuie scris într-un limbaj de programare, rezultatul acestei activităţi numindu-se **program**. În cadrul limbajului de programare, **datele** sunt specificate prin intermediul **declaraţiilor**, iar **algoritmul** prin **instrucţiuni**.

1.4 Despre limbajele de programare

Limbajul de programare este un mijloc de comunicare între programator şi calculator. Un limbaj se numeşte **limbaj natural** dacă are capacitatea de a evolua, fiind rezultat în urma dezvoltării lui în timp. Un **limbaj artificial**, cum sunt şi limbajele de programare, nu evoluează în timp de la sine. Evident şi limbajele de programare evoluează, ele fiind un mijloc de comunicaţie între om şi maşină, dar aceasta numai din cauza oamenilor care le extind sau îmbunătăţesc pentru a fi cât mai utile domeniului în care se aplică. Descrierea unui limbaj se face prin **semantică** şi **gramatică**. La rândul ei, gramatica se împarte în **morfologie** şi **sintaxă**. Semantica studiază legăturile dintre cuvintele din dicţionarul limbajului şi obiectele descrise de acestea, adică înţelesul cuvintelor. Morfologia studiază modificările suferite de cuvinte pentru a realiza nuanţarea înţelesurilor. În limbajele de programare nu se folosesc modificări morfologice.

Un **program** este o secvenţă de instrucţiuni pe care un sistem de calcul le execută pentru a îndeplini o sarcină. Pentru ca programul să poată fi executat, el trebuie să fie scris într-un **limbaj de programare**. Limbajele de programare diferă de cele umane fiind complet neambigue şi foarte stricte cu privire la ce este şi ce nu este permis în program. Regulile care determină ce este permis construiesc **sintaxa** limbajului. Regulile sintactice specifică vocabularul de bază al limbajului şi modul în care se pot construi programe utilizând cicluri, ramificaţii sau subprograme. Un program corect sintactic este unul care nu are erori în urma compilării (compilarea este operaţia de traducere a unui program scris într-un limbaj de nivel înalt în limbajul maşină). Dacă apar erori programul trebuie corectat şi apoi, din nou, compilat. Deci, pentru a avea succes în programare trebuie să învăţaţi sintaxa limbajului de programare utilizat. Totuşi, sintaxa este numai o parte din poveste. Nu este suficient să scrii programe care se compilează fără erori. Dorim ca programul în urma execuţiei să producă rezultate corecte. Asta înseamnă că înţelesul programului trebuie să fie corect. Înţelesul programului se numeşte **semantică**, iar un program corect semantic face ceea ce doreşte programatorul.

Sintaxa unui limbaj de programare se memorează, după care treceţi la semantică şi încercaţi să înţelegeţi cum lucrează anumite instrucţiuni ale limbajului.

1.4.1 Gramatica formală

Pentru că maşina de calcul este numai un suport fără cel mai mic gram de inteligenţă s-a dezvoltat o teorie a limbajelor formale care, printre altele, descrie noţiuni de gramatică necesare înţelegerii aspectelor sintactice ale limbajelor de programare. În viziunea lui N. Chomsky, creatorul teoriei limbajelor formale, gramatica **G = G(T, N, P, S)** unui limbaj **L(G)** este specificată prin:

a) un vocabular **T** de simboluri terminale, numite şi **cuvinte**;
b) o mulţime **N** de simboluri neterminale, numite şi **categorii gramaticale**;
c) o mulţime **P** de producţii numite şi **reguli sintactice**;
d) un simbol **S** din **N**, numit **simbol de start**.

Limbajul **L(G)** este mulţimea cuvintelor (adică a şirurilor de simboluri terminale) care se pot obţine din simbolul **S** prin intermediul regulilor de producţie. Fie limbajul **L** definit prin următoarea gramatică:

T={a, b, c}; N={S, A, B}; P={S → AB, A → a, A → ac, B → b, B → cb}

Semnul → are semnificaţia "trece prin substituţie în". În limbajul de mai sus se pot genera

următoarele producţii de cuvinte:

S → AB → aB → ab
S → AB → aB → acb
S → AB → acB → acb
S → AB → acB → accb

Se poate arăta că în limbajul de mai sus nu există mai multe cuvinte şi că schimbarea ordinii substituţiilor conduce la aceleaşi cuvinte, adică **S → AB → Ab → ab**. Se observă că **abc** este un cuvânt care se poate obţine în două moduri, din acest motiv el va avea două semnificaţii după cum litera c se obţine din substituţia **A → ac** sau **B → cb**. Expresii cu mai multe semnificaţii există şi în limbile vorbite. Fie propoziţia: "Aceasta este masa". Ea poate să semnifice că în faţa noastră stă obiectul pe care ne luăm masa sau că avem în faţa noastră un cablu care este masă electrică a unui dispozitiv electronic. Dacă nu se cunoaşte contextul, adică propoziţiile din care rezultă cea actuală, semnificaţia este neprecizată.

Există şi alte metode pentru a descrie gramatica limbajelor formale, cea mai frecvent utilizată fiind cea a lor I. Backus şi P. Naur. Aici se folosesc metasimbolurile de tip paranteze: **< >** şi metasimbolul: **|**, pentru conjuncţia "fie". Simbolul → este înlocuit prin **::=**. Parantezele se folosesc pentru a descrie modul de înlocuire a simbolurilor prin cuvinte sau fraze care descriu semnificaţiile simbolurilor. Rezultatul este obţinerea unei mulţimi de definiţii. Regula **S → AB** se poate obţine din definiţia:

<propoziţie>::= <subiect><predicat>
<subiect>::= oamenii | câinii
<predicat>::= dorm | mănâncă

Semnificaţia liniilor de mai sus este:
1. O **<propoziţie>** se defineşte ca fiind formată dintr-un **<subiect>** urmat de un **<predicat>**.
2. Un **<subiect>** se defineşte ca fiind fie cuvântul "**oamenii**" fie cuvântul "**câinii**".
3. Un **<predicat>** se defineşte ca fiind fie cuvântul "**dorm**" fie cuvântul "**mănâncă**".

Ideea de bază este că o **<propoziţie>** poate fi derivată dintr-un simbol de start prin aplicarea unor reguli de înlocuire.

1.4.2 Scheme sintactice

Regulile de producţie ale unei gramatici pot fi transcrise şi sub forma unor desene care se numesc **scheme sintactice**. Fiind date regulile de producţie, prin eliminarea simbolurilor neterminale, după mai multe faze, se obţin numai simboluri terminale. Convenind că simbolurile neterminale se scriu cu majuscule - mai jos **A** şi **B** - iar cele terminale cu litere mici - mai jos **x**, **y**, **z** şi **w**, pentru regulile următoare scrise folosind formalismul Backus-Naur:

S::= A B
A::= x | y
B::= z | w

Gramatica transcrisă folosind scheme sintactice se prezintă în *Figura 3*. În funcţie de limbajul de programare abordat, unii autori preferă să

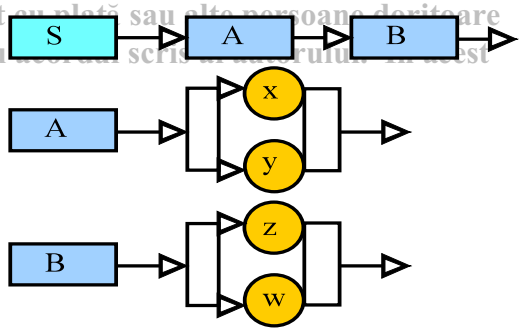


*Figura 3 - Modul de specificare a gramaticii prin diagrame de sintax*ă

folosească în specificaţiile tehnice ale limbajului, diagrame de sintaxă, iar alţii, **F**ormalismul **B**ackus-**N**aur (**FBN**). Diagramele de sintaxă sunt lizibile, însă necesită mult spaţiu. Din acest motiv producătorii de limbaje de programare preferă să folosească FBN, în care descrierea este mai scurtă, dar şi mai greu de înţeles.

După explorarea mai multor metode formale pentru definiţia limbajului, comitetul pentru standardizarea limbajului C a decis să folosească FBN pentru sintaxa limbajului şi proza pentru constrângeri şi semantică. Orice încercare mai ambiţioasă s-a considerat că:

- va duce la întârzierea apariţiei standardului;
- va face limbajul mai puţin accesibil utilizatorilor.

În standardul C termenii "comportament nespecificat", "nedefinit" şi "definit prin implementare" sunt folosiţi pentru a categoriza rezultatele produse de programe ale căror proprietăţi nu sunt descrise sau nu pot fi descrise complet în standard. Scopul acestor termeni a fost crearea unei oarecare varietăţi în rândul implementărilor de C în vederea categorizării lor după calitate, existând implementări de C mai populare care deviază uneori de la standard. Comportamentul nespecificat lasă la latitudinea implementatorului limbajului traducerea unor secvenţe de program, fără a permite însă ca traducerea programului să nu reuşească din acest motiv. Comportamentul nedefinit permite ca implementatorul să nu prindă anumite erori de program care sunt dificil de diagnosticat. De asemenea, acoperă problematica anumitor extensii ale limbajului prin furnizarea de definiţii în unele cazuri de comportamente nedefinite. Comportamentul definit prin implementare lasă implementatorului libertatea alegerii unei rezolvări proprii, însă aceasta trebuie să fie explicată utilizatorului. Pe baza acestui comportament utilizatorul trebuie să poată lua decizii semnificative referitoare la scrierea codului.

2 Scurtă istorie şi descriere a elementelor limbajului C

Istoria limbajului C începe la firma Bell Labs care a iniţiat dezvoltarea lui la începutul anilor '70 pentru scrierea sistemului de operare UNIX utilizat de către noile calculatoare - pentru vremea de atunci - **DEC** (**D**igital **E**quipment **C**orporation). Până atunci sistemele de operare erau scrise în limbaj de asamblare. Acesta era obositor, greu de întreţinut şi consumator de timp. Cei de la Bell Labs au înţeles că era nevoie de un nou limbaj de programare de nivel înalt, cât mai rapid şi cât mai uşor de întreţinut, care să permită implementarea proiectelor din firmă. Întrucât limbajele de programare de nivel înalt existente pe atunci (COBOL, FORTRAN, PL/I, Algol) erau prea încete pentru codul sistemelor de operare, programatorii de la Bell Labs au decis să scrie propriul limbaj de programare. Noul limbaj s-a bazat pe limbajul B al lui Ken Thompson care a fost inspirat de **BCPL** (**B**asic **C**ombined **P**rogramming **L**anguage) scris de Martin Richards, care era o versiune simplificată a limbajului **CPL** (**C**ambridge **P**rogramming **L**anguage). Pentru că B avea câteva restricţii, în 1972, **Brian Kernighan** şi **Dennis Ritchie** de la **AT&T Bell Labs**, au creat limbajul C ca o extensie a limbajului B.

Se spune că totul a pornit de la un joc cu "asteroizi" pe care îl jucau în timpul serviciului folosind calculatorul central al companiei. Performanţele acestuia lăsau de dorit. Cu puterea de calcul al unui 386 şi deservind aproximativ 100 de utilizatori, au ajuns la concluzia că nu pot controla suficient de bine "navele spaţiale", motiv pentru care erau distruşi repede la trecerea printre asteroizi. Având în birou un DEC PDP-7 care nu era folosit, au hotărât să rescrie jocul. Problema era că DEC nu avea sistem de operare, aşa că au început să scrie unul. În scurt timp scrierea sistemului de operare a devenit un proiect mai important decât jocul de la care a pornit totul. El fiind scris în limbaj de asamblare, un limbaj maşină umanizat, în care şirurile binare de 0 şi 1 corespunzătoare diferitelor instrucţiuni ale limbajului maşină s-au înlocuit prin grupuri de litere şi cifre - înainta mult prea greu. Atunci s-au hotărât să rescrie sistemul de operare într-un limbaj de nivel înalt care să fie portabil pe diferite tipuri de sisteme de calcul. În acest caz, tot ceea ce ar fi fost necesar era reimplementarea compilatorului pentru fiecare sistem de calcul nou, după care sistemul de operare se putea compila. Limbajul care a fost ales în acest scop s-a numit B dar el nu permitea utilizarea tuturor avantajelor oferite de instrucţiunile lui PDP-11. Astfel a fost inventat un nou limbaj, C.

C nu a devenit popular imediat după crearea lui. Am putea spune că numai după 6 ani de la inventarea lui, în 1978, s-a auzit de el prin memorabila carte a lui Brian Kernighan şi Dennis Ritchie - THE C PROGRAMMING LANGUAGE - care a schimbat totul. Din acest moment C a fost implementat pe calculatoare pe 8 biţi sub sistemul de operare CP/M. În 1981 când a început revoluţia PC prin calculatoarele IBM PC, C era unicul limbaj care avea puterea să le exploateze total. Astfel a deviat de la sistemul de operare UNIX sub care a fost scris iniţial şi a devenit un limbaj popular pe toate

microcalculatoarele.

Programatorii adorau limbajul C pentru că, spre deosebire de alte limbaje, el permite programatorului un control absolut asupra calculatorului. Cu acest control vine şi o responsabilitate - sunt multe posibilităţi în C pentru a bloca un calculator - însă programatorii au găsit în sfârşit un limbaj care este o unealtă şi nu un obstacol în programarea calculatoarelor.

2.1 Caracteristicile limbajului C

Limbaj de asamblare avansat

Programatorii care ajung să lucreze în C venind din Basic sau Pascal sunt surprinşi de "nivelul de jos" al programării în C. Face puţine lucruri automat, însă permite scrierea lor în amănunt, cu mâna, dacă programatorul doreşte acest lucru. Nu are instrucţiuni care lucrează direct cu şiruri de caractere, mulţimi, liste. Nu există operatori care să manipuleze un tablou sau un şir întreg. Nu are facilităţi de intrare/ieşire şi nici nu permite accesul la fişiere. Toate aceste caracteristici ale limbajelor de nivel înalt sunt explicit asigurate prin apelul unor funcţii care nu sunt parte a limbajului şi sunt stocate sub forma unor biblioteci de funcţii standard. Autorii lui îl numesc un limbaj de nivel relativ jos, în sensul că este destul de aproape de limbajul maşină; practic l-am putea numi un limbaj de asamblare cu câteva caracteristici de limbaj de nivel înalt.

Viteză mare

Motivul pentru care C a fost inventat este viteza. Programele scrise în C se execută foarte rapid, deşi dezvoltarea lor este de multe ori înceată. Pentru creşterea vitezei, C a renunţat la majoritatea verificărilor făcute în general de limbajele de programare de nivel înalt cu scopul creşterii siguranţei funcţionării programului.

Portabilitate

Am spus că limbajul maşină este specific fiecărui UCP. Pentru că limbajul de asamblare nu este decât un limbaj maşină cu o formă mai elegantă, înseamnă că şi acesta este specific fiecărui UCP, iar programele scrise în el nu pot fi portate pe maşini diferite, adică nu este portabil. C fiind un limbaj de asamblare în multe feluri, înseamnă că nici programele C nu sunt portabile. Totul depinde de modul în care s-a scris C. Respectând câteva reguli de bază este posibil ca programele C să fie portate la fel cu cele scrise în Basic sau Pascal. Am spus că C pune la dipoziţia programatorului biblioteci. Această modalitate de definire ajută la izolarea limbajului de trăsăturile specifice ale UCP şi construcţiei hardware a maşinii pe care se rulează o implementare particulară de C, rezultând posibilitatea scrierii unui cod portabil.

Pentru scrierea sistemelor de operare

C s-a născut din încercarea de creare a unui limbaj de programare pentru scrierea sistemelor de opeare. Primul sistem de operare scris în C a fost UNIX şi a avut aproximativ 10000 de linii. Azi atât UNIX cât şi Windows au toate specificaţiile necesare pentru programarea lor descrisă în limbajul C.

Lizibilitate grea

Sintaxa C este departe de a fi sugestivă. Din această cauză se poate ajunge la situaţia unor programe care nu mai pot fi înţelese, deşi sunt scrise în C şi funcţionează corect.

2.2 Standarde C

La fel de eficient ca limbajul de asamblare, dar mai prietenos, cu multe instrucţiuni apropiate de echivalentul lor din limbajul de asamblare şi mai puternic prin noile instrucţiuni adăugate în anii '80, C a devenit cel mai popular limbaj având sute de variante în implementare şi o comunitate de programatori care creşte rapid. Revoluţia C era în pericol riscând să divizeze limbajul în prea multe variante incompatibile între ele; era vremea ca limbajul să fie standardizat.

În America, responsabilitatea standardizărilor revine Institutului Naţional American de Standarde (**A**merican **N**ational **S**tandards **I**nstitute sau, mai pe scurt, **ANSI**). Comitetul de autorizare ANSI pentru C s-a numit X3J11, iar limbajul este azi definit prin standardul ANSI X3.159-1989.

Aria internaţională a standardizărilor este acoperită de Organizaţia Internaţională de Standardizare (**I**nternational **S**tandards **O**rganization sau **ISO**). ISO a format comitetul JTC1/SC22/WG14 pentru revizuirea muncii lui X3J11. Azi, standardul ISO pentru C este ISO 9889:1990 şi este în esenţă identic cu X3.159.

Standardul ISO C este autoritatea finală prin care s-a reconstruit limbajul C. De obicei, în documetaţii, el este referit sub forma "**standardul**", la fel cum limbajul C definit în cartea lui Kernighan şi Ritchie este cunoscut sub numele de "K&R C".

2.3 Ce a rezolvat standardizarea

Limbajul C a beneficiat enorm de pe urma standardizării, ca urmare devenind un limbaj mult mai uşor de utilizat. În K&R C nu exista un mecanism pentru verificarea parametrilor transferaţi funcţiilor. Nu se verificau nici numărul, nici tipul parametrilor transferaţi. Din acest motiv întrega responsabilitate cădea pe umerii programatorului care trebuia să fie atent la asemenea erori în apelul funcţiilor. Problema era aşa de serioasă încât s-a scris un utilitar special, `lint`, în acest scop.

Calculele în virgulă flotantă erau o glumă în limbajul K&R C. Toate acestea se făceau folosind tipul de dată numit `double`, deşi exista un tip de dată numit `float` pentru lucrul cu valori mai mici. Valorile de tipul `float` fiind mai mici era normal ca toate calculele să fie efectuate mai rapid, dar datorită conversiei lor la `double`, timpul de calcul creştea.

Deşi a existat o `bibliotecă standard` sub forma unei colecţii de subprograme care însoţeau limbajul C, nu era nimic standardizat cu privire la elementele pe care această bibliotecă le conţinea. Acelaşi subprogram putea să aibă denumiri diferite sau să lucreze în mod diferit.

Odată cu rezolvarea acestor probleme, C standard a devenit mult mai robust şi uşor de utilizat, motiv pentru care voi aborda descrierea acestui C şi nu a celui descris de K&R în prima lor carte.

2.4 Elemente de C

Acest paragraf descrie elementele limbajului C: nume, numere, caractere folosite la scrierea unui program C. Sintaxa ANSI C numeşte aceste componente simboluri, ele fiind

elementele de bază recunoscute de compilator. Un simbol este un text al programului sursă care nu poate fi descompus în alte elemente componente. Următoarea listă cuprinde tipurile de simboluri existente în limbajul C: **cuvânt cheie**, **identificator**, **constantă**, **şir**, **operator**, **caracter de punctuaţie**.

- **cuvintele cheie**: au o semnificaţie predefinită pentru compilatorul C, motiv pentru care un identificator nu poate avea acelaşi nume - scriere sintactică - cu un cuvânt cheie. Cuvintele cheie ale limbajului C sunt:

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

NOTA: Toate cuvintele cheie ale limbajului C se scriu cu litere mici.

- **identificatorul**: este un nume dat de utilizator unei variabile, tip de dată, funcţie sau unei etichete din program. Crearea unui identificator se face prin specificarea lui într-o declaraţie de variabilă, tip sau funcţie, fiind caracterizat prin *tip*, *vizibilitate*, *durată de existenţă (persistenţă)* şi *legare*. După declarare, un identificator poate fi utilizat în program ţinând locul valorii asociate lui;
- **constanta**: reprezintă un număr, un caracter sau şiruri de caractere care sunt folosite ca o unică valoare în program şi sunt caracterizate prin *valoare* şi *tip*. C cunoaste constante reale, întregi, enumerate şi caracter;
- **şirul**: reprezintă o secvenţă de caractere cuprinsă între simbolurile " ", de exemplu "un sir de caractere";
- **operatorul**: este un simbol care specifică modul în care sunt manipulate valorile asupra cărora acţionează. Poate fi format dintr-un singur caracter sau din mai multe;
- **caractere de punctuaţie**: sunt caractere speciale care au utilizări diverse, de la organizarea programului sursă până la definirea unor operaţii pe care le execută compilatorul sau programul compilat.

2.5 Primul program

Cel mai scurt program care se poate scrie în C este:

```
main() { }
```

deşi este un program funcţional nu va produce un rezultat vizibil pe ecran.

Lansarea în execuţie a unui program sub sistemul de operare DOS are ca efect transferul controlului programului lansat în execuţie. În acest scop DOS trebuie să ştie unde să intre

în program pentru a începe execuţia lui. **main** înseamnă principal, reprezentând acest punct de început din care DOS începe să execute instrucţiunile programului. Numele se scrie urmat de paranteze rotunde, adică **main()**, iar corpul programului este închis între acolade. De obicei, între acolade se află un număr oarecare de instrucţiuni:

```
1  /* SALUT.C */
2  #include<stdio.h>
3
4  /* Afiseaza "Salutare tinerete." pe ecran */
5
6  main()
7  {
8          printf("Salutare, tinerete!");
9
10         return(0);
11 }
```

Efectul acestui program este afişarea textului "**Salutare, tinerete!**" pe ecran. Se observă că acoladele sunt scrise una sub cealaltă, iar liniile de program **printf("Salutare, tinerete!");** **return(0);** sunt aliniate.

Fiecare linie de program este numerotată pentru a simplifica explicaţiile. Aceste linii NU FAC PARTE DIN PROGRAMUL C (din acest motiv ele NU vor fi scrise la tastarea acestui program şi nici la cele care vor urma).

Acest mod de scriere nu este necesar, dar mulţi programatori îl folosesc, el făcând parte din stilul de scriere al programelor C.

Pentru a face **programul executabil**, prima oară trebuie să-l stocăm pe disc într-un fişier text, să zicem cu numele **salut.c**. Apoi vom avea nevoie de un program compilator C pentru a crea fişierul executabil. Compilatorul este responsabil pentru traducerea programului din limbajul C în limbajul maşină. Programul compilator va produce din **salut.c** un nou fişier cu numele **salut.obj**. Acesta este **fişierul obiect**, reprezentând primul pas în obţinerea fişierului executabil. Acesta este un fişier în limbaj maşină, toate datele şi instrucţiunile lui fiind dimensionate, în lungime, la cea de octet (8 biţi). Totuşi, acesta nu este încă un program executabil. Urmează faza de legare prin care codul predefinit al funcţiilor C folosite în fişierul text vor fi adăugate fişierului obiect.

O **funcţie** C este o porţiune de program pentru rezolvarea unei sarcini distincte din program. De exemplu, funcţia **printf()** rezolvă sarcina scrierii textului `Salutare, tinerete!` pe ecran. Această funcţie este definită în una din bibliotecile standard ale limbajului C. O bibliotecă se compune dintr-o colecţie de funcţii, scrise de noi sau de firme, pentru a putea fi folosite în programele C. Toată programarea din C se bazează pe funcţii, **main()** fiind tot o funcţie care va cuprinde corpul programului principal.

Pentru acest caz, programul de legare va căuta definiţia funcţiei **printf()** în bibliotecă, de unde o va extrage şi o va adăuga la programul obiect, în final rezultând programul executabil **salut.exe**.

Linia 8 este cea mai importantă din program, fiind şi cea mai des folosită funcţie pentru afişarea datelor pe ecranul monitorului. Surprinzător însă, **printf()** nu face parte din limbajul C. C este de fapt divizat în două secţiuni majore: ***limbajul*** în sine şi ***bibliotecile***

lui. După cum am mai spus, **printf()** face parte dintr-o bibliotecă. Ea preia şirul (textul dintre ghilimele) cuprins între acolade şi face afişarea lui pe ecranul monitorului. O funcţie poate citi datele transferate ei prin ceea ce se află între parantezele ei rotunde (argumentele funcţiei), le prelucrează, apoi va întoarce un rezultat. Din acest motiv la folosirea unei funcţii C trebuie să ştie ce are de făcut, ce fel de date are de preluat şi ce fel de dată va fi rezultatul?

Atunci când definim propriile funcţii este responsabilitatea noastră să informăm C despre aceste detalii. Pentru că într-o bibliotecă sunt sute de funcţii C disponibile ar fi dificil să memorăm pentru fiecare dintre ele aceste detalii ca să le putem comunica limbajului C în cazul utilizării lor în programe. Din acest motiv există fişierele antet (header files) care au rolul stocării acestor detalii corespunzătoare tuturor funcţiilor din biblioteci.

În cazul nostru, **linia 2** furnizează compilatorului C toate detaliile necesare despre funcţia **printf()** prin includerea fişierului antet corect pentru funcţia **printf()**. Fişierul antet se numeşte **stdio.h**. Funcţia **printf()** are o linie în acest fişier text care spune limbajului C ce fel de date poate prelua funcţia şi ce fel de răspuns poate întoarce. Fişierul antet conţine o mulţime de astfel de descrieri pentru multe alte funcţii. Astfel, se includ informaţii despre zeci de funcţii în programul C, însă C le va ignora pe toate dacă nu are nevoie de ele.

#include este o instrucţiune specială numită **directivă de preprocesor** şi este o comandă adresată programului compilator - ea nu apare în programul final. Comenzile de acest fel instruiesc compilatorul să realizeze o acţiune oarecare. În acest caz directiva îi spune compilatorului să plaseze întregul conţinut al fişierului cu numele **stdio.h** în programul C, începând cu linia din care este scrisă directiva, la noi, **linia 2**. Numai după ce compilatorul execută această linie va şti toate detaliile necesare utilizării funcţiei **printf()**.

Liniile 1 (**/* salut.c */**) şi **3** (**/* Afiseaza "Salutare, tinerete!" pe ecran */**) sunt comentarii. Compilatorul le ignoră în momentul traducerii, ele având rol numai pentru programator, fiind folosite pentru descrierea efectului unor instrucţiuni sau porţiuni de program. În C, începutul comentariului se marchează prin secvenţa de caractere */** iar terminarea lui prin secvenţa de caractere ***/**. Când compilatorul întâlneşte secvenţa **/*** se opreşte din traducerea programului până când întâlneşte caracterele ***/** pentru închiderea comentariului. Folosirea comentariilor va face programul mai uşor de înţeles. Acestea se pot plasa oriunde în program cu condiţia să nu cuprindă cumva instrucţiuni ale programului. În această idee, programul anterior se poate scrie sub forma:

```
1  /* Programul va fi salvat pe disc sub numele de: salut.c */
2
3  #include<stdio.h> /* Fisierul antet pentru functia printf() */
4
5      /* Programul afiseaza
6  "Salutare, tinerete!" pe ecran */
7
8  main()
9  {
10         printf("Salutare, tinerete!");
11
12         return(0);
13 }
```

2.6 main()

Linia 8 este scrisă sub forma **main()**. Ştiţi deja că **main** este punctul din care începe programul şi că este o funcţie.

Instrucţiunile limbajului pot fi executate numai dacă sunt scrise în interiorul unei funcţii. Funcţiile se folosesc pentru compartimentarea unor programe mari în sarcini mai mici şi din acest motiv, mai uşor de rezolvat. Acţiunile din interiorul unei funcţii sunt separate de acţiunile realizate în interiorul alteia, programul devenind **modular**. Aceasta este o altă denumire pentru specificarea spargerii programului în bucăţi mai mici, cât mai uşor de gestionat.

Funcţia **main()** trebuie să fie prezentă în orice program executabil C pentru că este locul de unde se începe execuţia programului. Porţiunea cuprinsă între acolade se numeşte **corpul programului.** Pentru programul **salut.c** acesta este format din liniile **printf("Salutare, tinerete!");** şi **return(0);**, iar **liniile 10** şi **12** se numesc **instrucţiuni.** În C, orice secvenţă de instrucţiuni cuprinsă între acolade - caracterele { , } din **liniile 9** şi **13** - se numeşte **bloc.** În particular, corpul unei funcţii este întotdeauna cuprins între acolade, fiind un bloc. Uneori, denumirea de **bloc** se mai foloseşte pentru a sublinia că **un grup de instrucţiuni sunt legate între ele**, adică nu sunt tratate independent.

2.7 printf()

Linia 10 este un exemplu clasic al unei linii de program C. Instrucţiunile de acest fel se termină prin caracterul **punct şi virgulă** (;) numit şi **terminator de instrucţiuni.** Orice acţiune în C este o **instrucţiune** şi orice instrucţiune C se termină cu punct şi virgulă. Acest caracter special anunţă compilatorul că instrucţiunea este completă.

Modul tipic de utilizare al funcţiei în C este prezentat prin **printf()**. Datele care dorim să fie prelucrate de funcţie se pun între parantezele rotunde. Aceste date ce vor fi apoi transferate funcţiei poartă denumirea de **argumente.** În acest caz, **printf()** are un singur argument, scris sub forma şirului de caractere **"Salutare, tinerete!"**. În C există mai multe forme de date, numite tipuri de date, care pot fi prelucrate prin instrucţiuni ale limbajului C.

printf() este foarte complexă şi asigură afişarea unei varietăţi de tipuri de date pe ecran. Există posibilitatea manipulării formei de afişare a datelor pe ecran. De exemplu, se pot folosi caracterele **\t** şi **\n** în şirul caracterelor de afişat astfel:


```
1  /* Programul va fi salvat pe disc sub numele de: salut.c */
2
3  #include<stdio.h> /* Fisierul antet pentru functia printf() */
4
5          /* Programul afiseaza
6          "Salutare, tinerete!" pe ecran */
7
8  main()
9  {
10         printf("Salutare,\n\t tinerete!");
11
12         return(0);
13 }
```

Utilizarea acestor caractere va duce la afişarea pe ecran a textului `Salutare, tinerete!` sub forma:

```
Salutare,
tinerete!
```


`tinerete!` se afişează sub `Salutare,` pe o linie nouă şi aliniat. Trecerea la linie nouă, datorită caracterului **`/n`** se mai numeşte linie nouă (**`newline`**) şi constă în avansul la începutul următoarei linii de ecran. Există un astfel de set de caractere speciale care nu se afişează pe ecran, dar influenţează afişarea celorlalte caractere. Toate acestea încep prin caracterul **`backslash`** (**/**) iar la întâlnirea lui, următorul caracter se traduce într-o acţiune cu o semnificaţie specifică literei puse după **/**. Câteva astfel de caractere speciale sunt:

Caracter special	Semnificaţie
`\a`	alertă (**`ring bell`**) - emisia unui sunet de către difuzorul PC-ului.
`\n`	linie nouă (**`newline`**) - trecere la linie nouă pe ecranul monitorului.
`\b`	şterge la stânga cursorului (**`backspace`**).
`\t`	tabulator (**`tab`**) - salt cu un număr fixat de caractere pe orizontală.
`\v`	tabulator vertical (**`vertical tab`**).
`\r`	retur de car (**`carriage return`**) - salt la începutul liniei de pe ecran.
`\\`	caracterul **`backslash`**.

ANTAL Tiberiu Alexandru
tel.: 0040-264-530918
e-mail: antaltiberiu@pcnet.ro

2.8 return()

Ultima linie din program este **`return(0);`**. Am inclus această instrucţiune din cauza lui **`main()`** care este o funcţie. C aşteaptă ca funcţiile să întoarcă o valoare cu excepţia cazului în care se specifică explicit lipsa valorii întoarse. În acest caz particular, nu s-a indicat lipsa valorii întoarse pentru funcţia **`main()`**, motiv pentru care C presupune că va întoarce o valoare întreagă, acesta fiind cazul implicit pentru funcţii care întorc valoare. Valoarea numerică **`0`** se va întoarce prin instrucţiunea **`return(0);`** la terminarea funcţiei **`main()`**. Scrierea acestei instrucţiuni într-o formă parţială (numai cuvântului **`return`**) sau omiterea ei totală va duce la transferarea unui întreg cu valoare aleatoare sistemului de operare. Această valoarea aleatoare ar putea fi **`0`**, caz în care aleator, programul s-ar termina cu succes, sau mai probabil diferită de zero, caz în care va indica terminare cu eroare a programului.

ANTAL Tiberiu Alexandru
tel.: 0040-264-530918
e-mail: antaltiberiu@pcnet.ro

3 Variabile, tipuri de date și constante

Un limbajul C, un **obiect** este o **regiune din RAM** în care se poate stoca o singură valoare la un moment dat. Obiectul primește un nume prin care valoarea stocată va putea fi identificată în mod unic. Constantele şi variabilele sunt obiecte care se pot manipula în programele C prin operatori şi instrucţiuni. Declaraţiile prezintă variabilele care se vor utiliza în program, le asociază un tip şi eventual o valoare iniţială.

3.1 Variabile

În C toate variabilele trebuie declarate înainte de a fi folosite în program. Programatorul realizează aceasta printr-o **declarație** care asociază numele variabilei unui tip de dată. Declaraţia specifică modul de intepretare a numelui de către compilatorul C.

Denumirea tehnică pentru numele de variabilă este identificator de variabilă sau, mai pe scurt, variabilă. Numele de variabilă sugerează că valorile stocate sub aceste nume pot varia sub controlul strict al programatorului prin intermediul unor obiecte numite operatori. Variabilele sunt caracterizate prin:

- **tip** (**type**)
- **vizibilitate** (**scope**)
- **durata de existență** sau **persistență** (**storage class**)
- **legare** (**linkage**)

Din motive didactice, în acest capitol voi aborda numai descrierea tipului, celelalte caracteristici vor fi discutate pe parcurs. Trebuie să ştiţi însă că aceste caracteristici sunt specificate simultan prin declaraţia variabilei.

Nume de variabile corecte

Literele, cifrele şi caracterul **underscore** (_) sunt caractere permise într-un nume de variabilă. Primul caracter trebuie să fie literă sau underscore. Conform standardului C caracterul **underscore** este de evitat la începutul unui nume de variabilă pentru că de obicei funcţiile din biblioteci încep cu acesta. Caracterele mici şi mari sunt distincte în C. Astfel, **a** şi **A** sunt două nume diferite. Tradiţional, literele mici se folosesc pentru nume de variabile, iar cele mari pentru constante simbolice.

Exemple de **nume corecte** sunt: **x, v_intreg, x1, val_de_inceput**
Exemple de nume **incorecte** sunt: **v-intreg, 1x, val de inceput, in#lire.**

În general, caracterul **undescore** se foloseşte în interiorul unui nume de variabilă pentru a creşte lizibilitatea lui. Numărul maxim de caractere care formează numele depinde de compilatorul folosit, însă sunt semnificative, conform **standardului**, numai primele **31** de

caractere, adică numai acestea sunt folosite pentru a distinge numele de variabile între ele. Unele implementări impun restricţii de lungime pentru numele de variabile externe - care pot fi folosite de exemplu din limbajul de asamblare - de cel mult `6` caractere.

Numele de variabile nu pot avea aceeaşi scriere sintactică cu numele cuvintelor cheie. Astfel, nume de variabile ca `if`, `else`, `for`, `case` sunt incorecte.

Este bine ca numele variabilei să exprime scopul variabilei în program şi să nu fie exagerat de lung.

Declararea variabilelor

O declaraţie specifică un tip - acestea vor fi discutate în continuare - şi o listă de una sau mai multe nume de variabile. În exemplul care urmează, `int` şi `float` sunt tipuri de date.

```
int i, j, k;
float x, vector[108];
```

Forma utilizată mai sus condensează declaraţiile numelor de variabile `i`, `j` şi `k` respectiv `x` şi `vector` în câte o singură linie de program C. Este posibilă şi scrierea distribuită a declaraţiilor sub forma:

```
int i;
int j;
int k;
float x;
float vector[108];
```

3.2 Tipuri de date

În limbajul C există un grup de **`categorii predefinite de date`** denumite **`tipuri`**. Utilizatorul poate modela o problemă utilizând numai aceste categorii predefinite, eventual poate defini noi categorii în termenii celor existente. **`Tipurile predefinite`** în C sunt grupate în două clase: **`simple`** (uneori numite interne sau de bază) şi **`structurate`**. Asocierea tipului unui nume de variabilă se face prin intermediul declaraţiei de variabilă şi face ca:

- domeniul de valori pe care variabila le poate lua, adică valorile care pot fi stocate în urma evaluării unor expresii sau valorile care pot fi atribuite direct variabilei, să fie cunoscute;
- tipul unei valori specificate de variabilă să poată fi dedus din declaraţia acesteia, fără a executa calcule aritmetice;
- fiecare operator sau funcţie să folosească argumente de un anumit tip şi să întoarcă rezultate de un anumit tip.

3.2.1 Tipuri simple

Tipurile simple ale limbajului C sunt: **`void`**, **`char`**, **`short`**, **`int`**, **`long`**, **`float`** şi **`double`**.

3.2.1.1 Tipul void

Cuvântul rezervat **`void`** are `3` utilizări: pentru a specifica **`valoarea "inexistentă" întoarsă de o funcţie`**, pentru a specifica un tip de argumente de **`funcţie care nu are argumente`** şi pentru a specifica **`un poantor la un tip nespecificat`**. Toate aceste denumiri şi situaţii speciale se vor lămuri numai după studierea funcţiilor şi a tipului structurat de dată numit

poantor.

3.2.1.2 Tipuri întregi

Limbajul C suportă mai multe categorii de întregi. Numele lor reflectă spaţiul de RAM alocat pentru stocare. Teoretic, un întreg **short**, ar trebui să ocupe spaţiu mai puţin în RAM decât tipul întreg **long**.

În tabelul care urmează se prezintă spaţiul de memorie alocat pentru tipurile de date întregi de către compilatorul Visual C++ 6.0 şi domeniul de valori corespunzător care poate fi stocat în variabilele de tipul respectiv.

Tip	Spaţiu	Domeniu de valori	Minim şi maxim în "limits.h"
char	1 octet	[-128, 127]	[CHAR_MIN, CHAR_MAX]
signed char	1 octet	[-128, 127]	[SCHAR_MIN, SCHAR_MAX]
unsigned char	1 octet	[0, 255]	[0, UCHAR_MAX]
short	2 octeţi	[-32768, 32767]	[SHRT_MIN, SHRT_MAX]
unsigned short	2 octeţi	[0, 65535]	[0, USHRT_MAX]
int	4 octeţi	[-2147483648, 2147483647]	[INT_MIN, INT_MAX]
unsigned int	4 octeţi	[0, 4294967295]	[0, UINT_MAX]
long	4 octeţi	[-2147483648, 2147483647]	[LONG_MIN, LONG_MAX]
unsigned long	4 octeţi	[0, 4294967295]	[0, ULONG_MAX]

Fiecare compilator are un set de constante utile folosite la determinarea domeniului de valori ale tipurilor de bază. De exemplu, numele **INT_MIN** şi **INT_MAX** sunt disponibile în fişierul **limits.h** sub formă de constante care pot fi utilizate la scriere de programe. **INT_MAX** este cel mai mare număr întreg care poate fi stocat într-o variabilă de tipul **int** pentru cazul compilatorului curent. Să zicem că trecem de la compilatorul firmei Microsoft la cel al firmei Borland. **INT_MAX** va referi valoarea maximă care se poate stoca cu acest nou compilator. Acest lucru va rămâne valabil chiar şi la schimbarea sistemului de operare. Fişierul **limits.h** va conţine toate limitările compilatorului, iar includerea lui în programul utilizator va permite accesul la toate aceste valori extreme actualizate.


```
1  /* INTEX1.C */
2  #include <limits.h>
3  #include <stdio.h>
4  int main(void)
5  {
6          unsigned long z=ULONG_MAX;
7
8          printf("valoarea maxima int = %i ", INT_MAX);
9          printf("valoarea minima int = %i\n", INT_MIN);
10         printf("valoarea maxima unsigned = %u\n", UINT_MAX);
11         printf("valoarea maxima long int = %li\n", LONG_MAX);
12         printf("valoarea maxima unsigned log = %lu\n",z);
13
14
```

```
15      return 0;
16  }
```

Rezultate:

```
valoarea maxima int = 2147483647 valoarea minima int = -2147483648
valoarea maxima unsigned = 4294967295
valoarea maxima long int = 2147483647
valoarea maxima unsigned log = 4294967295
```

Pentru înţelegerea acestui exemplu trebuie să ştiţi că funcţia **printf("valoarea maxima int = %i ", INT_MAX)** scrie pe ecran valoarea stocată în **INT_MAX** sub controlul specificatorului de format **%i**. În funcţia **printf()**, pentru fiecare tip de dată, trebuie să folosim un specificator de format corespunzător tipului de dată afişat. În acest scop se prezintă tabelul următor:

Tip	Specificator de format
char	**%c**
signed char	**%c**
unsigned char	**%c**
short	**%hi**
unsigned short	**%hu**
int	**%i, %d**
unsigned int	**%u**
long	**%li**
unsigned long	**%lu**

Tipul int

Când doriţi să lucraţi cu valori întregi mici în loc de **int** folosiţi **short** şi aveţi impresia că aţi folosit mai puţină memorie. Invers, dacă valorile trebuie să fie mari, folosiţi tipul **long**. Problema pusă în termenii lui "mare" şi "mic" este fără prea mult sens datorită impreciziei standardului C. Este suficient că ştiţi că **SHRT_MAX** este undeva pe la **32.767**, iar **LONG_MAX** pe la **2.147.483.647**. Aceste valori nu sunt "clare" pentru tipul **int** se stochează uneori pe **2**, alteori pe **4** octeţi. Astfel, pentru o implementare particulară de C, nu avem de unde să ştim dacă valoarea maximă este de ordinul a **32 de mii** sau a **2 miliarde**. Din acest motiv, programele portabile în adevăratul sens al cuvântului nu folosesc tipul **int**, ci numai **short** sau **long**.

Tipul **unsigned** este fără semn. Din acest motiv toţi biţii se pot folosi pentru stocarea valorii numerice, în loc ca cel mai semnifivativ să fie pe post de bit de semn. Aceasta înseamnă că valoarea cea mai mare **unsigned** poate fi cel mult dublul valorii maxime ale lui **int**.

Tipul char

Tipul de dată **char** se foloseşte pentru stocarea şi prelucrarea caracterelor. Valorile caracter se formează prin plasarea lor între apostrofuri (’). Astfel, **char b_mic = 'b';** stochează valoarea ASCII - numită şi codul ASCII - a literei **b** mic, valoarea numerică **98**, în variabila cu numele **b_mic**.

```
1  /* CHAREX1.C */
2  #include<limits.h>
3  #include<stdio.h>
4
5  int main(void)
6  {
7          char a_mic = 'a';
8          char b_mic = 'b';
9
10         printf("valoarea minima char = %i, ", CHAR_MIN);
11         printf("valoarea maxima char = %i\n", CHAR_MAX);
12
13         printf("dupa '%c' urmeaza '%c'\n", b_mic, b_mic+1);
14         printf("litera mare '%c' este '%c'\n", b_mic, b_mic + 'A'-'a');
15
16         return 0;
17 }
```

Rezultate:

```
valoarea minima char = -128, valoarea maxima char = 127
dupa 'b' urmeaza 'c'
litera mare 'b' este 'B'
```

Cele două constante definesc valoarea minimă şi maximă pentru un caracter. Pentru că în acest caz spaţiul pentru stocare de 1 octet este garantat de **standard** putem presupune valabil domeniul de [0, 255]. Totuşi, nu este definit dacă `char` este cu semn sau fără, astfel încât domeniul ar putea fi şi [-128, 127].

Compilatorul poate efectua operaţii aritmetice cu tipul `char`. Astfel, `b_mic+1` înseamnă `98+1`, adică `99`. Expresia se poate citi în sensul codificării ASCII ca următorul caracter după `b` mic. Calculul codului ASCII corespunzător lui `B` mare se face în expresia `b_mic + 'A'-'a'`. Dacă veţi parcurge codificarea ASCII veţi observa că "distanţa" între codurile ASCII ale aceleiaşi litere mari şi mici este constantă, motiv pentru ea se poate calcula prin diferenţa `'A'-'a'`.

Întregi în baze diferite

În C este posibilă scrierea valorilor întregi şi în baze diferite de cea zecimală. Este posibilă utilizarea numai a bazelor `8` şi `16`, adică numerele se pot scrie în octal sau în hexazecimal. Constantele `octale` se specifică prin plasarea unui `0` în faţa cifrei, astfel dacă 9 este un număr zecimal corect, 09 ar fi incorect pentru că este o constantă octală, iar 9 nu este o cifră corectă în sistemul de numeraţie cu baza 8.

Specificarea unui `x` după `0` determină compilatorul să interpreteze numărul întreg ca o constantă `hexazecimala`. Aici literele `a`, `b`, `c`, `d`, `e`, `f` sunt folosite pentru a reprezenta numerele de la `10` la `15` care în hexazecimal `sunt cifre`. Este nesemnificativ dacă se scrie cu litere mari sau mici, astfel `0x1ffa`, `0x1FFA` şi `0x1FfA` reprezintă acelaşi număr, adică `0X1FFA`.

```
1  /* BAZEEX1.C */
2  #include<limits.h>
3  #include<stdio.h>
4
5  int main(void)
6  {
7          int zec=22, oct=022, hex=0x22;
```

```
8       printf("zec=%d, oct=%d, hex=%d\n", zec, oct, hex);
9       printf("zec=%d, oct=%o, hex=%x\n", zec, oct, hex);
10
11      return 0;
12  }
```

Rezultate:

```
zec=22, oct=18, hex=34
zec=22, oct=22, hex=22
```

%d determină **afişarea** întregului **în notaţia zecimală**; aceasta este echivalentă cu **%i**;
%o determină **afişarea** întregului **în notaţia octală**;
%x determină **afişarea** întregului în **notaţia hexazecimală**, cu litere mici pentru "**abcdef**";
%X determină **afişarea** întregului în **notaţia hexazecimală**, cu litere mari pentru "**ABCDEF**".

Deşi discuţia este un pic mai tehnică, aş dori să descriu ce se petrece când declarăm o variabilă. Compilatorul este informat despre numele variabilei şi tipul acesteia. De exemplu, pentru o declaraţie de forma **int i1**, la întâlnirea lui **int** compilatorul alocă 4 octeţi din RAM pentru a stoca o valoare întreagă. De asemenea, generează o tabelă de simboluri în care va introduce simbolul **i1** şi adresa relativă de RAM la care s-au stocat cei 4 octeţi. Astfel, dacă mai târziu în program vom scrie **i1=7;** la momentul execuţiei programului, când se va ajunge la această instrucţiune, valoarea **7** va fi plasată în memorie la adresa locaţiei rezervate pentru stocarea valorii lui **i1**. În acest sens variabila întreagă **i1** este un obiect căruia îi sunt asociate două "valori". Prima este valoarea numerică întreagă stocată acolo de noi şi a doua "valoarea" adresei locaţiei de memorie dată de compilator. În unele cărţi aceste două valori sunt cunoscute sub numele de **r-value** şi **l-value**.

3.2.1.3 Tipuri reale

Fişierul antet standard care conţine constantele legate de tipurile reale din C este **float.h**. Sunt definite constantele corespunzătoare valorilor extreme şi preciziei celor trei tipuri reale.

Tip	Specificator de format	Spaţiu	Domeniu de valori	Precizie în zecimale	Minim şi maxim în float.h
float	**%f %e %g**	4 octeţi	±3.4E+/-38	7	[FLT_MIN, FLT_MAX]
double	**%lf %le %lg**	8 octeţi	±1.7E+/-308	15	[DBL_MIN, DBL_MAX]
long double	**%Lf %Le %Lg**	10 octeţi	±1.2E+/-4932	19	[LDBL_MIN, LDBL_MAX]

Tipul float

Este **cel mai puţin precis** dintre tipurile reale. Calculele cu acest tip sunt cele mai rapide, însă este relativ uşor să apară **erori de depăşire** (**overflow** sau **underflow**) pentru că domeniul pentru reprezentarea acestui tip este îngust. **Spaţiul de stocare este cel mai mic**, motiv pentru care este preferat pe sistemele de calcul cu RAM puţin.

Tipul double

Este tipul real **cu spaţiu de stocare mediu** în C. Calculele se efectuează mai încet decât la tipul **float**, dar mai **precis**. Spaţiul de stocare fiind mai

mare, domeniul de valori care se poate stoca este şi el mai larg.

Tipul **long double**

Este tipul de dată cu spaţiul de stocare cel mai mare în C. Calculele ce folosesc tipuri **long double** sunt cele mai încete dintre tipurile reale, dar şi cele mai precise.


```
1  /* FLOATEX1.C */
2  #include<float.h>
3  #include<stdio.h>
4
5  int main()
6  {
7                  double x=3.1416, y=1.7e-5, z=7000000000.0;
8
9                  printf("x=%lf\ty=%lf\tz=%lf\n",x,y,z);
10                 printf("x=%le\ty=%le\tz=%le\n",x,y,z);
11                 printf("x=%lg\ty=%lg\tz=%lg\n",x,y,z);
12
13                 printf("x=%7.3lf\ty=%.3le\tz=%.4g\n",x,y,z);
14
15                 return 0;
16 }
```

Rezultate:

```
x=3.141600      y=0.000017       z=7000000000.000000
x=3.141600e+000 y=1.700000e-005  z=7.000000e+009
x=3.1416        y=1.7e-005       z=7e+009
x=  3.142       y=1.700e-005     z=7e+009
```


%lf îi spune funcţiei **printf()** să afişeze numărul cu 6 zecimale, indiferent de mantisa numărului;

%le îi spune funcţiei **printf()** să afişeze numărul cu 6 zecimale, folosind notaţia exponenţială, adică în loc de **0.00012** se afişează **1.2e-04** (**1.2** înmulţit cu 10^{-4});

%lg acest specificator face afişarea numai a datelor utile, zerourile în plus nefiind afişate. De asemenea, numărul este afişat în formă cât mai scurtă posibil. De exemplu, în loc de **0.00012** valoarea se va afişa într-o forma mai concisă: **1.2e-04**;

%7.3lf numărul total de caractere folosit pentru afişarea valorii este **7**, iar **3** numărul de caractere zecimale. Se afişează **3.142** pentru că numărul are numai **5** caractere din **7** şi **2** spaţii în stânga valorii pentru a completa diferenţa de la **5** la **7** caractere. Se mai observă că valoarea reală a fost automat rotunjită;

%.3le specifică **3** zecimale în formatul exponenţial;

%.4lg specifică **4** zecimale, dar acestea nu sunt afişate pentru că sunt 0-uri, iar formatul trebuie să fie cel mai scurt.

Declaraţie cu iniţializare

O variabilă poate primi o valoare iniţială în linia declaraţiei. Dacă numele variabilei este urmat de semnul egal (=) şi o expresie valoarea expresiei este stocată în variabilă, procedura numindu-se iniţializarea variabilei. Iniţializarea se face o singură dată înainte de începerea execuţiei programului, iar valoarea expresiei de iniţializare trebuie să fie o constantă. Acesta este cazul iniţializării explicite. Unele variabile sunt iniţializate implicit de C în funcţie de durata lor de existenţă - aceste cazuri se vor discuta în cadrul capitolului de funcţii. Exemple de iniţializări explicite sunt:

```
char ln = '/n';
```

```
int a = 0;
float eps = 0.00001;
char mesaj[ ] = "Ai o eroare!";
```

3.3 Constante

O constantă este un obiect cu unică valoare în program caracterizat prin:

- `tip` (**type**)
- `valoare` (**value**)

După modul său de scriere o valoare constantă se încadrează în unul din următoarele tipuri de constante: întregi, reale, caracter, enumerate sau şir de caractere.

3.3.1 Constante numerice

3.3.1.1 Constante întregi

Constanta întreagă este un **`număr întreg cu sau fără semn`** care nu-şi schimbă valoarea în program şi care se poate scrie în sistemele de numeraţie zecimal (baza `10`), octal (baza `8`) sau hexazecimal (baza `16`). Dacă şirul de cifre care formează constanta întreagă este prefixat cu cifra `0` numărul este scris în octal, dacă prefixul este `0x` numărul este scris în hexazecimal. Tipul numărului determină spaţiul pe care acesta se reprezintă în RAM-ul sistemului de calcul. **`Implicit`**, constanta întreagă **`are tipul int`** şi se reprezintă pe 2 octeţi (16 biţi), în cazul în care se specifică sufixele `u`, `U` (**`unsigned`**) numărul este fără semn, iar pentru sufixele `l`, `L` tipul este **`long`** şi numărul se reprezintă pe 4 octeţi (32 de biţi). În exemplele care urmează constanta întreagă zecimală `28` este scrisă în toate formele posibile:

```
28 /* constanta intreaga zecimala */
28L /* constanta intreaga zecimala de tipul long */
034 /* constanta intreaga octala reprezentand 28 in zecimal */
034L /* constanta intreaga octala de tipul long */
0x1c /*constanta intreaga hexazecimala reprezentand 28 in zecimal */
0x1cL /*constanta intreaga hexazecimala de tipul long*/
```

3.3.1.2 Constante reale

Constanta reală numită şi flotantă este **`un număr real cu semn`** ce nu-şi schimbă valoarea în program şi care se reprezintă în RAM printr-o porţiune întreagă, una fracţionară şi un exponent. Prezenţa punctului zecimal este obligatorie în constanta reală, excepţie făcând cazul când se include exponentul.

```
23.123 /* numarul real scris "normal" sub forma 23,123 */
1.456e1 /* 14.56 */
1456e-2 /* 14.56 */
1.456e-1 /* 0.1456 */
-1.23e3 /* -1230 */
.123e01 /* 1.23 */
```

Din exemplele anterioare se observă că implicit constantele reale sunt pozitive, cu excepţia cazului în care sunt precedate de semnul minus. Ele pot avea tipurile **`float`**, **`long`** şi **`long double`**. **`Implicit`** o constată reală este de tipul **`double`**. Dacă se folosesc sufixele **`f`** sau **`F`** tipul este **`float`**, iar în cazul sufixelor `l` sau **`L`** tipul este **`long double`**.

```
777.L /* constanta reala de tipul long double */
777.F /* constanta reala de tipul float */
777.D /* constanta reala de tipul double */
777. /* constanta reala de tipul double */
```

3.3.1.3 Constante caracter

Constanta caracter se formează prin **includerea între apostrofuri (') a unui singur caracter.** Caracterele sunt codificate conform reprezentării **ASCII** (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange). Aceasta asociază fiecărui caracter, grafic sau negrafic, o valoare numerică întreagă unică în domeniul `0-127`. Valoarea constantei caracter este valoarea numerică a caracterului respectiv asociat în reprezentarea ASCII (codul ASCII). Din acest motiv constantele caracter pot participa în operaţii numerice asemenea unor întregi. Codurile ASCII ale caracterelor se prezintă sub forma unei liste care specifică codificarea în octal, zecimal şi hexazecimal pentru fiecare caracter împreună cu reprezentarea lui grafică la tipărire şi semnificaţia lui (pentru caracterele negrafice):

Octal	Zecimal	Hexazecimal	Nume, Explicaţie
000	0	0x00	NUL
001	1	0x01	SOH
002	2	0x02	STX
003	3	0x03	ETX, Control-C
004	4	0x04	EOT
005	5	0x05	ENQ
006	6	0x06	ACK
007	7	0x07	BEL
010	8	0x08	BS, şterge la stânga cursorului
011	9	0x09	HT, tabulator orizontal
012	10	0x0a	LF, trecere la linie nouă
013	11	0x0b	VT, tabulator vertical
014	12	0x0c	FF, form feed
015	13	0x0d	CR, retur de car
016	14	0x0e	SO
017	15	0x0f	SI
020	16	0x10	DLE
021	17	0x11	DC1, XON, Control-Q
022	18	0x12	DC2
023	19	0x13	DC3, XOFF, Control-S
024	20	0x14	DC4
025	21	0x15	NAK
026	22	0x16	SYN
027	23	0x17	ETB
030	24	0x18	CAN
031	25	0x19	EM
032	26	0x1a	SUB
033	27	0x1b	ESC, escape
034	28	0x1c	FS
035	29	0x1d	GS
036	30	0x1e	RS
037	31	0x1f	US
040	32	0x20	spaţiu
041	33	0x21	!, semn de exclamare
042	34	0x22	", ghilimele
043	35	0x23	#, hash
044	36	0x24	$, dolar
045	37	0x25	%, procent
046	38	0x26	&, ampersand
047	39	0x27	', apostrof
050	40	0x28	(, paranteză deschisă
051	41	0x29	), paranteză închisă
052	42	0x2a	*, asterisc
053	43	0x2b	+, plus
054	44	0x2c	,, virgulă
055	45	0x2d	-, minus
056	46	0x2e	., punct
057	47	0x2f	/, oblique stroke
060	48	0x30	0
061	49	0x31	1
062	50	0x32	2
063	51	0x33	3
064	52	0x34	4
065	53	0x35	5
066	54	0x36	6
067	55	0x37	7
070	56	0x38	8
071	57	0x39	9
072	58	0x3a	:, două puncte
073	59	0x3b	;, punct şi virgulă
074	60	0x3c	<, mai mic decât
075	61	0x3d	=, egal
076	62	0x3e	>, mai mare decât
077	63	0x3f	?, semn de întrebare
0100	64	0x40	@, şi comercial
0101	65	0x41	A
0102	66	0x42	B
0103	67	0x43	C
0104	68	0x44	D
0105	69	0x45	E

0106	70	0x46	F	0144	100	0x64	d
0107	71	0x47	G	0145	101	0x65	e
0110	72	0x48	H	0146	102	0x66	f
0111	73	0x49	I	0147	103	0x67	g
0112	74	0x4a	J	0150	104	0x68	h
0113	75	0x4b	K	0151	105	0x69	i
0114	76	0x4c	L	0152	106	0x6a	j
0115	77	0x4d	M	0153	107	0x6b	k
0116	78	0x4e	N	0154	108	0x6c	l
0117	79	0x4f	O	0155	109	0x6d	m
0120	80	0x50	P	0156	110	0x6e	n
0121	81	0x51	Q	0157	111	0x6f	o
0122	82	0x52	R	0160	112	0x70	p
0123	83	0x53	S	0161	113	0x71	q
0124	84	0x54	T	0162	114	0x72	r
0125	85	0x55	U	0163	115	0x73	s
0126	86	0x56	V	0164	116	0x74	t
0127	87	0x57	W	0165	117	0x75	u
0130	88	0x58	X	0166	118	0x76	v
0131	89	0x59	Y	0167	119	0x77	w
0132	90	0x5a	Z	0170	120	0x78	x
0133	91	0x5b	[, paranteză dreptunghiulară deschisă	0171	121	0x79	y
0134	92	0x5c	\, backslash	0172	122	0x7a	z
0135	93	0x5d	], paranteză dreaptunghiulară închisă	0173	123	0x7b	{, acoladă deschisă
0136	94	0x5e	^, caret	0174	124	0x7c	|, bară verticală
0137	95	0x5f	_, underscore	0175	125	0x7d	}, acoladă închisă
0140	96	0x60	`, back quote	0176	126	0x7e	~, tilda
0141	97	0x61	a	0177	127	0x7f, delete, şterge caracterul de pe poziţia curentă	
0142	98	0x62	b				
0143	99	0x63	c				

Unele caractere ASCII sunt negrafice, utilizarea lor fiind specială. În C ele se scriu precedate de caracterul "\". Valorile zecimale ale acestor coduri sunt cuprinse în domeniul 0-31. De exemplu, caracterul cu numele **`BS`** (**`backspace`**) are ca efect trecerea din poziţia curentă a cursorului cu un caracter la stânga. În C el se scrie sub forma '**`\b`**'. Caracterul cu numele **`BEL`** (**`audible alert`**) are ca efect emiterea unui sunet de către difuzorul calculatorului şi în C se scrie sub forma '**`\a`**'. Aceste caractere speciale se mai numesc şi **`secvenţe escape`**. Ele vor fi tratate mai pe larg la funcţia **`printf()`**.

Un caracter se va stoca într-o variabilă de tipul **`char`** şi se poate specifica prin codul lui octal, zecimal sau hexazecimal. Astfel, în locul lui '**`A`**' se poate scrie **`0101`**, **`65`** sau **`0x41`**. O eroare frecventă este când în locul valorii întregi **`0`** scriem '**`0`**'. '**`0`**' este o constantă caracter având codul ASCII **`48`**, care nu are nici o legătură cu valoarea numerică **`0`** (zero). Există posibilitatea specificării unui caracter oarecare prin scrierea '**`\ooo`**', unde **`ooo`** este un număr în octal sau prin '**`\xhh`**', unde **`hh`** este un număr în hexazecimal corespunzător codului ASCII al caracterului. Conform celor spuse, constanta caracter zero se poate defini prin '**`\0`**'. Aceasta are o semnificaţie specială în limbajul C fiind caracterul care marchează terminarea unui şir de caractere.

3.3.1.4 Constanta şir de caractere

Din punct de vedere tehnic, un şir de caractere se scrie în C ca **`un tablou cu elemente de tipul caracter`**. Întrucât tabloul este, asemenea poantorului, un tip de dată structurată

pentru moment este suficient să ştiţi cum se scrie.

O constantă şir este o secvenţă vidă - fără nici un caracter - sau cu mai multe caractere cuprinse între ghilimele: "**john**", "**balada**", "**te salut baietas**". Semnificaţia specială a caracterului "\" se păstrează şi aici, el putând apărea în interiorul unor constante şir. De obicei, "\" apare sub formele:

\' caracterul **apostrof**, folosit şi la constanta şir;
\" caracterul **ghilimele**, folosit la constanta şir;
\\ caracterul **backslash**, adică \.

De exemplu, "**Ghio mi-o zis** \"**Salutare ...**\"" este o constantă şir care conţine două ghilimele, iar de exemplu, ' \' ' este constanta caracter '.

3.3.2 Constante cu nume

Dacă ideea folosirii caracterelor **F**, **L**, **D** etc. pentru specificarea tipului unei constante pare prea puţin riguroasă, C are cuvântul cheie **const** care se poate scrie în faţa declaraţiei unei variabile pentru a defini un nume de **constantă cu tip**. Valoarea acesteia nu se va putea modifica. Câteva exemple sunt:

```
const char sapt=7;
const double pi=3.141592653590;
const char mesaj[ ] = "Salutare !";
```

Prin folosirea lui **const** tipul este specificat explicit. O constantă este tratată ca o **r-value** (**right value**), adică poate să apară numai în dreapta unui egal (=). O instrucţiune de forma **sapt=8;** este incorectă şi produce mesajul de eroare "invalid lvalue".

3.3.3 Constante ale preprocesorului

Preprocesorul este o caracteristică specifică limbajului C, fiind **un editor de texte neinteractiv** care este "plasat" în faţa compilatorului. Din acest motiv compilatorul nu "vede" instrucţiunile scrise în program, ci numai pe acelea care formează ieşirea preprocesorului. Asemenea oricărui editor de texte, preprocesorul are o seamă de instrucţiuni de substituţie de şiruri de caractere şi de includere de fişiere în textul documentului curent - care este programul C scris de noi. O astfel de instrucţiune este **#include** care are ca efect includerea conţinutului fişierului specificat în programul care va fi compilat. Operaţiile de căutare şi substituţie ale preprocesorului se definesc prin instrucţiunea **#define**.

```
1  #include <stdio.h>
2
3  #define PI 3.141592653500L
4  #define ZILE_SAPT 7
5  #define CR '\xd' /* Retur de Car */
6  #define PIpe2 PI/2.
7
8  int zile=ZILE_SPAT;
9  long pi_in_long=PI; /* Valoarea lui PI */
```

Forma generală a **#define**-ului este:

Formă: `#define nume text_de_substituţie`

Numai cuvintele întregi sunt substituite, iar şirurile de caractere C - secvenţe de caractere cuprinse între ghilimele - şi comentariile nu se modifică. De exemplu, şirul de caractere `PI` va fi substituit prin şirul de caractere `3.1415926535OOL` în tot programul C. Observaţi că punct şi virgulă (;) lipseşte de la sfârşitul instrucţiunilor preprocesorului, iar între `PI` şi `3.1415926535OOL` nu apare semnul egal (=). Din motive istorice constantele preprocesorului se scriu cu litere mari. În `linia 6` observaţi o `expresie constantă`. Aceasta este o expresie care are toţi termenii constante. Expresiile obişnuite sunt evaluate în timpul execuţiei programului, expresia constantă este evaluată în timpul compilării programului întrucât toţi termenii ei sunt determinaţi deja la acel moment.

3.3.4 Constante enumerate

Constantele enumerate se prezintă sub forma unei liste de întregi cu nume.

Formă: `enum identificator { listă }`

În limbajul C K&R, listele de valori erau definite folosind `#define`-ul astfel:

```
#define duminica 0
#define luni 1
#define marti 2
#define miercuri 3
#define joi 4
#define vineri 5
#define sambata 6
```


În limbajul C modern, există posibilitatea declarării unor tipuri enumerate de date folosind `enum` astfel:

```
enum zi {duminica, luni, marti, miercuri, joi, vineri, sambata} zidelucru;
```


Efectul este acelaşi ca în cazul codului anterior, adică `duminica = 0`, , `sambata = 6` dar aici, faţă de lista de `#define`-uri de mai sus, apare ascunderea informaţiilor şi posibilitatea declarării unor variabile de tipul zi. Valoarea primei constante este implicit `0`, valorile nespecificate cresc în progresie aritmetică cu raţia `1` de la ultima valoare specificată. Enumerarea constă în specificarea unei mulţimi de constante întregi. O declaraţie de tip enumerat dă numele opţional - în exemplul de mai sus `zi` - unui identificator de tipul enumerat - `zidelucru` - şi defineşte o mulţime de identificatori întregi - `duminica`, `luni`, `marti`, ..., `sambata` - uneori numiţi şi `membri`. O variabilă de tipul enumerat stochează una din valorile specificate la definirea tipului enumerat. Următoarele reguli se aplica pentru membrii unei mulţimi enumerate:

- pot exista valori constante duplicate. Astfel, de exemplu, se poate asocia valoarea 0 la doi identificatori diferiţi, de exemplu `null` şi `zero`, în aceeaşi mulţime enumerată;
- numele din lista enumerată trebuie să fie distincte de alte nume cu aceeaşi vizibilitate, incluzând aici numele de variabile şi numele din alte liste enumerate. Vizibilitatea este porţiune de program în care un nume poate fi folosit. Noţiunea se va lămuri la prezentarea funcţiilor.

Variabilele de tipul **enum** pot fi utilizate la indexarea expresiilor şi ca operanzi ai operatorilor aritmetici şi relaţionali. Un exemplu clasic de tip enumerat este cel **boolean**, de care C nu beneficiază implicit. În C, la declararea variabilei enumerate este obligatorie folosirea lui **enum** înaintea tipului enumerat (vezi exemplul care urmează - **Boolean** este un tip de dată, iar **sfarsit** şi **inceput** sunt variabile de acest tip).


```
enum Boolean /* Declararea unui tip enumerat numit Boolean */
{
        false,   /* false = 0, true = 1 */
        true
};
enum Boolean sfarsit, inceput; /* doua variabile de tipul Boolean */
```


În ANSI C, expresiile care definesc valorile membrilor din listă trebuie să fie constante întregi.

Conf. dr. ing. ANTAL Tiberiu Alexandru
Universitatea Tenhică din Cluj-Napoca
Facultatea Construcţii de Maşini
Catedra de Mecanică şi Programare
Curs de limbaj C

Copyright 2001. Toate drepturile sunt rezervate autorului.

Multiplicarea acestui document în scop comercial este interzisă.

Sudenţii participanţi la orice formă de învăţământ superior bugetar pot multiplica acest document pentru uzul personal.

Sudenţii participanţi la orice formă de învăţământ cu plată sau alte persoane doritoare pot multiplica documentul numai contra cost şi cu acordul scris al autorului. În acest scop mă puteţi contactă la:

ANTAL Tiberiu Alexandru
tel.: 0040-264-530918
e-mail: antaltiberiu@pcnet.ro

4 Operatori

Până acum am vorbit despre variabile şi constante. Valorile stocate în variabile se pot modifica sub controlul operatorilor. Operatorul acţionează asupra operanzilor, iar "numele" operatorului spune ce se va face cu valorile stocate în operanzi. După numărul de operanzi asupra cărora acţionează avem operatori unari, binari şi ternari.

4.1 Operatorul de atribuire

Operatorul de atribuire în C este semnul `=`. În operaţiile de atribuire operandul din stânga operatorului de atribuire trebuie să fie o `l-value` modificabilă - un nume de variabilă. Valoarea numerică din dreapta egalului se atribuie variabilei din stânga. Dacă în stânga avem o expresie aritmetică, aceasta este evaluată în prealabil, pentru a i se găsi valoarea. Dacă este cazul, după evaluare, expresia aritmetică se converteşte la tipul numelui de variabilă la care se atribuie. După atribuire, expresia de atribuire ia valoarea operandului din stânga lui `=` dar fără a fi o `l-value`. Dacă vorbim despre adunarea "1+2", atunci "+" este operatorul, "1" şi "2" sunt operanzii, efectul este adunarea valorilor numerice ale celor doi operanzi, iar rezultatul adunării este 3. Dacă vorbim despre atribuirea simplă "i=1;" atunci "=" este operatorul, "i" şi "1" sunt operanzii, efectul este stocarea valorii numerice "1" în variabila cu numele"i", iar rezultatul atribuirii este valoarea numerică "1".

```
1 int a,b,c;
2 a=b=c=77;
3 printf("%d\n",c=108);
```

Când se face o atribuire în C, valoarea atribuită este stocată într-un registru al UCP. Această valoare din registru este cea care se foloseşte în continuare, eventual ea poate fi rescrisă de următoarea atribuire.

În `linia 2`, valoarea de `77`, se plasează în RAM la adresa corespunzătoare variabilei `c` şi se stochează într-un registru din UCP, în acelaşi timp. Valoarea din registru este apoi atribuită lui `b` şi lui `a`.

În `linia 3`, valoarea `108` se atribuie lui `c`, iar valoarea `108` se plasează şi în registru. Valoarea salvată în registru este cea afişată de `printf()` sub controlul descriptorului `"%d"`. Atribuirea din `linia 2` se numeşte `multiplă`, iar cea din `linia 3`, `simplă`.

4.2 Operatori aritmetici

Operatorii aritmetici ai limbajului C sunt:

Formă de scriere în C	Nume operaţie
+	adunare
-	scădere
*	înmulţire
/	împărţire
%	modulo (restul împărţirii)

"+" şi "-" pot să fie atât **unari** cât şi **binari**, adică sunt permise formele de scriere:


```
1  x=-y;
2  x=z-u;
3  x=+k; /* Acelasi lucru cu x=x+k */
4  x=a+b;
```

În **linia 1** operatorul "-" este unar pentru că are un singur operand, pe **y**, iar rezultatul întors de el este valoarea lui **y** înmulţită cu **-1**. În **linia 2** operatorul "-" are doi operanzi pe **z** şi **u**, valoarea întoarsă de el fiind diferenţa dintre **z** şi **u**. În **linia 3** "+" este unar, iar ca rezultat este echivalent cu forma de scriere din comentariul alăturat.

C are un operator numit "restul împărţirii". Dacă pentru împărţirea întreagă rezultatul expresiei **17/3** este **5** la folosirea operatorului modulo, în locul celui de împărţire, rezultatul expresiei **17%3** este **2**. Un astfel de calcul are sens numai pentru numere întregi întrucât, numai aici există rest. La împărţirea unor valori reale nu există rest, ci numai o fracţie.

Unele limbaje de programare au operatori speciali pentru ridicare la putere ("^" în Basic sau "**" în FORTRAN). C nu are un astfel de operator, dar are implementată funcţia de bibliotecă **pow()** în acest scop.

În continuare dau un exemplu de utilizare al operatorilor aritmetici:


```
1  /* OPARIT.C */
2  int main()
3  {
4          int a, b, c;      /* variabile intregi */
5
6          a = 12;
7          b = 3;
8          c = a + b;        /* adunare   */
9          c = a - b;        /* scadere   */
10         c = a * b;        /* inmultire */
11         c = a / b;        /* impartire */
12         c = a % b;         /* modulo */
13         c = 12*a + b/2 - a*b*2/(a*c + b*2);
14         c = c/4+13*(a + b)/3 - a*b + 2*a*a;
15         a = a + 1;        /* incrementarea variabilei a */
16         b = b * 5;
17
18         a = b = c = 20;     /* atribuire multipla */
19         a = b = c = a + b * c/3;
20
21         a = (b = (c = 20)); /* identica cu linia 18 */
```

```
22      return 0;
23  }
```

Rezultatele programului sunt neinteresante pentru a fi afişate, dar ca exerciţiu, pentru că ştiţi deja cum să afişaţi valoarea unor întregi utilizând **printf()**, puteţi insera câteva instrucţiuni de ieşire în diferite puncte ale programului pentru a verifica dacă intuiţi bine rezultatele calculate în program. Nu uitaţi să adăugaţi **#include<stdio.h>** pentru cazul în care utilizaţi **printf()**.

4.2.1 Conversii la utilizarea operatorilor aritmetici

Dacă tratăm cazul operatorului de adunare "+", el trebuie să poată aduna numai întregi, numai reali, cât şi întregi cu reali. Cel mai simplu ar fi fost să existe câte un operator de adunare pentru fiecare caz, însă atât pentru întregi (**char**, **short** , **int**, **long**) cât şi pentru reali (**float**, **double**, **long double**) există mai multe tipuri. Se observă că numărul de variante şi de combinaţii este mare. C-ul elimină această problemă prin operatorul "+" care va alege singur tipul de adunare pe care o va face. Dacă "+" are doi operanzi de tipul **int** se va realiza adunare întreagă, dacă are doi operanzi de tipul **float** se va realiza adunare flotantă - de numere reale. Ce se petrece în cazul în care unul din operanzi are tipul **int**, iar celălat tipul **float**? Fie exemplul:

```
1   /* OPARITCV.C */
2   #include<stdio.h>
3   int main (void)
4   {
5       int a,b,c;
6       double x,y,z;
7
8       a=7;
9       b=3;
10
11      x=7.0;
12      y=3.0;
13
14      c=a/b; /* a si b sunt intregi => se realizeaza impartire intreaga */
15      printf("%i\n",c);
16
17      z=x/y; /* x si y sunt reali => se realizeaza impartire flotanta */
18      printf("%f\n",z);
19
20      z=a/b;
21      printf("%f\n",z);
22      return 0;
23  }
24
```

Rezultate:

```
2
2.333333
2.000000
```

Conform celor spuse, compilatorul decide dacă rezultatul întors de operand este întreg sau real, funcţie de tipul operanzilor. În acest caz pentru **linia 14** (**c=a/b**) împărţirea este

întreagă pentru că **a** şi **b** sunt de tip întreg (**int**). Interesant este rezultatul afişat în **linia 21**, aici împărţirea este tot de tip întreg, însă din cauză că **z** este real, rezultatul împărţirii se converteşte automat la tipul **double**. O astfel de împărţire produce probleme din cauză că împărţirea a doi întregi va da întotdeauna un întreg, deşi poate noi am dori ca împărţirea să se facă la fel ca pentru cazul unor valori reale, din moment ce atribuim rezultatul împărţirii unei variabile reale. Conform **standardului**, într-o astfel de împărţire partea zecimală se pierde fără să apară rotunjiri în sus sau în jos ca în alte limbaje de programare. O variantă de trecere la împărţirea flotantă ar fi ca valorile lui **a** şi **b** să fie atribuite la două variabile reale, apoi împărţirea să se efectueze între aceste variabile.

Când un operator are operanzi de tipuri diferite se realizează conversia automată a operazilor mai "limitaţi" spre operanzii mai "complecşi" pentru evitarea pierderii de informaţie. Dacă se doreşte realizarea explicită a unor conversii - **conversie forţată** - din cauza că nu se face automat sau dorim să fim siguri de portabilitatea codului, se poate folosi **operatorul de conversie** (**type cast**) care are forma:

Formă:	**(tip) expresie**

Operatorul de conversie modifică temporar tipul variabilei sau expresiei la care se aplică. Din acest motiv în **liniile 11**, **12** şi **13** ale programului următor (**OPARITC1.C**) se realizează împărţire flotantă. Modalitatea de lucru a compilatorului de exemplu pentru **linia 12** este: **a** se converteşte temporar la un **double**, din acest motiv **7** devine **7.0**. Acum, compilatorul are de împărţit un **double** la un **int** şi îl va converti automat (în engleză termenul este **promoted**) pe **b** la tipul **double** (**b** devine **3.0**), realizând împărţirea **double** (flotantă) conform regulilor aritmeticii numerelor reale, iar rezultatul va fi **2.333333**. În **linia 16**, parantezele rotunde nu rezolvă nimic, iar rezultatul împărţirii este **2**, care va fi convertit la **2.000000**.

```
1  /* OPARITC1.C */
2  #include<stdio.h>
3  int main (void)
4  {
5          int a,b;
6          double z;
7
8          a=7;
9          b=3;
10
11         z=(double)a/(double)b;
12         z=(double)a/b;
13         z=a/(double)b;
14         printf("%f\n",z);
15
16         z=(double)(a/b);
17         printf("%f\n",z);
18
19         return 0;
20 }
```

Rezultate:

```
2.333333
2.000000
```

Pentru rigoare reiau cazul operanzilor de tipuri diferite când C execută conversia

automată a unui tip de operand la altul, pe baza unor reguli stricte. Regulile se aplică numai pentru operatorii binari şi numai dacă cei doi operanzi nu sunt de acelaşi tip. Pentru a determina care conversii trebuie realizate, compilatorul aplică următorul algoritm operanzilor unui operator binar:

1. Când unul din operanzi este de tipul `long double`, celălalt se converteşte şi el la tipul `long double`;
2. În cazul în care condiţia de mai sus nu este îndeplinită şi unul din operanzi este de tipul `double`, celălalt operand se converteşte şi el la tipul `double`;
3. În cazul în care condiţia de mai sus nu este îndeplinită şi unul din operanzi este de tipul `float`, celălalt operand se converteşte şi el la tipul `float`;
4. În cazul în care nici una dintre condiţiile de mai sus nu este îndeplinită (din cauza tipului operanzilor) urmează conversiile întregi, astfel:
   a) Când unul dintre operanzi este de tipul `unsigned long`, celălalt operand este convertit la tipul unsigned long;
   b) Când condiţiile de mai sus nu sunt îndeplinite iar unul din operanzi este de tipul `long` şi celălalt de tipul `unsigned int`, ambii operanzi se convertesc la tipul `unsigned long`;
   c) Când condiţiile de mai sus nu sunt îndeplinite şi unul dintre operanzi este de tipul `long`, celălalt operand se converteşte la tipul `long`;
   d) Când condiţiile de mai sus nu sunt îndeplinite şi unul dintre operanzi este de tipul `unsigned int`, celălalt operand se converteşte la tipul `unsigned int`;
   e) Când nici una din condiţiile de mai sus nu sunt îndeplinite ambii operanzi se convertesc la tipul `int`.

```
float f;
double d;
int i;
unsigned long ul;

d=i*ul; /* i convertit la unsigned long (4.), rezultatul inmultirii convertit la double */
d=ul+f; /* ul se converteste la float (3.), rezultatul adunarii se converteste la double */
```

Dacă rezultatul unei operaţii de conversie depăşeşte domeniul de valori ce corespunde tipului rezultat, valoarea va fi trunchiată şi rezultatul va fi dubios. După cum vedeţi tipul `char` nu apare mai sus pentru că el este în realitate un tip `int` mai "modest". În calcule, când este cazul, tipul `char` se converteşte automat la cel `int`.

4.3 Operatorii de incrementare (++) şi decrementare (--)

C are doi operatori unari speciali pentru incrementarea (adunarea lui 1 la valoarea unei variabile) şi decrementarea (scăderea cu 1 a valorii unei variabile) variabilelor. De ce s-au implementat aceşti operatori într-un limbaj atât de mic cum este C? Răspunsul stă în corespondenţa directă cu limbajul maşină. Toate UCP au o formă de instrucţiune "`inc`" prin care se creşte cu 1 conţinutul unei locaţii de memorie, la fel şi pentru decrementare. Din acest motiv C va traduce operatorii de incrementare şi decrementare direct în instrucţiuni ale limbajului maşină. Fie secvenţa de instrucţiuni:

```
1 int a,b;
2 a=7;
3 b=3;
```

```
4  a++; /* a devine 8, forma postfixata */
5  ++b; /* b devine 4, forma prefixata */
6  b-- /* b devine 3 */
7  --a; /* a devine 7 */
```

Se observă că operatorii `++` şi `--` au o formă **`postfixată`** şi una **`prefixată`**. În ambele cazuri se adună, respectiv se extrage 1 din valoarea variabilei. Diferenţele apar la valorile atribuite.

```
1  /* INCDEC.C */
2  int main()
3  {
4          int a, b;     /* intregi in [-32768, 32767] */
5
6          a=7;
7          b=++a; /*echivalent cu  ->      1) a=a+1;      */
8          printf("b=%d, a=%d\n", a,b);    2) b=a;
9
10         a=7;
11         b=a++; /*echivalent cu  ->      1) b=a;        */
12         printf("b=%d, a=%d\n", a,b);    2) a=a+1;
13
14         return 0;
15 }
```

Rezultate:

```
b=8,  a=8
b=8,  a=7
```

`++, --` prefixat

Valoarea expresiei unare este valoarea după incrementare sau decrementare. La utilizarea operatorilor prefixaţi, incrementarea sau decrementarea are loc prima dată, apoi valoarea modificată va fi atribuită. În **`linia 8`**, valoarea curentă **`7`** din **`a`** devine **`8`**, apoi valoarea **`8`** se copiază, datorită lui **`=`**, în **`b`**.

postfixat `++, --`

Valoarea expresiei unare este valoarea operandului. Pentru operatorii postfixaţi, incrementarea sau decrementarea are loc în faza a doua. Valoarea nemodificată este atribuită, apoi este modificată. În **`linia 12`**, valoarea **`7`** este copiată, datorită lui **`=`**, în **`b`**, apoi valoarea din **`a`** este crescută cu **`1`**, devenind **`8`**.

Pentru realizarea celor descrise, C fie foloseşte, fie nu foloseşte un registru temporar pentru a salva valoarea curentă. În cazul operatorilor prefixaţi, **`b=++a;`** se realizează incrementarea, apoi se transferă valoarea. În cazul postfixat, **`b=a++;`** C stochează valoarea curentă de **`7`** într-un registru, apoi are loc incrementarea (şi valoarea devine **`8`**). În continuare se extrage valoarea din registru şi se stochează în **`b`**. Astfel, inclusiv în acest caz, incrementarea are loc înainte de atribuire.

4.4 Valori de adevăr în C

C nu are un tip de dată special pentru lucrul cu valori de adevăr. Problema este implementată prin prisma valorilor numerice ale variabilelor.

adevărat (True) Orice valoare numerică nenulă are valoarea de adevăr **adevărat** (True). Astfel, întregii **1** şi **-7** au valoarea de adevăr adevărat pentru că sunt nenuli. Similar este tratată şi valoarea reală **0.007**.

fals (False) Orice valoare numerică nulă are valoarea de adevăr **fals** (False). Astfel, valorile numerice **0**, **-0**, **+0**, **0.00** **0e-01** au toate valoarea de adevăr **fals**.

Din acest motiv testarea valorilor de adevăr în C este foarte directă. E suficient să stocăm valoarea într-un registru, după care verificăm dacă există un bit nenul. Dacă da, atunci valoarea se poate identifica imediat ca fiind adevărată. Dacă nici un bit nu este setat pe 1 atunci valoarea este falsă.

4.5 Operatori relaţionali

Operatorii relaţionali **realizează comparerea a 2 valori numerice**, întorcând valoarea de adevăr **adevărat** sau **fals** în funcţie de validitatea relaţiei specificate. Dacă relaţia este **falsă**, valoarea numerică rezultată este **0**; dacă e **adevărată**, valoarea numerică este **1**. Excepţie fac de la această regulă operatorii **==** şi **!=**, când se garantează numai valoarea numerică de **0** pentru **fals**.

Conf. dr. ing. ANTAL Tiberiu Alexandru
tel: 0040-264-538918
e-mail: antaltiberiu@pcnet.ro

Operator relaţional	Relaţia testată
>	primul operand este **mai mare** decât al doilea
>=	primul operand este **mai mare sau egal** decât al doilea
<	primul operand este **mai mic** decât al doilea
<=	primul operand este **mai mic sau egal** decât al doilea
==	primul operand este **egal** cu al doilea
!=	primul operand este **diferit** de al doilea

Operanzii pot fi de tipul întreg, real sau poantor. Tipurile operanzilor pot să fie diferite, caz în care se vor realiza conversiile uzuale pentru tipurile întregi şi reale în vederea comparaţiei. În cazul operanzilor poantor, aceştia trebuie să fie poantori la acelaşi tip. Funcţie de implementarea de compilator folosită, comparaţiile poantorilor oferă posibilităţi adiţionale (comparaţia unui poantor cu valoarea numerică **0** sau a unui poantor de tip **void**) care pot fi însuşite prin consultarea documentaţiei compilatorului.


```
1 int a,b,c;
2 a=7;
3 b=a>5;
4 c=b>=144;
```


În **linia 3** ,valoarea numerică **1**, adică **adevărat**, se atribuie lui **b**. În **linia 4**, valoarea numerică **0**, adică **fals**, se atribuie lui **c**.

Pentru folosirea acestor operatori în exemple C clasice voi anticipa puţin şi voi descrie pe scurt instrucţiunea **if**, deşi ea se va discuta formal numai în capitolul de instrucţiuni. Una dintre formele de scriere posibile ale lui **if** este:

```
if (condiţie)
        instrucţiune care se execută când condiţie este adevărată;
else
        instrucţiune care se execută când condiţie este falsă;
```


Nu confundaţi = cu ==

O eroare frecventă a începătorilor în C este utilizarea operatorului de atribuire (=) în locul celui de testare a egalităţii (==).


```
1  /* EREGAL.C */
2  #include<stdio.h>
3
4  int main(void)
5  {
6          int a = 0;
7
8          if (a=0)
9                  printf("a=0\n");
10         else
11                 printf ("a<>0\n");
12
13         return 0;
14 }
```

Rezultate:

```
a<>0
```


În linia 6 lui `a` i se atribuie valoarea `0`. Testul nu este în realitate decât o altă atribuire. Compilatorul va suprascrie vechea valoare a lui `a` cu valoarea `0` şi o salvează într-un registru. Apoi, este testată valoarea din registru. Pentru că are valoarea numerică `0` este **falsă** şi porţiunea `else` din construcţia `if` se va executa. Programul va lucra corect dacă în loc de `a=0` se va scrie `a==0`.

4.6 Operatori logici

Operatorii logici sunt binari, mai puţin cel de negare (!) şi **lucrează cu valorile logice de adevărat/fals permiţând combinarea expresiilor relaţionale** (care se obţin cu ajutorul operatorilor relaţionali). Rezultatul unui operator logic este o valoare logică, tipul rezultatului este `int`. O expresie relaţională este: `a=((b>5) && (c<=123)) || (d>22) || (!e)`. C implementează următorii operatori logici: **ŞI**, **SAU** şi **NU**. Ar fi fost mai intuitivă folosirea unor cuvinte pentru aceşti operatori, C însă i-a implementat prin simbolurile `&&`, `||` şi `!`. Operatorul **NU** (`!`) neagă valoarea operandului. Pentru ceilalţi operatori prezint în continuare tabele de adevăr după care operează:

Tabel de adevăr pentru **ŞI** logic

&&	fals	adevărat
fals	fals	fals
adevărat	fals	adevărat

Tabel de adevăr pentru **SAU** logic

||	fals	adevărat
fals	fals	adevărat
adevărat	adevărat	adevărat

Ca de obicei, valorii de adevăr `adevărat` i se asociază orice valoare numerică nenulă, iar valorii de adevăr `fals`, valoarea numerică `0`. Operanzii pot fi de tip întreg sau poantor.

Spus în cuvinte, folosind valori de adevăr, `ŞI` logic între doi operanzi întoarce valoarea de adevăr `adevărat` numai dacă ambii operanzi au valoarea de adevăr `adevărat`. `SAU` logic, descris folosind valorile numerice corespunzătoare valorilor de adevăr, întoarce valoarea întreagă `1` dacă unul dintre operanzi este nenul, altfel întoarce valoarea `0`.

Pentru expresiile care conţin mai mulţi operatori logici consecutivi, C garantează că evaluarea lor se face de la stânga la dreapta şi că va realiza scurtcircuitare când valoarea de adevăr a unei expresii se poate determina fără a-i evalua toţi termenii;

```
if (a && b&& c || d)
        instrucţiune;
```


În expresia logică de mai sus prima oară se evaluează `a`. Dacă ia valoarea `adevărat`, se evaluează `b`. Dacă `b` ia valoarea de adevăr `adevărat` se trece la `c` etc., adică evaluarea se face de la stânga la dreapta. Mai departe, dacă rezultatul de `adevărat` sau `fals` poate fi stabilit la un moment dat, nu se mai fac evaluări în acelaşi scop în continuare. De exemplu, dacă `a` este `adevărat` şi `b` `fals`, `c` nu ar mai fi evaluat - pentru că un `fals` într-un lanţ de `&&`-uri face ca tot lanţul să fie `fals` (vezi tabela de adevăr pentru `&&`) - termenul tehnic fiind de scurtcircuitare. Următoarea evaluare care se face este pentru `d`.

Utilizarea parantezelor rotunde

În porţiunea de program care urmează prezint un exemplu de program care nu face ce aţi putea crede. În `linia 4` a exemplului următor, prin condiţia dintre parantezele rotunde încerc să zic: "dacă a nu este egal cu 3". Din nefericire, `!a` se evaluează prima oară, iar pentru că `a` are valoarea `3`, adică este `adevărată`, `!a` primeşte valoarea de `fals`, adică `0`. Apoi, `0` se compară cu `3` şi rezultatul afişat va fi "`a egal cu 3`".

```
1  int a;
2  a=3;
3
4  if (!a == 3)
5          printf("a diferit de 3\n");
6  else
7          printf("a egal cu 3\n");
```

Pentru eliminarea acestei probleme se va folosi o pereche de paranteze în plus sub forma `(!(a == 3))`. Prin acestea codul va fi mai clar, iar mentenanţa lui şi depanarea vor fi mult uşurate în viitor.

4.7 Operatori pe biți

Operatorii pe biţi sunt binari, mai puţin cel de negare (~). Operanzii sunt interpretaţi ca o secvenţă de biţi, excepţiile fiind descrise în cadrul notelor din tabelul următor. Următorii operatori la nivel de bit sunt definiţi în C:

Denumire	Scriere în C	Notă
ŞI pe biți	&	expresie1 & expresie2

SAU pe biţi	**|**	**expresie1 | expresie2**
SAU EXCLUSIV pe biţi	**^**	**expresie1 ^ expresie2**
NU pe biţi	**~**	**~expresie1** operatorul mai este numit şi "complementul lui 1" întrucât rezultatul lui este complementul faţă de 1 al operandului
deplasare la stânga	**<<**	**expresie1 << expresie2** operandul **expresie1** este interpretat ca o secvenţă de biţi, iar operandul **expresie2** specifică numărul de biţi cu care se realizează deplasarea la dreapta a secvenţei de biţi corespunzătoare operandului **expresie1**
deplasare la dreapta	**>>**	**expresie1 >> expresie2** operandul **expresie1** este interpretat ca o secvenţă de biţi, iar operandul **expresie2** specifică numărul de biţi cu care se realizează deplasarea la stânga a secvenţei de biţi corespunzătoare operandului **expresie1**

Ei se pot aplica numai tipurilor întregi (**char**, **short**, **int** şi **long** cu sau fără semn), iar rezultatul lor este tot o valoare de tipul întreg. C a fost iniţial creat de Kernighan şi Ritchie pe o maşină PDP-11 pentru scrierea unui sistem de operare şi din acest motiv ei au avut nevoie de instrumente cât mai flexibile pentru manipularea regiştrilor maşinii. Aşa s-au născut operatorii la nivel de bit. Veţi observa că **&** pe bit şi **&&** logic respectiv | pe bit şi || logic seamănă. Întrucât Kernighan şi Ritchie foloseau mai mult operatorii pe biţi decât cei logici au rezervat un singur caracter operatorilor pe biţi pentru o scriere cât mai scurtă. În continuare sunt prezentate tabelele de adevăr pentru operatorii la nivel de biţi:

Tabel de adevăr pentru **ŞI pe biţi**

&	0	1
0	0	0
1	0	1

Tabel de adevăr pentru **SAU pe biţi**

|	0	1
0	0	1
1	1	1

Tabel de adevăr pentru **SAU EXCLUSIV pe biţi**

^	0	1
0	0	1
1	1	0

```
1  /* OPPEBIT1.C */
2  #include<stdio.h>
3  void main()
4  {
5          unsigned short a, b, c;
6          char *format1, *format2;
```

```
7
8       format1 = " %04X %s %04X = %04X\n";
9       format2 = "%c%04X =%04X\n";
10      Curs de limbaj C
11      a=0x0FF0;
12      b=0xFF00;
13
14      c = a << 4;printf(format1,a,"<<",4,c);
15      c = a >> 4;printf(format1,a,">>",4,c);
16      c = a & b;printf(format1,a," & ",b,c);
17      c = a | b;printf(format1,a," | ",b,c);
18      c = a ^ b;printf(format1,a," ^ ",b,c);
19      c = ~a;printf(format2,'~',a,c);
20      a = ~c;printf(format2,'~',c,a);
21  }
```

Rezultate:

```
 0FF0 << 0004 = FF00
 0FF0 >> 0004 = 00FF
 0FF0  &  FF00 = 0F00
 0FF0  |  FF00 = FFF0
 0FF0  ^  FF00 = F0F0
~0FF0 = F00F
~F00F = 0FF0
```

Specificatorul de format **%04X** spune că se vor folosi **0**-uri pentru poziţiile neocupate, lăţimea numărului fiind de **4** cifre, numărul fiind afişat în hexazecimal (**X** este descriptor pentru baza **16**). Se ştie că un număr întreg scris în baza de numeraţie **10**, poate fi convertit în unul binar - baza de numeraţie **2** - caz în care numărul din baza **10** se va transforma într-o secvenţă de **0** şi **1**. Aceşti operatori **acţionează la nivel de bit**, în sensul că **primul bit al primului operand va fi pus în corespondenţă cu primul bit al celui de al doilea operand**, se efectuează operaţia dictată de operator, apoi se trece la perechea a doua de biţi ş.a.m.d.

Pentru cei care nu înţeleg hexa, programul **OPPEBIT2.C** prezintă valorile hexa în binar. Conversia valorii numerice întregi fără semn în binar se face folosind o funcţie din biblioteca standard C. Ea se numeşte **ltoa** şi pentru rigoare o descriu pe scurt în continuare:

ltoa - converteşte un întreg lung într-un şir de caractere

Prototip: **char * _ltoa(long valoare, char * şir, int baza);**
Fişier prototip: **ltoa - <stdlib.h>**
Valoare întoarsă: **_ltoa** întoarce un poantor la un şir de caractere. În caz de eroare nu se întoarce o valoare specială.
Parametri: **valoare** - număr care va fi convertit;
şir - şir rezultat în urma conversiei;
baza - baza sistemului de numeraţie în care se face conversia, un număr întreg fără semn între **2** şi **36**.
Remarcă: **şir** este terminat în caracterul **null**.

```
1   /* OPPEBIT2.C */
2   #include<stdio.h>
3   #include<stdlib.h>
4
5   int main(void)
6   {
```

```
7       unsigned short a, b, c;
8       char ca[17], cb[17], cc[17];
9       char *format1,*format2, *format3;
10
11      format1 =" %16s %s %04X = %16s\n";
12      format2 ="\n %16s %s\n %16s\n ----------------\n %16s\n\n";
13      format3 =" %c% 16s = %16s\n";
14
15      a=0x0FF0;
16      b=0xFF00;
17
18      c = a << 3;
19      itoa(a,ca,2);
20      itoa(c,cc,2);
21      printf(format1,ca,"<<",3,cc);
22
23      c = a >> 3;
24      itoa(a,ca,2);
25      itoa(c,cc,2);
26      printf(format1,ca,">>",3,cc);
27
28      c = a & b;
29      itoa(a,ca,2);
30      itoa(b,cb,2);
31      itoa(c,cc,2);
32      printf(format2,ca," & ",cb,cc);
33
34      c = a | b;
35      itoa(a,ca,2);
36      itoa(b,cb,2);
37      itoa(c,cc,2);
38      printf(format2,ca," | ",cb,cc);
39
40      c = a ^ b;
41      itoa(a,ca,2);
42      itoa(b,cb,2);
43      itoa(c,cc,2);
44      printf(format2,ca," ^ ",cb,cc);
45
46      c = ~a;
47      itoa(a,ca,2);
48      itoa(c,cc,2);
49      printf(format3,'~',ca,cc);
50      c = ~c;
51      itoa(a,ca,2);
52      itoa(c,cc,2);
53      printf(format3,'~',cc,ca);
54
55      return 0;
56  }
```

Rezultate:

```
    111111110000 << 0003 =  111111110000000
    111111110000 >> 0003 =         111111110

    111111110000  &
1111111100000000
 ----------------
```

```
     111100000000
 1111111110000  |
 1111111100000000
 ----------------
 1111111111110000
```



```
     111111110000  ^
1111111100000000
 ----------------
1111000011110000
```



```
~ 1111111110000 = 1111000000001111
~1111000000001111 =  111111110000
```


Deşi programul foloseşte tipul de dată tablou, declarat în **`linia 8`**, care încă nu a fost prezentat, rezultatele lui vin să lămurească felul în care se execută operaţiile la nivel de biţi. Observaţi că **`_itoa`** fiind o funcţie standard, are caracterul **`underscore`** în faţa numelui şi că prototipul ei se include în program în **`linia 3`**.

Programele folosesc tipul întreg **`short`** fără semn pentru că acesta se reprezintă pe 16 biţi. Dacă foloseam tipul **`int`**, după cum am mai spus deja, pe unele maşini el se reprezintă pe 16 biţi, iar pe altele pe 32.

Se observă că **`NU`** pe biţi (**`~`**) neagă valorile biţilor din număr, astfel un **`1`** devine **`0`** şi un **`0`** devine **`1`**. Deplasarea la stânga (**`<<`**) cu un bit corespunde înmulţirii valorii numărului întreg cu 2, deplasarea cu 2 biţi la stânga este echivalentă cu o înmulţire cu 4, etc. Poziţiile rămase vacante se vor completa cu 0. Deplasarea la dreapta (**`>>`**) cu 1 bit corespunde împărţirii numărului cu 2. Observaţi că bitul din dreapta se pierde în procesul deplasării fără a mai exista posibilitatea recuperării lui.

Folosiţi **`unsigned`** cu **`>>`**

Pentru deplasarea la dreapta, în cazul numerelor cu semn, cel mai semnificativ bit - bitul corespunzător puterii celei mai mari a lui 2 - care reprezintă semnul, este inserat în noul număr. Dacă tipul este **`unsigned`**, pe poziţiile vacante se vor insera numai **`0`**-uri. Din punctul de vedere al reprezentării interne a numerelor întregi totul este corect, însă dacă cunoaştem acest mod de lucru, bitul de semn poate să fie o surpriză. În cazul numerelor cu semn, surpriza nu apare dacă numărul este pozitiv.

Cazuri clasice de utilizare

Un caz clasic de utilizare a operatorului **`&`** este mascarea unor biţi dintr-un număr. Mascarea înseamnă că efectul lor în valoarea numărului este ascuns. Fie expresia **`a=a & x0f`** în care **`a`** este variabilă caracter. **`x0f`** este în binar **`00001111`**. Datorită lui **`ŞI pe biţi`**, în poziţiile unde avem **`0`** în reprezentarea binară **`00001111`**, obţinem **`0`**, iar în urma **`ŞI`**-ului **`pe biţi`**, valoarea respectivilor biţi din **`a`** nu mai contează. Un alt caz clasic de utilizare, de data aceasta pentru operatorul **`|`**, este setarea pe **`1`** a unor biţi dintr-un număr. În urma expresiei **`a=a | x0f`**, acei biţi care sunt **`1`** în numărul **`x0f`** vor fi **`1`** şi în **`a`**.

4.8 Atribuirea compusă

C permite scrierea prescurtată a expresiilor cu mai mulţi operatori. Se poate combina operatorul de atribuire cu unul din operatorii descrişi până acum, astfel că în locul formulei:

```
(expresie1) = (expresie1) <operator> (expresie2);
```

se poate scrie:

```
expresie1 <operator> = expresie2;
```

Iată câteva exemple de scriere condensată a expresiilor:

```
a=a+b; /* se scrie si sub forma */ a +=b;
a=a*b; /* se scrie si sub forma */ a *=b;
a=a/(b+c) /* se scrie si sub forma*/ a /=b+c;
```

C evaluează mai întâi expresia din dreapta operatorului compus, din acest motiv `a/=b+c` este `a=a/(b+c)` şi nu este `a=a/b+c`.

Efectul celor două forme de scriere este acelaşi, însă metoda de evaluare nu. În varianta lungă, valoarea lui `<expresie1>` se "evaluează" de două ori (la întâlnirea operatorilor de adunare şi atribuire), iar în varianta compusă numai o dată. Pentru mai multă claritate să vedem cam cum se generează codul în limbajul maşină pentru `a=a+3;` "ia valoarea lui `a` din RAM şi încarc-o într-un registru", "ia valoarea din registru şi adun-o cu `3`", "ia valoarea din registru şi stocheaz-o înapoi în RAM pe locul lui `a`". Dacă am fi scris numai `a+=3`, din cele de mai sus ar rămâne: "ia valoarea lui `a` stocată în RAM şi adună-o cu `3`". Putem spune că o expresie de forma `expresie1 <operator> = expresie2` este echivalentă cu `expresie1 = expresie1 <operator> (expresie2)`, cu excepţia că `expresie1` este evaluată o singură dată.

Din nou, de ce într-un limbaj atât de mic ca C s-au implementat astfel de atribuiri? Pe vremea lui K&R, când optimizarea codului compilat nu prea exista, posibilitatea scrierii în acest mod creştea viteza de lucru şi scădea dimensiunea programului.

Pe vremea lui K&R diferenţa între un program scris fără şi altul scris cu aceşti operatori era mare (altfel, aceşti operatori nu ar fi fost implementaţi). Azi, în cazul unui compilator cu un optimizator bun, codul ar trebui să fie identic în ambele cazuri. Putem spune că aceşti operatori au fost păstraţi numai din motive de compatibilitate cu versiunile vechi de C.

Operanzii unei atribuiri compuse trebuie să fie de tip întreg sau real. Fiecare operator de atribuire compusă realizează conversia pe care operatorul binar corespunzător o face şi restrânge tipurile operanzilor corespunzători. Pentru cazul adunării (`+=`) şi scăderii (`-=`) operatorii pot avea ca operand stâng un tip poantor caz în care operandul drept trebuie să fie un întreg.

4.9 Operatorul sizeof

În standardul C nu s-a fixat spaţiul de RAM folosit la stocarea tipurilor de date pentru diferitele implementări de compilatorare. Astfel, este posibil ca o implementare de C să folosească 16 biţi (2 octeţi) pentru tipul **int**, iar alta 32 de biţi (4 octeţi). Teoretic, ar putea fi şi implementări care folosesc 64 de biţi pentru întregi lungi (**long int**). Din acest motiv este dificil să cunoaştem în avans spaţiul folosit la stocarea unui obiect. Operatorul **sizeof** vine să rezolve această problemă.

Formă:	**sizeof identificator**	sau	**sizeof (nume-tip)**

Operandul este fie un identificator, adică o expresie unară, fie o expresie conversie de tip, adică un specificator de tip cuprins între paranteze. Operatorul **sizeof** întoarce mărimea operandului în raport cu dimensiunea tipului **char**. El permite evitarea scrierii de cod cu mărimea tipurilor de date dependente de tipul de maşină utilizat. Rezultatul lui **sizeof** este de obicei de tipul **unsigned int**.

```
1  /* SIZEOF.C */
2  #include <stdio.h>
3
4  int main(void)
5  {
6          long a;
7
8          printf("\"a\" se stocheaza pe %u octeti\n", sizeof a);
9          printf("un tip short se stocheaza pe %u octeti\n", sizeof (short));
10
11         return 0;
12 }
```

Rezultate:

```
"a" se stocheaza pe 4 octeti
un tip short se stocheaza pe 2 octeti
```

Aplicat unui **tablou** - o structură de date care va fi discutată în capitolul 9 - operatorul va avea ca rezultat numărul total de octeţi al tabloului. În cazul tipurilor de date structură (**struct**) sau reuniune (**union**) - care se vor discuta în capitolul 10 - operatorul va avea ca rezultat numărul de octeţi al obiectului.

4.10 Operatorul de evaluare secvenţială

Operatorul de evaluare secvenţială, deseori numit şi operatorul virgulă, evaluează cei 2 operanzi între care este interpus secvenţial de la stânga la dreapta.

Formă:	**expresie, expresie_atribuită**

Este folosit în mod tipic la evaluarea uneia sau a mai multor expresii, acolo unde, prin context, este permisă doar o singură expresie. Operandul stâng (**expresie**) este evaluat ca o expresie **void**, deci nu are tip, iar rezultatul operaţiei are aceeaşi valoare şi tip cu operandul din dreapta (**expresie_atribuită**). Operanzii pot avea orice tip. Operatorul nu întoarce o **l-value**. Un exemplu de utilizare va fi dat în capitolul de instrucţiuni, la prezentarea ciclului **for**.

4.11 Operatorul expresie condiţională

C are un singur operator ternar (cu trei operanzi) care este o alternativă pentru instrucţiunea **if (condiţie) instrucţiune1; else instrucţiune2;** prezentată deja sumar.

Formă: `condiţie ? expresie1 : expresie2;`

Prima oară se evaluează **condiţie** şi dacă aceasta are valoarea numerică **0**, valoarea de adevăr este **falsă**. Altfel, valoarea de adevăr este **adevărată**. Când **condiţie** este **adevarata** valoarea imediat după **?**, adică valoarea lui **expresie1**, este folosită pentru a da valoarea expresiei condiţionale. Dacă condiţie este **falsă** valoarea după **:**, adică **expresie2**, se foloseşte pentru a da valoare expresiei condiţionale.

Tipul celor două expresii trebuie să fie identic; nu prea are sens să scrieţi o expresie care se evaluează când la o valoare **double** când la una **char**, în funcţie de o condiţie. Dacă totuşi tipul celor două expresii diferă, tipul expresiei rezultă din aplicarea regulilor de conversie discutate deja. De exemplu, dacă **i** şi **n** sunt de tipul **int**, iar **x** de tipul **float**, **expresia i>0 ? x : n** va avea tipul **float** indiferent de valoarea lui **i**.

Expresia condiţională se poate folosi în orice context în care se poate folosi o "expresie clasică". În general, ea duce la scrierea mai succintă a codului (vezi exemplele care urmează unde acelaşi efect este obţinut cu operatorul ternar **? :** şi cu **if**).


```
int a,b=10,c=-3;
b=(a>c)?a:c;
```

```
int a,b=10,c=-3;
if (a>c)
        b=a;
else
        b=c;
```


Poate veţi fi miraţi că aici am folosit paranteze rotunde în jurul condiţiei; explicaţia acestei scrieri stă în paragrafele care urmează.

4.12 Precedenţa operatorilor

C tratează operatorii cu importanţă diferită conform precedenţei lor. Există aproximativ 40 de operatori în C . Fiecare dintre ei are un "grad de importanţă" specific. Fie programul:

```
1  /* PRECEDE.C */
2  #include<stdio.h>
3  int main(void)
```

```
{
        int a = 3*7+50/9;

        printf("a = %i\n",a);

        return 0;
}
```


Rezultat:

```
a = 26
```

Expresia `3*7+50/9` ar putea avea următoarele semnificaţii: `((3*7)+50)/9` sau(`3*(7+50))/9` sau `3*(7+(50/9))` etc. Semnificaţia ei corectă este însă `(3*7)+(50/9)` pentru că C asociază o importanţă mai mare operatorilor `*` şi `/` decât operatorului `+`. Astfel, înmulţirea şi împărţirea se realizează înainte de adunare. Precedenţa este cea care defineşte ordinea de evaluare a operatorilor dintr-o expresie.

4.13 Asociativitatea operatorilor

Precedenţa nu ne spune totul. De exemplu, care este ordinea de evaluare a operatorilor într-o expresie în care avem mai mulţi operatori de aceeaşi precedenţă, cum sunt `*` şi `/` sau `-` şi `+`? În acest caz se aplică regula asociativităţii. Asociativitatea este de la stânga la dreapta sau de la dreapta la stânga.

```
/* ASOCIAT.C */
#include<stdio.h>
int main(void)
{
        int b = 3*7/9;

        printf("b = %i\n",b);

        return 0;
}
```


Rezultat:

```
b = 2
```


În `3*7/9`, deşi `*` şi `/` au aceeaşi precedenţă, asociativitatea lor este de la stânga la dreapta. Astfel, mai întâi se efectuează `*`, adică se rezolvă subexpresia `3*7`, care dă `21`, apoi `21/9`, care dă `2`.

În cazul unei expresii de forma `a=b+=c;`, `=` şi `+=` au aceeaşi precedenţă, dar asociativitatea de la dreapta la stânga. Operatorul `+=` se va rezolva prima oară, valoarea lui `c` se adună la `b`, apoi rezultatul adunării se atribuie lui `a`.

4.14 Tabel cu precedența și asociativitatea operatorilor C

Tabelul care urmează prezintă precedenţa în ordine descrescătoare precum şi asociativitatea operatorilor C. O mică parte din ei nu au fost încă discutaţi pentru că sunt

legaţi de subiecte încă netratate: poantor, tablou, structură şi apel de funcţie.

Simbol	Operaţia	Asociativitate
`[ ] () . -> postfix ++ postfix --`	Expresie	de la stânga la dreapta
`sizeof & * + - ~ ! (conversie tip) prefix ++ prefix --`	Unar	de la **dreapta la stânga**
`* / %`	Multiplicare	de la stânga la dreapta
`+ -`	Adunare	de la stânga la dreapta
`<< >>`	Deplasare la nivel de bit	de la stânga la dreapta
`< > <= >=`	Relaţionali	de la stânga la dreapta
`== !=`	Egalitate	de la stânga la dreapta
`&`	Şi la nivel de bit	de la stânga la dreapta
`^`	Sau exclusiv la nivel de bit	de la stânga la dreapta
`|`	Sau la nivel de bit	de la stânga la dreapta
`&&`	Şi logic	de la stânga la dreapta
`||`	Sau logic	de la stânga la dreapta
`?:`	Expresie condiţională	de la **dreapta la stânga**
`= *= /= %= += -= <<= >>= &= |= ^=`	Atribuire simplă şi compusă	de la **dreapta la stânga**
`,`	Evaluare secvenţială	de la stânga la dreapta

Operatorii din aceeaşi celulă a tabelului au aceeaşi precedenţă. Se observă că `+` şi `-` unar au o precedenţă mai mare decât operatorii corespunzători binari. Operatorii de atribuire simplă şi compusă au aceeaşi precedenţă însă mai mică decât a majorităţii celorlaţi operatori. Din cauză că precedenţa operatorilor pe biţi este mai mică decât a celor relaţionali, la testarea expresiilor la nivel de bit, trebuie folosite paranteze pentru ca rezultatele să fie corecte:


```
if ((a & MASCA) == x0f) { . . .}
```


NOTA: Într-o expresie se pot folosi parantezele rotunde ( ) pentru a impune o ordine de efectuare a operaţiilor. Dacă expresia are mai multe perechi de paranteze rotunde, evaluarea se începe cu subexpresia cuprinsă între parantezele cele mai interioare.

În C nu este specificată ordinea de evaluare a operanzilor unui operator (excepţie fac operatorii `&&`, `||`, `?:` şi `virgula`); astfel, dacă privim expresia `1/3 + 2*4` din punctul de vedere al operatorului de adunare, nu se ştie dacă mai întâi se evaluează `1/3` sau `2*4`.

4.15 Expresia

Am numit operand o entitate asupra căreia acţionează un operator. Expresia este **o secvenţă de operatori şi operanzi** care efectuează oricare din următoarele trei acţiuni:

- calculează o valoare;
- specifică un obiect sau funcţie;
- generează efecte secundare.

În limbajul C, **operanzii** fac parte din categoriile:
- constante;
- identificatori;
- şiruri;
- apeluri de funcţii;
- expresii de selecţie a membrilor unui tablou;
- expresii complexe formate prin combinarea operanzilor şi operatorilor sau prin cuprinderea operanzilor între paranteze.

Efectele secundare apar când valoarea unei variabile este modificată ca urmare a evaluării unei expresii. Toate operaţiile de atribuire au efecte secundare. Apelurile de funcţii pot avea efecte secundare dacă modifică valoarea unui articol extern vizibil lor, prin atribuire directă sau indirectă prin intermediul unui poantor. De exemplu, efecte scundare apar la următorul apel de funcţie:

```
aduna(i+1, i=j+2);
```

Argumentele unui apel de funcţie pot fi evaluate în orice ordine. Expresia **i+1** poate fi evaluată înainte sau după **i=j+2**, rezultatele funcţiei **aduna()** fiind diferite în fiecare caz.


```
x[i]=i++;
```

În exemplul de mai sus, valoarea lui **x[i]** este imprevizibilă. Lucrurile se vor clarifica mai bine după ce veţi parcurge şi capitolul legat de tablouri. Până atunci e suficient să ştiţi că forma de scriere **x[i]** extrage al **i**-elea element al tabloului **x**. Indicele poate lua fie valoarea nouă a lui **i**, fie cea veche. Practic, rezultatul poate varia în funcţie de compilatorul folosit sau de nivelul de optimizare activat pentru compilatorul curent.

În documentaţiile tehnice ale limbajului C veţi întâlni noţiunea de sequence point, adică **punct de secvenţă**. Voi încerca să lămuresc pe scurt această noţiune. Un punct de secvenţă este un punct aflat la sfârşitul unei expresii complete sau la atingerea operatorilor **||**, **&&**, **?:** , **virgulă** sau **înaintea apelului unei funcţii**, până la care este garantat că toate efectele secundare sunt terminate. În documentaţia ANSI/ISO C există fraza: "Între punctul de secvenţă anterior şi cel curent, unui obiect i se va modifica valoarea stocată cel mult o dată prin evaluarea expresiei. În plus, valoarea anterioară va fi referită numai pentru a determina valoarea de stocat". A doua propoziţie este cam criptică şi spune că: dacă un obiect este scris în interiorul unei expresii, orice referinţă la acesta înăuntrul expresiei din care face parte se face în scopul calculării valorii care urmează să fie stocată. Această regulă constrânge expresiile legale la cele în care referirea precede modificarea valorii.

Începătorii într-ale limbajului C nu fac diferenţa între termenii "comportament definit prin implementare", "comportament nespecificat" şi "comportament nedefinit". Deşi am

discutat deja semnificaţia acestor tremeni, să vedem ce se petrece în cazul particular care urmează. Comportamentul definit prin implementare presupune că implementarea compilatorului utilizat să selecteze un comportament care trebuie să fie documentat. Comportamentul nespecificat presupune ca implementarea compilatorului să selecteze un comportament care nu este neapărat documentat. Nedefinit înseamnă că orice se poate întâmpla. Pentru că **standardul** nu impune comportamentul compilatorului în cazul unor situaţii din categoria nedefinită, compilatorul poate să facă orice. În caz particular nu există nici o garanţie că restul programului va funcţiona normal. Cazul pe care îl prezint în continuare face parte din categoria celor nedefinite. Fie liniile de program:


```
int a =7;
printf("%i\n",a++*a++);
```


Răspunsul afişat de compilatorul folosit de mine este `56`. Când ordinea de evaluare contează, fără să fie clar definită (adică nu putem spune exact ordinea în care compilatorul va evalua diferitele părţi ale expresiei), spunem că semnificaţia expresiei este nedefinită. Scrierea acestor expresii este de evitat. Totuşi, amatorii le folosesc pentru că uneori compilatorul le evaluează în ordinea gândită de ei, iar aceştia presupun că totul este în regulă din moment ce evaluarea funcţionează conform aşteptărilor. Pentru cazul exemplului anterior, răspunsurile posibile teoretic ar fi fost `7*8` şi `8*7` care dau `56`. Un alt răspuns posibil putea fi `49`, în cazul în care compilatorul folosit ar fi ales să facă înmulţirea mai întâi şi apoi incrementările (alegerea ar fi fost corectă pentru că punctul de secvenţă apare pentru expresia `a++*a++` numai după ultimul `a++` unde expresia se termină). Postfix `++` înseamnă că se face incrementarea mai târziu, compilatorul nefiind obligat să o facă imediat. Pentru că în ANSI C nu se specifică modul de tratare a acestor cazuri, teoretic, codul de mai sus putea să afişeze `0` sau `34` sau `56465` sau să ducă la blocarea calculatorului.

Programatorii greşesc de multe ori încercând să facă prea multe într-o singură expresie, prognozând comportametul compilatorului pe baza ordinii de evaluare a operatorilor. De exemplu, ştim că operatorul de înmulţire (`*`) are precedenţă mai mare decât cel de adunare (`+`). Astfel, în expresia `a + b * c`, mai întâi se înmulţeşte `b` cu `c`, apoi rezultatul se adună la `a`. De multe ori zicem pe scurt că înmulţirea se face înaintea adunării. În acest caz este chiar adevărat, dar pentru expresia `a++ + b++ * c++` cele afirmate înainte nu ne mai ajută. Aici, pe lângă înmulţire şi adunare, toate cele trei variabile sunt şi incrementate. Nu putem şti care dintre ele va fi incrementată prima; totul depinde de compilator (de exemplu, compilatorul poate stoca valoarea lui `a` într-un registru şi s-o incrementeze imediat, deşi va trebui să reţină valoarea veche până după realizarea înmulţirii). În final, mai trebuie să ştiţi că parantezele nu dictează ordinea de evaluare globală mai mult decât o face precedenţa. Parantezele sunt mai tari şi suprascriu precedenţa forţând care operanzi sunt legaţi de care operatori. Din acest motiv putem spune că afectează semnificaţia generală a expresiilor, dar nu spun nimic despre ordinea de evaluare a subexpresiilor şi de rezolvare a efectelor secundare. Deci, nu se poate "forţa" ordinea evaluării unor subexpresii din expresia dată ca exemplu prin adăugarea de paranteze sub forma `a++ + (b++ * c++)`. Nici pentru această formă de scriere nu se va şti care dintre incrementări va fi efectuată prima. Parantezele vor forţa ca multiplicarea să aibă loc înainte de adunare, precedenţa ar fi forţat oricum această ordine de evaluare.

Morala este că scrierea unui cod care depinde de ordinea de evaluare este o experienţă ce trebuie depăşită. Astfel de expresii se vor evalua "corect" numai sub anumite compilatoare şi se justifică numai pentru a satisface curiozitatea programatorului cu privire la aceste expresii "interzise". Dacă nu este clar cum se va realiza ceva de un anumit compilator singura şansă este evitarea capcanei.

Pentru a uşura scrierea formulelor matematice în limbajul C prezint în continuare un tabel cu exemple de transcriere a unor astfel de formule în expresii ale limbajul C.

Formulă matematică	Scriere în C
ab	`a*b`
abc	`a*b*c`
$\frac{a}{b}$	`a/b`
$\frac{a}{bc}$	`a/b/c` sau `a/(b*c)`
$\frac{a+d}{b+c}$	`(a+d)/(b+c)`
$\frac{a^4+2a^2+1}{b\cos(x)+c}$	`(a*a*a*a+2*a*a+1)/(b*cos(x)+c)` sau `(pow(a,4.)+2.*pow(a,2.)+1.)/(b*cos(x)+c)`
$\frac{e^x+\ln(x)}{\sqrt{b+c}}$	`(exp(x)+log(x))/sqrt(b+c)`
$e^x+\frac{\ln(x)}{\sqrt{b+c}}$	`exp(x)+log(x)/sqrt(b+c)`
$e^x+\frac{\ln(\lvert x+1\rvert)}{\sqrt{b+c}}$	`exp(x)+log(fabs(x+1))/sqrt(b+c)`

În expresiile prezentate mai sus intervin funcţii din biblioteca standard a limbajului C. Descrierea acestora este prezentată în paragraful 14.5. Tot acolo sunt explicate şi condiţiile în care pot fi folosite aceste funcţii, adică valorile permise ale argumentelor şi valorile întoarse de către acestea.

5 Instrucţiunile limbajului C

Instrucţiunile sunt cele prin care se controlează execuţia programului. În limbajul C, care face parte din clasa celor imperative, problemele se rezolvă prin descrierea algoritmului de rezolvare a problemei în termenii instrucţiunilor limbajului. Deci, pentru a putea scrie un program C, trebuie să ştim algoritmul de rezolvare a problemei, apoi să fim suficient de bine familiarizaţi cu instrucţiunile limbajului C pentru a descrie algoritmul folosind aceste instrucţiuni, operaţie numită şi programare sau scriere a programului. Există câteva caracteristici generale pentru rezolvarea problemelor, care pleacă de la presupunerea că soluţiile se obţin prin manipularea datelor. Iată cam cât trebuie să ştim din C pentru a scrie un program:

1. citirea datelor în program - **operații de intrare**;
2. stocarea datelor în program - **tipuri de date și declarații**;
3. instrucţiuni de prelucrare, manipulare a datelor - **instrucțiuni C** care depind de tipul problemei de rezolvat;
4. afişarea rezultatelor calculate - **operații de ieșire**.

Instrucţiunile sunt organizate astfel încât:

a) unele sunt executate condiţionat, adică numai dacă o anumită condiţie sau o mulţime de condiţii sunt adevărate şi se numesc **instrucțiuni condiționale** (**if**, **switch**);

b) unele sunt repetate, atât timp cât o anumită condiţie este adevărată şi se numesc **instrucțiuni de ciclare** (**while**, **do-while**, **for**);

c) altele sunt grupate sub un singur nume şi pot fi executate, prin specificarea acelui nume din diferite locuri ale programului şi se numesc **funcții**.

Instrucţiunile au efecte şi nu întorc valori. Ele sunt executate secvenţial în ordinea scrierii lor în program. Unele instrucţiuni, prin efectul lor particular, pot întrerupe execuţia secvenţială a programului producând salturi între diferite porţiuni de program (**goto**, **continue**, **break**, **return**).

5.1 Instrucțiunea expresie și blocul

Expresiile prezentate până acum sau **printf()** şi **scanf()** devin instrucţiuni dacă sunt urmate de caracterul punct şi virgulă (;). În limbajul C ; poartă denumirea de **terminator de instrucțiuni**. Acesta marchează terminarea unei instrucţiuni şi are utilizări multiple. Aţi văzut deja că este folosit inclusiv la terminarea declaraţiilor, adică este şi **terminator de declarații**.

Formă:	**expresie;**

O expresie urmată de ; se numeşte **instrucțiune expresie**. Majoritatea instrucţiunilor

expresie C sunt atribuiri sau apeluri de funcţii. Toate efectele secundare ale instrucţiunii expresie sunt terminate înainte de executarea instrucţiunii următoare. Dacă expresia lipseşte, construcţia se numeşte **instrucţiune vidă**.

Instrucţiuni expresie C sunt:


```
1 a=0;
2 b=b+1;
3 printf("Te salut!\n");
```


Expresiile şi instrucţiunile expresie pot să fie complicate. Pentru ca o expresie să fie utilă trebuie să modifice spaţiul de date al programului (denumirea formală este "trebuie să producă efecte secundare"). **Liniile 1** şi **2** sunt exemple de instrucţiuni expresie corecte pentru că dau valori noi variabilelor **a** şi **b**. La fel, **linia 3** este un apel de funcţie, fiind şi ea o instrucţiune expresie corectă. Instrucţiuni expresie "degenerate" ale celor de mai sus ar putea fi:


```
0;
b+1;
```

ANTAL Tiberiu Alexandru
tel.: 0040-264-530918
e-mail: antaltiberiu@xnet.ro

Ele sunt sintactic corecte, dar fiecare calculează o valoare fără ca apoi să mai facă ceva cu ea. Din acest motiv valorile calculate se pierd, iar instrucţiunile sunt inutile. Cazul "degenerat" prezintă instrucţiuni expresie care nu produceau efecte secundare. Este posibil şi opusul acestei situaţii şi anume ca o expresie să aibă efecte secundare multiple (vezi **a++*a++**). În acest caz expresia este fie confuză prin rezultatul întors, fie nedefinită.

Acoladele ({ }) sunt folosite pentru a grupa o secvenţă de instrucţiuni şi declaraţii într-o **instrucţiune compusă** care mai poartă şi denumirea de **bloc**. Prin această grupare blocul devine sintactic echivalentul unei singure instrucţiuni.

Formă:

```
{
        instrucţiune1
        ...
        instrucţiunen
}
```


Deci acest grup de instrucţiuni cuprins între acolade se va putea scrie oriunde C permite scrierea unei singure instrucţiuni. Un astfel de exemplu sunt acoladele între care se scriu instrucţiunile care formează corpul funcţiei **main()**. Nu se pune ; după acolada de închidere (}) a unui bloc.

5.2 Instrucţiunea if-else

În sfârşit, am ajuns la descrierea construcţiei **if-else** pe care am utilizat-o deja în capitolul precedent. Se foloseşte pentru **luarea unei decizii** în mersul logic al rezolvării problemei, fiind modul tradiţional de programare a unei întrebări de tipul **dacă-atunci** sau **în cazul în care-atunci**.

Formă: `if (expresie)`

```
                instrucţiune1
        else
                instrucţiune2
```

expresie este o expresie care poate fi de tipul aritmetic sau poantor. Dacă **expresie** este nenulă (!= 0), adică **adevărată**, se execută **instrucţiune1**. Altfel (== 0) se execută **instrucţiune2**. Porţiunea else **instrucţiune2**; este opţională. Astfel, o formă validă este şi:

```
Formă:  if (expresie)
                instrucţiune1
```

În acest caz, **instrucţiune1** se execută dacă şi numai dacă **expresie** este nenulă. Dacă este nulă se trece peste **instrucţiune1** şi programul continuă cu instrucţiunea următoare.

if

Întrucât **if** testează numai valoarea numerică pentru **expresie**, în loc de **if (expresie != 0)** se poate scrie prescurtat **if (expresie)**. Câteva exemple de utilizare ale lui **if** sunt prezentate în continuare:

```
1  /* IF1.C */
2  int main()
3  {
4          int   a1 = 11, a2 = 11, a3 = 11;
5          char  c1 = 40, c2 = 40, c3 = 40;
6          float x1 = 12.987, x2 = 12.987, x3 = 12.987;
7
8  /* Primul grup de instructiuni de comparare */
9          if (a1 == a2) a3 = -13;  /* a3 = -13          */
10         if (a1 > a3)  c1 = 'A';  /* c1 = 65           */
11         if (!(a1 > a3)) c1 = 'B';/* c1 nu se modifica */
12         if (c2 <= c3) x1 = 0.0;  /* x1 = 0.0          */
13         if (x1 != x2) x3 = c3/2; /* x3 = 20           */
14
15 /* Al doilea grup de instrucţiuni de comparare */
16         if (a1 = (x1 != x2)) a3 = 1000; /* a1 = un numar pozitiv oarecare si  a3 = 1000  */
17         if (a1 = a2) a3 = 222;  /* a1 = a2 Şi a3 = 222 */
18         if (a1 != 0) a3 = 333;  /* a3 = 333            */
19         if (a1) a3 = 444;       /* a3 = 444            */
20
21 /* Al treilea grup de instructiuni de comparare */
22         a1 = a2 = a3 = 77;
23         if ((a1 == a2) && (a1 == 77)) a3 = 33; /* a3 = 33 */
24         if ((a1 > a2) || (a3 > 12))   a3 = 22; /* a3 = 22 */
25         if (a1 && a2 && a3) a3 = 11;           /* a3 = 11 */
26         if ((a1 = 1) && (a2 = 2) && (a3 = 3)) x1 = 12.00; /* a1 = 1, a2 = 2, a3 = 3, x1 = 12.00 */
27         if ((a1 == 2) && (a2 = 3) && (a3 = 4)) x1 = 14.56; /* x1 nu se modifica */
28
29 /* Al patrulea grup de comparatii */
30         if (a1 == a1) a3 = 27.345; /* a3 se modifica intotdeauna */
31         if (a1 != a1) a3 = 27.345;  /* a3 nu se modifica        */
32         if (a1 = 0) a3 = 27.345;    /* a1 = 0, a3 nemodificat   */
33
34         return 0;
35 }
```

Observaţi că instrucţiunile **if** prezentate se termină în ;. Asta pentru că gramatical, după condiţia de test din **if**, urmează o instrucţiune. Deoarece instrucţiunile din exemplul anterior sunt instrucţiuni expresie, acestea trebuie să se termine în ;.

if-else

if execută o singură instrucţiune în funcţie de rezultatul deciziei. Dacă doriţi să se execute mai multe instrucţiuni trebuie să folosiţi **instrucţiunea compusă**.


```
1  if (b == 0.0)
2          printf("Fractia e nedefinita\n");
3  else
4  {
5          f=a/b;
6          printf("Fractia e %f\n", f);
7  }
```


Mai sus, dacă **b** are o valoare nenulă, se execută tot grupul de instrucţiuni cuprinse între acoladele din **liniile 4** şi **7**.

Suprapunerea **if**-urilor

C permite ca un **if** să conţină pe locul instrucţiunilor un alt **if**. O astfel de secvenţă de **if**-uri se numesc **imbricate** sau **suprapuse**. Fie secvenţa:

```
if (expresie1)
        instrucţiune1
else if (expresie2)
        instrucţiune2
else if (expresie3)
        instrucţiune3
else
        instrucţiune4
```


ea este caracteristică unei multi-decizii. Expresiile **expresie1**, **expresie2** şi **expresie3** sunt evaluate în ordine astfel: dacă **expresie1** este **adevărată** se execută **instrucţiune1** şi secvenţa de **if**-uri se termină; altfel se execută ramura **else**, care conţine un nou **if** unde decizia se ia pe baza lui **expresie2** etc. **instrucţiunex** poate să fie o singură instrucţiune sau una compusă. Ultimul **else** tratează cazul "nici una dintre expresiile anterioare nu este **adevărată**" **instrucţiune4** este atinsă în cazul când nici una dintre condiţiile puse nu au fost satisfăcute. Să vedem ce se petrece însă pentru o situaţie de forma:


```
if (expresie1)
        if (expresie2)
                instrucţiune1
        else
                instrucţiune2
```


În acest caz se pune problema apartenenţei lui **else**. C rezolvă această ambiguitate asociindu-l pe **else** celui mai apropiat **if**. Conform acestei reguli, **else** este asociat lui **if** (**expresie2**). Dacă această situaţie nu ne convine trebuie să folosim acoladele după cum se prezintă în exemplul care urmează:

```
int a=7;
if (a>0)
```

```
int a=7;
if (a>0)
```

```
if (a < 100)
  printf("a in (1, 100)\n);
else
  printf("a in afara lui (0, 100)\n");
```

```
{
 if (a < 100)
   printf("a in (1, 100)\n);
}
else
 printf("a este negativ\n");
```

Cazuri clasice de utilizare

Pentru a verifica dacă o variabilă `a` este într-un domeniu dat $[a_{min}, a_{max}]$ vom folosi `if` sub forma: `if (a>= a`$_{min}$ `&& a <= a`$_{max}$`) . . .`, iar condiţia scrisă se citeşte astfel "dacă `a` este mai mare sau egal cu a_{min} şi mai mic sau egal cu a_{max}". Începătorii într-ale C vor fi tentaţi să pună problema de mai sus sub forma: `if (a`$_{min}$ `<= a <= a`$_{max}$`) . . .` Această expresie este analizată de compilatorul C la fel cu expresia a_{min} `+ a +` a_{max}. Adică se adună a_{min} la `a`, apoi la rezultat se mai adună a_{max}. Folosind parantezele rotunde ordinea de rezolvare a operatorilor poate fi pusă într-o formă didactică astfel, `(a`$_{min}$ `+ a) + a`$_{max}$. Similar, expresia a_{min}`<= a <= a`$_{max}$ este echivalentă cu expresia `(a`$_{min}$ `<= a) <= a`$_{max}$. Se verifică dacă a_{min} este mai mic sau egal cu `a`, rezultatul este 0 sau 1 după cum relaţia este `falsă` sau `adevărată`. Apoi se compară valoarea numerică de 0 sau 1 - în funcţie de rezultat - cu a_{max}, iar rezultatul este valoarea 1 sau 0, după cum relaţia este `adevărată` sau `falsă`. Se observă că la a doua comparaţie nu mai participă valoarea variabilei `a`, ci întotdeauna una din valoarile numerice de 0 sau 1, rezultatele celei de a doua comparaţii fiind altceva decât ne-am propus noi iniţial- pentru că nu participă `a`-ul ca operand.

La realizarea unei operaţii, în funcţie de îndeplinirea unei condiţii, `if` se scrie în mod tipic sub forma: `if (a > maxim) . . .` Operatorul `>` întoarce valoarea 1 dacă `a` este mai mare decât `maxim` sau 0 altfel. `if` îl interpretează pe 0 ca `fals` şi pe 1 ca `adevărat`.

Conf. dr. ing. ANTAL Tiberiu Alexandru
Universitatea Tenhică din Cluj-Napoca
Facultatea Construcţii de Maşini
Catedra de Mecanică şi Programare
Curs de limbaj C
Copyright 2001. Toate drepturile sunt rezervate autorului.
Multiplicarea acestui document în scop comercial este interzisă.
Sudenţii participanţi la orice formă de învăţământ superior bugetar pot multiplica acest document pentru uzul personal.
Sudenţii participanţi la orice formă de învăţământ cu plată sau alte persoane doritoare
ANTAL Tiberiu Alexandru
tel.: 0040-264-530918
e-mail: antaltiberiu@pcnet.ro

5.3 Instrucţiunea switch

`switch` este o variantă mai flexibilă decâ `if pentru cazul multi-deciziilor.`

```
Formă:      switch(expresie)
            {
                  case expr-const1 : instrucţiuni1
                  case expr-const2 : instrucţiuni2
                  . . .
                  default: instrucţiuni3
            }
```

Mai întâi se testează valoarea lui `expresie`; dacă aceasta este egală cu una dintre valorile constantelor întregi `expr-consti` se face ramificarea pe cazul corespunzător. Fiecare caz separat este marcat printr-o etichetă formată din unul sau mai mulţi întregi sau printr-o expresie constantă (`expr-consti`). În caz de egalitate între valoarea lui `expresie` şi cea constantă din `case` se continuă execuţia programului, începând cu instrucţiunile (`instrucţiuni1, instrucţiuni2` etc.) acelui caz. Grupul `instrucţiuni3` marcat cu eticheta `default:` sunt executate dacă nu s-a găsit nici o egalitate. Porţiunea `default:` este opţională; dacă aceasta nu există nu se face nimic. În secvenţa de program care urmează se pune problema selectării unei variante pe baza conţinutului variabilei `cmd`. Implementările sunt scrise folosind atât `if` cât şi `switch`. `fisiere()`, `program()`, `compilare()`, `optiuni()` şi `alte_comenzi()` sunt nume de funcţii. Implementarea cu `switch` arată modul în care se

poate executa acelaşi grup de instrucţiuni pentru mai multe etichete diferite (cazul funcţiei **fisiere**() care este selectată la apăsarea literelor **f** mic şi **F** mare).

```
if ((cmd =='F') || (cmd =='f')) fisiere();
  else if (cmd == 'R') program();
    else if (cmd == 'C') compilare();
      else if (cmd == 'O') optiuni();
        else alte_comenzi(cmd);
```

```
switch(cmd)
{
 case 'F': case 'f': fisiere();break
 case 'R': program();break;
 case 'C': compilare();break;
 case 'O': optiuni;break;
 default: alte_comenzi(cmd);
}
```

Din punctul de vedere al tipului expresiei de testat, **switch** este mai puţin flexibil decât **if**. La **switch** se pot testa numai valori de tipul întreg, iar valoarea expresiei de testat se poate compara numai cu constante. La **if** se pot testa şi valori de tip real, iar în expresia de testat pot interveni atât variabile cât şi constante. În exemplul prezentat mai sus observaţi o instrucţiune neprezentată încă, **break**. Pentru a înţelege rolul ei, fără a spune mai multe despre ea, prezint programul următor:

```
1  /* SWITCH.C */
2  #include<stdio.h>
3
4  enum zile {luni=1, marti, miercuri,joi,vineri,simbata, duminica};
5
6  int main(void)
7  {
8          enum zile azi;
9
10         printf("Baga un numar intre 1 si 7 = ");
11         scanf("%i",&azi);
12         switch(azi)
13         {
14                 case luni:
15                 case marti:
16                 case miercuri:
17                 case joi:
18                 case vineri: puts("Cara-te la lucru ...!");break;
19                 case simbata: printf("Relaxeaza-te si ");
20                 case duminica: printf("fa o excursie !");break;
21                 default: printf("Nu exista asa o zi !");
22         }
23         return(0);
24 }
25
```

Rezultate:

```
Baga un numar intre 1 si 7 = 3
Cara-te la lucru ...!
```

```
Baga un numar intre 1 si 7 = 6
Relaxeaza-te si fa o excursie !
```

```
Baga un numar intre 1 si 7 = 7
fa o excursie !
```

Am spus că la întâlnirea "egalităţii" **switch** face ca programul să se continue din acel punct, **break** întrerupe această continuare firească forţând saltul la acolada din **linia 23**, iar execuţia programului continuă cu următoarea linie de program după această acoladă (mai sus, **linia 24**).

5.4 Instrucţiuni de ciclare

Instrucţiunile de ciclare sau mai pe scurt ciclurile asigură posibilitatea execuţiei unei instrucţiuni în mod repetat. Toate ciclurile sunt formate din două părţi: una sau mai multe expresii care controlează reluarea ciclului şi corpul ciclului, adică instrucţiunea sau grupul de instrucţiuni a cărei execuţie se reia. C are trei instrucţiuni de ciclare: **`while`**, **`do-while`** şi **`for`**.

5.4.1 Instrucţiunea de ciclare `while`

`while` este ciclul de bază în C. Are o singură expresie de control - **`expresie`** - şi execută reluarea unei instrucţiunii - **`instrucţiune`** - care se mai numeşte şi **`corpul ciclului`** - atâta timp cât **`expresie`** ia valoarea de adevăr **`adevărat`**.

```
Forma:      while (expresie)
                instructiune
```

Execuţia instrucţiunii **`while`** seamănă cu **`if`**. Dacă condiţia exprimată prin **`expresie`** este **`adevarata`**, se execută **`instrucţiune`**. În continuare, după execuţia lui **`instrucţiune`**, condiţia este testată din nou. Dacă este încă **`adevărată`**, se reia **`instrucţiune`**. Cât timp condiţia rămâne **`adevărată`**, corpul ciclului se reia. Când condiţia devine **`falsă`** (pentru că depinde într-un fel oarecare de valori care sunt modificate în corpul ciclului) corpul ciclului nu se reia, execuţia programului continuând cu prima instrucţiune care urmează după aceasta. Gramatical, **`expresie`** trebuie să fie de tipul aritmetic sau poantor. Secvenţele de evaluare pentru **`expresie`** şi **`instrucţiune`** sunt:

```
expresie
instrucţiune
...
expresie
```

Cazuri clasice de utilizare

Reluarea unui grup de instrucţiuni atât timp cât o condiţie este adevărată este aplicaţia clasică a ciclului **`while`**. În exemplul următor, instrucţiunea reluată este compusă şi corespunde liniilor de program în domeniul **`7-10`**. Atâta timp cât valoarea variabilei **`i`** este mai mică decât **`6`**, instrucţiunile care alcătuiesc instrucţiunea compusă a corpului lui **`while`** se reiau.

```
1  /* WHILE.C */
2  #include <stdio.h>
3  int main(void)
4  {
5          int i = 0;
6          while (i < 6)
7          {
8                  printf("Valoarea lui i este %i\n", i);
9                  i++;
10         }
11         return 0;
12 }
```

Rezultate:

```
Valoarea lui i este 0
```

```
Valoarea lui i este 1
Valoarea lui i este 2
Valoarea lui i este 3
Valoarea lui i este 4
Valoarea lui i este 5
```


Variante de scriere a programului anterior sunt şi:

```
#include <stdio.h>
int main(void)
{
        int i = 0;
        while (i < 6)
        {
                printf("i este %i\n", i);
                i=i+1;
        }
        return 0;
}
```

```
#include <stdio.h>
int main(void)
{
        int i = 0;
        while (i < 6)
                printf("i este %i\n", i++);
        return 0;
}
```

De evitat

Dacă în exemplul de mai sus puneam caracterul ; după **`linia 6`** (adică, scriem **`while(i < 6);`**) ciclul **`while`** va fi "executat la infinit". Prin definiţia gramaticală C aşteaptă ca **`while`** să se încheie într-o instrucţiune care este corpul ciclului. Caracterul ;, numit **`instrucţiunea vidă`** satisface această cerinţă. Creşterea valorii lui **`i`** în corpul ciclului este esenţială pentru atingerea condiţiei de ieşire din ciclu. Scris în această formă, ciclul **`while`** se termină după caracterul ;. Instrucţiunea compusă în care se modifică valoarea lui **`i`** nu mai face parte din **`while`**. Deci, valoarea lui **`i`** nu se modifică rămând **`0`**, iar condiţia de părăsire a ciclului **`(i>=6)`** nu se mai îndeplineşte. C va testa la infinit îndeplinirea condiţiei de ieşire din ciclu.

Experienţa arată că cele mai frecvente erori apar la punerea condiţiei de părăsire a ciclului. De exemplu, să presupunem că doriţi să afişati valoarea lui **`i`** cât timp este mai mică sau egală cu **`7`** şi scrieţi programul:

```
1  /* WHILE1.C */
2  #include<stdio.h>
3
4  int main(void)
5  {
6          int i=1;
7          printf("am inceput\n");
8          while (i == 7)
9                  printf("i = %i\n", i++);
10         printf("s-a terminat\n");
11 }
```


Rezultate:

```
am inceput
s-a terminat
```

Programul de mai sus este greşit! Trebuie să reţineţi că este esenţială scrierea corectă a condiţiei de părăsire a ciclului. În cazul de mai sus condiţia **`i == 7`** este **`falsă`** de la început, motiv pentru care se sare peste corpul lui **`while`** şi se continuă programul cu următoarea instrucţiune (**`linia 11`**). Ciclul **`while`** infinit se poate obţine şi din neatenţie. Dacă în programul **`WHILE.C linia 6`** se va scrie sub forma **`while (i < 6);`**, între caracterul **`)`** - paranteza rotundă închisă- şi caracterul **`;`** - punct şi virgulă - se formează corpul ciclului.

Din această cauză valoarea `i`-ului nu se mai incrementează, iar condiţia de părăsire a ciclului nu se îndeplineşte. Ocazional, vom dori să nu se facă nimic în corpul ciclului `while`. Un astfel de exemplu este linia `while ((ch = getchar()) != '\n');`. Aici se citeşte un caracter cu funcţia `getchar()` care se stochează în `ch` şi se verifică dacă rezultatul este diferit de caracterul `'\n'`. Deci acest ciclu va relua citirea unui caracter de la tastatură atâta timp cât caracterul este diferit de `'\n'`. Putem spune că secvenţa de program aşteaptă introducerea lui `'\n'` - apăsarea tastei <**Enter**> - pentru ca programul să treacă mai departe la instrucţiunea următoare.

5.4.2 Instrucţiunea de ciclare do-while

Cea de a doua instrucţiune de ciclare este `do-while`. Ea testează condiţia de terminare a ciclului după executarea instrucţiunii. `while` testează această condiţie înainte de execuţia instrucţiunii care formează corpul ciclului. Din acest punct de vedere putem spune că `do-while` este un `while` cu "susul în jos".

Formă:
```
do
   instrucţiune
while (expresie);
```

Datorită testării condiţiei de reluare a ciclului după execuţia lui `instrucţiune`, corpul acestui ciclul se va executa cel puţin o dată. Dacă `expresie` este `falsă` corpul nu se reia. Secvenţele de evaluare ale lui `expresie` şi `instrucţiune` sunt:

```
instrucţiune
expresie
...
expresie
```

```
1  /* DO-WHILE.C */
2  #include<stdio.h>
3  int main()
4  {
5          int a=3;
6
7          printf("inceput\n");
8          do {
9                  printf("\ta = %i\n",a);
10                 a--;
11         } while (a > 0);
12         printf("sfirsit\n");
13
14         return 0;
15 }
```

Rezultate:

```
inceput
        a = 3
        a = 2
        a = 1
sfirsit
```

În `linia 5` se iniţializează valoarea lui `a` cu `3`, `linia 10` face decrementarea lui `a`, iar expresia

testată este `a > 0` din **linia 11**. Corpul ciclului este format din instrucţiunile **liniilor 8-11** cuprinse între acolade.

Un exemplu

În practică ciclul **do-while** este mai puţin folosit decât cel **while**. Există însă cazuri când este necesar sau cel puţin convenabil de utilizat. Să presupunem că dorim să scriem un program care citeşte de la tastatură un număr întreg, apoi afişează numărul de cifre şi semnul numărului. Pentru că un număr are cel puţin o cifră, este convenabil ca execuţia corpului ciclului să se facă cel puţin o dată. Programul care urmează implementează o astfel de rezolvare.

```
1  /* DO-WHILE1.C */
2  #include<stdio.h>
3  int main(void)
4  {
5          int a, c=0;
6          char semn;
7
8          printf("a = ");
9          scanf("%i",&a);
10         printf("%i are ", a);
11
12         if (semn = (a < 0)) a=-a;
13
14         do
15                 c++;
16         while ((a/=10)!=0);
17
18         printf("%i cifre si este %s\n", c, semn ? "negativ" : "pozitiv");
19
20         return 0;
21 }
```

Rezultat:

```
a = 5839
5839 are 4 cifre si este pozitiv
```


5.4.3 Instrucţiunea de ciclare for

for este sintactic cel mai complicat dintre cele trei cicluri. În esenţă este asemănator cu **while**, având chiar o condiţie de terminare identică cu cea de la **while**. **for** este cea mai concisă formă de ciclu care reuneşte condiţiile de început, condiţia de terminare împreună cu instrucţiunile de avans ce trebuie parcurse înainte de reluarea execuţiei corpului ciclului.

Formă:
```
for (expresie1; expresie2; expresie3)
        instrucţiune
```

Gramatical, cele trei componente ale lui **for** sunt expresii. În cazurile comune **expresie1** şi **expresie3** sunt atribuiri, iar **expresie2** este o expresie relaţională. Oricare dintre cele trei expresii pot fi omise, însă caracterele ; trebuie scrise. Caracterele ; - punct şi virgulă - separă cele trei expresii şi nu au nimic în comun cu terminatorul de instrucţiuni C. Când compilatorul întâlneşte un ciclu **for**, va evalua la început pe **expresie1**. Apoi, pe **expresie2**, iar dacă este **adevărată**, se execută corpul ciclului (**instrucţiune**). Se continuă cu

expresie3 pentru a trece la pasul următor, apoi **expresie2** se evaluează din nou pentru a verifica dacă există pasul următor. Pe timpul execuţiei ciclului **for**, secvenţele de evaluare ale expresiilor sunt:

```
expresie1
expresie2
instrucţiune
expresie3
expresie2
instrucţiune
expresie3
. . .
expresie2
instrucţiune
expresie3
expresie2
```


expresie1 se evaluează o singură dată. **expresie3** se evaluează după fiecare reluare a corpului ciclului (**instrucţiune**). Ultima expresie care se va evalua este **expresie2** pentru că atunci când ea devine **falsă**, ciclul se termină.

Comparaţie **while - for**

Construcţia:

```
for (exp_iniţializare; exp_terminare; exp_avans)
    instrucţiune;
```

este echivalentă cu:

```
exp_iniţializare;
while(exp_terminare)
{
instrucţiune;
exp_avans;
}
```


Comparaţie C-BASIC-FORTRAN-Pascal

Este utilă comparaţia ciclului **for** cu echivalenţii lui din alte limbaje de programare. Ciclul **for** prezentat în tabelul care urmează este "aproximativ echivalent" cu următoarele instrucţiuni de ciclare din limbajele BASIC, FORTRAN şi PASCAL:

Limbaj de programare	Formă de scriere
C	`for (i=a; i<= b; i=i+p)`
BASIC	`for i=a to b step p`
FORTRAN	`do 7 i=a,b,p`
Pascal	`for i:=a to b`

Unele limbaje de programare cum sunt Pascal şi Ada permit numai incrementarea sau decrementarea variabilei de control a ciclului - **i** în exemplele de mai sus. Dacă se doreşte modificarea cu o valoare diferită de **1** a variabilei de control, trebuie folosit un ciclu de tipul **while**. Altele cum este FORTRAN, permit ca aceeaşi variabilă de control să fie numai de tipul întreg. Din această prismă C este cel mai puţin restrictiv pentru că nu are "o variabilă de control a ciclului", toate cele trei componente ale ciclului **for** fiind expresii arbitrare. În C, spre deosebire de FORTRAN, dacă **expresie2** este **falsă** înainte de prima

execuţie a corpului ciclului (**instrucţiune**), acesta nu se va parcurge. În C, spre deosebire de Pascal, o variabilă folosită pentru controlul execuţiei ciclului îşi păstrează valoarea finală după terminarea ciclului. La terminarea unui ciclu, datorită realizării condiţiei de părăsire a ciclului - condiţia testată devine **falsă** - valoarea variabilei de control, în C, după ciclu va fi prima pentru care condiţia a fost **falsă** şi nu ultima pentru care a reuşit. De asemenea, este legală modificarea acestei variabile în interiorul ciclului şi a valorii finale **b** care controlează terminarea ciclului. Pentru exemplificarea unui ciclu **for**, unde "pasul" de parcurgere este diferit de **1**, prezint un program care calculează valoarea funcţiei cosinus în intervalul [0, π] cu pasul de **0.1** radiani.

```
1  /* FOR.C */
2  #include<stdio.h>
3  #include<math.h>
4
5  #define PI 3.141592553
6  int main (void)
7  {
8          double unghi;
9
10         for(unghi=0.0; unghi <= PI; unghi+=0.1)
11                 printf("cos(%3.2lg) = %lf\n", unghi, cos(unghi));
12
13         return 0;
14 }
```

Rezultate:

```
. . .
cos(2.8) = -0.942222
cos(2.9) = -0.970958
cos( 3) = -0.989992
cos(3.1) = -0.999135
```

+= se foloseşte pentru a creşte valoarea variabilei **unghi** cu **0.1** la fiecare reluare a ciclului. În **linia 3**, **math.h**, este antetul standard pentru declaraţiile funcţiilor matematice, în particular, pentru funcţia **cos()**.

Extinderea ciclului for

Ciclul **for** pare a fi ideal atâta timp cât avem o singură expresie de iniţializare şi una singură pentru avans. Dacă două sau mai multe astfel de expresii ar fi necesar de executat, acolo unde **for** permite numai una singură, s-ar părea că ar fi nevoie de o altă instrucţiune C. Problema poate fi uşor depăşită prin folosirea operatorului virgulă. El garantează evaluarea secvenţială a expresiilor de la stânga la dreapta, valoarea şi tipul expresiei fiind date de cea mai din dreapta. Fie codul **for(a=1, b=2, i=0; a=b=5, i<10; a++, b--, i++), expresie1** este formată din **a=1, b=2, i=0**. Aici, mai întâi se execută **a=1**, apoi **b=2** şi în final **i=0**. Dacă într-un caz nu avem nevoie de **expresie1** şi **expresie3**, este corect să scriem un **for** de forma: **for(; i<7;)** care este echivalent cu **while (i<7)**.

Ciclul for infinit

Dacă se omite **expresie2**, implicit se presupune că aceasta are valoarea 1 (**adevărat**), deci ciclul nu se va termina. Astfel, o construcţie de forma **for(;;)** produce un ciclu for infinit. O diferenţă între **while** şi **for** este că la **for**, **expresie2** este opţională, pe când la **while**, **expresie** este obligatorie. Omiterea lui **expresie** în ciclul **while** va fi semnalată de compilator ca o eroare de sintaxă. Dacă doriţi în mod deliberat să scrieţi un ciclu **while** infinit folosiţi construcţia **while(1)**.

De ce atâtea instrucţiuni de ciclare?

`for` este de preferat când cele trei expresii manipulează aceeaşi variabilă sau structură de date pentru iniţializare, test de terminare şi avans. Dacă acest şablon de prelucrări nu este respectat, probabil `while` ar oferi o rezolvare mai clară, cel puţin pentru că în cazul lui `for` ne aşteptăm la un tipic descris mai sus, nu şi pentru `while`.

5.5 Instrucţiunea break

Crearea deliberată a unui ciclu infinit pare să fie o acţiune fatală, din moment ce un astfel de program nu se va termina niciodată. Nu şi în C ! Aici, se poate sări în afara oricărui tip de ciclu prin folosirea instrucţiunii `break`. Instrucţiunea `break` termină execuţia celui mai apropiat `do-while`, `for`, `while` sau `switch`. Spun "cel mai apropiat" pentru că este posibilă suprapunerea mai multor cicluri. Efectul este acela de trecere a controlului la următoarea instrucţiune după cea terminată.


```
Formă:      break;
```

`break` este frecvent utilizată în `switch` pentru a termina o ramură particulară. Nu se poate folosi în afara instrucţiunilor de ciclare sau a lui `switch`.

```
1  /* BREAK.C */
2  #include <stdio.h>
3
4  int main(void)
5  {
6    int a;
7    for(a = -2 ; a < 3 ; a++)
8    {
9      if (a == 0)
10       {
11         printf("In corpul ciclului for, IESIRE cu break, a este %d\n", a);
12         break;
13       }
14       printf("In corpul ciclului for, a este %d\n", a);
15   }
16   return 0;
17 }
```


Rezultate:

```
In corpul ciclului for, a este -2
In corpul ciclului for, a este -1
In corpul ciclului for, IESIRE cu break, a este 0
```


Cazuri clasice de utilizare

`break` se utilizează la testarea unor condiţii de terminare excepţională a ciclurilor. Într-un astfel de caz se doreşte saltul în afara corpului ciclului mai devreme decât în cazul realizării condiţiei de părăsire a ciclului - cazul terminării normale.

5.6 Instrucţiunea continue

`break` forţează saltul imediat în afara celui mai apropiat ciclu, `continue` forţează saltul la următoarea reluare a ciclului. Instrucţiunea `continue` transferă controlul execuţiei

programului la începutul ciclurilor **do-while**, **for**, **while** din punctul în care apare, sărind peste instrucţiunile rămase în corpul corespunzător ciclului în care se găseşte. În cazul ciclurilor **while** şi **do-while**, saltul se face la reevaluarea condiţiei de terminare a ciclurilor - **expresie**. În cazul lui **for**, saltul se face la expresia de avans - **expresie3** - apoi se reevaluează condiţia de părăsire a ciclului - **expresie2**.

Formă: `continue;`

Cazuri clasice de utilizare

continue se foloseşte în elaborarea unui program cu cicluri, când dorim să facem salt la începutul ciclului pentru a testa expresia de părăsire a ciclului, eventual pentru a relua corpul acestuia, fără a trece prin toate instrucţiunile din corpul ciclului curent.

```
1  /* CONTINUE.C */
2  #include<stdio.h>
3  #include<math.h>
4
5  int main(void)
6  {
7    int a,b;
8
9    for(a = -2; a < 3; a++)
10     for(b = -2; b < 3; b++)
11     {
12       if (a <= 0) break;
13       if (b == 0) continue;
14       printf("a= %2d, b=%2d, sqrt(a)/b = %f\n", a,b, sqrt(a)/b);
15     }
16   return 0;
17 }
```

Rezultate:

```
a=  1, b=-2, sqrt(a)/b = -0.500000
a=  1, b=-1, sqrt(a)/b = -1.000000
a=  1, b= 1, sqrt(a)/b = 1.000000
a=  1, b= 2, sqrt(a)/b = 0.500000
a=  2, b=-2, sqrt(a)/b = -0.707107
a=  2, b=-1, sqrt(a)/b = -1.414214
a=  2, b= 1, sqrt(a)/b = 1.414214
a=  2, b= 2, sqrt(a)/b = 0.707107
```

În exemplul anterior, expresia $\frac{\sqrt{a}}{b}$ se poate evalua numai dacă **b**$\neq$**0** şi **a**$>$**=0**. În **linia 12**, când condiţia **a** $<$**=0** este adevărată, se sare în afara ciclului **for** interior pentru că nu se poate calcula $\sqrt{a}$. În **linia 13**, când condiţie **b** == **0** este **adevărată**, împărţirea nu se poate efectua - pentru că **b** este zero - şi se trece la o nouă reluare a ciclului **for** interior.

Ce sunt **break** şi **continue**?

Instrucţiunile **break** şi **continue** sunt forme particulare ale instrucţiunii de salt necondiţionat **goto**. Fără a intra în prea multe amănunte, unele firme de software interzic folosirea lui **goto** în programe. Pentru că **break** şi **continue** sunt într-o relaţie foarte apropiată de **goto**, aceleaşi firme propun utilizarea moderată a acestor instrucţiuni. De ce această oroare faţă de **goto**? În 1966, Böhm şi Jacopini publică o lucrare în care arată că orice program poate fi scris

fără utilizarea saltului necondiţionat. Utilizarea lui `goto` îngreunează mult înţelegerea programului şi în general va scădea viteza lui de execuţie. Putem spune deci că un program cu salturi necondiţionate va fi de calitate îndoielnică.

5.7 Instrucţiunea goto

Pentru că C are o instrucţiune de salt necondiţionat `goto` mă simt obligat să o prezint şi pe aceasta.

Formă:
```
goto etichetă;
...
etichetă: instrucţiune
```

Instrucţiunea `goto` face saltul la instrucţiunea `instrucţiune` marcată cu eticheta `etichetă`. În exemplul care urmează, `goto` transferă controlul programului la instrucţiunea etichetată cu `iesire`: când `a` ia valoarea `0`.

```
1  /* GOTO.C */
2  #include <stdio.h>
3
4  int main(void)
5  {
6          int a;
7
8          for(a = -2 ; a < 3 ; a++)
9          {
10         if (a == 0) goto iesire;
11         printf("In corpul ciclului for, a este %d\n", a);
12         }
13 iesire: printf("Am iesit cu goto din for!\n");
14         return 0;
15 }
```

Rezultate:

```
In corpul ciclului for, a este -2
In corpul ciclului for, a este -1
Am iesit cu goto din for!
```

Ce este o `etichetă`?

Un `nume de etichetă` urmează aceleaşi reguli de scriere ca pentru numele de variable. Eticheta `se poate ataşa oricărei instrucţiuni din corpul funcţiei` în care apare instrucţiunea `goto`.

Şi totuşi uneori se foloseşte!

Există situaţii când folosirea unui `goto` este îndreptăţită. Cazul clasic este abandonarea unei construcţii de mai multe cicluri suprapuse la apariţia unei erori fatale în interiorul unuia dintre ciclurile suprapuse. Folosirea lui `break` ar duce la ieşirea numai din cel mai apropiat ciclu, ieşirea directă din toate ciclurile suprapuse fiind imposibilă fără utilizarea lui `goto`.

Universitatea Tenhică din Cluj-Napoca
Facultatea Construcţii de Maşini
Catedra de Mecanică şi Programare
Curs de limbaj C

6 Funcţia

Este ideal ca la scrierea unui program acesta să fie cât mai uşor de înţeles şi de întreţinut. Pentru aceasta programele lungi trebuie transformate în porţiuni de programe mai scurte, de sine stătătoare în funcţionare şi ca înţeles, care puse cap la cap, să dea programul iniţial. În acest scop s-a creat **`funcţia`**. Ea poate fi privită ca **`o cutie neagră cu un nume şi argumente care realizează o sarcină specifică`**. Cea mai dificilă sarcină a programatorului este aducerea programului de la forma fără funcţii, la o formă mai compactă, prin rescrierea lui sub forma unor funcţii cu număr optim de parametri, în scopul reutilizării unor funcţii şi în alte programe, al ascunderii informaţiei, al clarităţii şi întreţinerii optime a programului.

`Funcţia` trebuie să ducă la compartimentarea programului în porţiuni de program prin care să se:

- **`definească o sarcină clară`**, care se reia de mai multe ori în acelaşi program, eventual se poate refolosi şi în alte programe;
- obţină **`o cutie neagră`**, adică detaliile de implementare să nu trebuiască să fie cunoscute în restul programului pentru a o utiliza;
- **`realizeze implementarea cât mai bună a sarcinii`**. În acest scop, este probabil ca funcţia să-şi depăşească atribuţiile suficiente pentru rezolvarea sarcinii particulare din programul pentru care s-a scris, dacă se anticipează posibilitatea extinderii funcţionalităţii ei şi pentru alte programe;
- ajungă - utilizând apelurile funcţiilor- la un program în care sarcinile de rezolvat se observă clar, adică să obţinem un **`program lizibil`**, în loc să avem un program lung, complicat, dificil de înţeles, cu porţiuni de cod care se repetă;
- poată **`rescrie oricând`** în vederea obţinerii unei robusteţi şi performanţe mai bune fără alte modificări în restul programului. Această rescriere trebuie să fie posibilă din moment ce restul programului nu ştie modul de implementare al funcţiilor.

Programul C este, în general, format din mai multe funcţii care pot fi stocate în unul sau mai multe fişiere sursă C. Limbajul C a fost proiectat pentru utilizarea simplă şi eficientă a funcţiilor. El este orientat pentru proiectarea aplicaţiilor prin funcţii, limbajul C numindu-se din acest motiv, funcţional. Am văzut deja că în C variabilele trebuie declarate înainte de a fi folosite în program. Procedura este identică şi la funcţii. Ele se declară şi se definesc pentru a putea fi folosite în program. Asemănarea şi diferenţa între o declaraţie şi definiţie de funcţie se va lămuri în paragrafele următoare.

6.1 Apelul funcţiilor

După cum am mai spus, funcţia este o porţiune de program, scrisă de noi sau de altcineva, care rezolvă o sarcină distinctă din program. În liniile care urmează doresc să clarific modul de apelare a funcţiilor prin prisma observării exemplelor prezentate până acum. Următoarele concluzii se pot trage:

- o funcţie **se apelează prin numele ei** urmat de o pereche de paranteze rotunde;
- dacă funcţia are **argumente**, acestea se vor plasa între paranteze rotunde şi vor fi separate prin virgule.

Iată câteva exemple de apeluri de funcţii:


```
printf("Un apel de functie"); /* functia cu numele printf() are 1 argument, sirul "Un apel de functie" */
printf("%d\n",i); /* aceeasi functie printf() are aici 2 argumente */

cos(x+y+1.618034); /* functia matematica cos() are ca argument o expresie */
getch(); /* functia getch() nu are argumente */
```


Argumentele unei funcţii pot fi constante, nume de variabile, funcţii sau expresii complexe. Din acest motiv în loc de :

```
z=x+y+1.618034;
printf("%f\n", cos(z));
```

se poate scrie:

```
printf("%f",cos(x+y+1.618034));
```

Observaţi mai sus că funcţia cu numele **printf()** are ca argument o altă funcţie, **cos()**. Multe funcţii întorc o valoare care poate fi folosită în expresii complexe:

```
x=r*cos(fi);
a=atan(x)*cos(z)*(x+y)/(z+3.2);
```


În exemplul anterior **atan()** şi **cos()** sunt funcţii din biblioteca matematică standard care se folosesc în expresiile: $x = r \cdot \cos(\varphi)$ şi $a = \text{atan}(x) \cdot \cos(z) \cdot \frac{x+y}{z+3.2}$.

6.2 Definirea unei funcţii

Funcţiile sunt echivalentul subprogramelor şi procedurilor din alte limbaje de programare de nivel înalt. Acestea **primesc ca date de intrare** un număr de **parametri**, execută apoi o porţiune de cod numită **corpul funcţiei** şi în final întorc o valoare de un anumit tip calculată în corp. Funcţiile care nu întorc valori se declară de tipul **void**. O funcţie **trebuie întodeauna declarată într-un prototip de funcţie** înainte de a fi apelată - noţiunea de prototip va fi lămurită în paragrafele care urmează. Funcţia **grupează instrucţiuni sub un singur nume. Numele** funcţiei este folosit **pentru apelarea funcţiei** şi produce execuţia grupului de instrucţini care fac parte din corpul funcţiei. Funcţia din care se apelează o altă funcţie se numeşte **funcţie apelantă**, iar funcţia care se apelează se numeşte **funcţie apelată. Funcţia** apelată, de obicei, **întoarce o valoare** în funcţia apelantă calculată pe baza instrucţiunilor din corpul ei. Valoarea întoarsă în funcţia apelantă se poate ignora dacă aşa doreşte programatorul. Modalitatea de întoarcere a valorii calculate de funcţia apelată în funcţia apelantă este instrucţiunea **return**. Forma ei de scriere este: **return expresie;. expresie** se va converti la tipul de valoare pe care îl întoarce funcţia, dacă este cazul. Funcţia ne permite să executăm acelaşi grup de instrucţiuni, cuprinse în corpul ei, din diferite porţiuni ale unui program şi pentru diferite valori ale variabilelor grupate în corpul funcţiei, folosind parametri. Din cele spuse rezultă

că o funcţie se defineşte prin:
- **numele** folosit pentru apelul funcţiei;
- o lista de zero sau mai mulţi **parametri** care se transferă funcţiei pentru prelucrări directe sau în vederea dirijării unor prelucrări;
- un **corp** care conţine instrucţiunile ce îndeplinesc sarcina pentru care funcţia a fost creată;
- o **valoare întoarsă.**

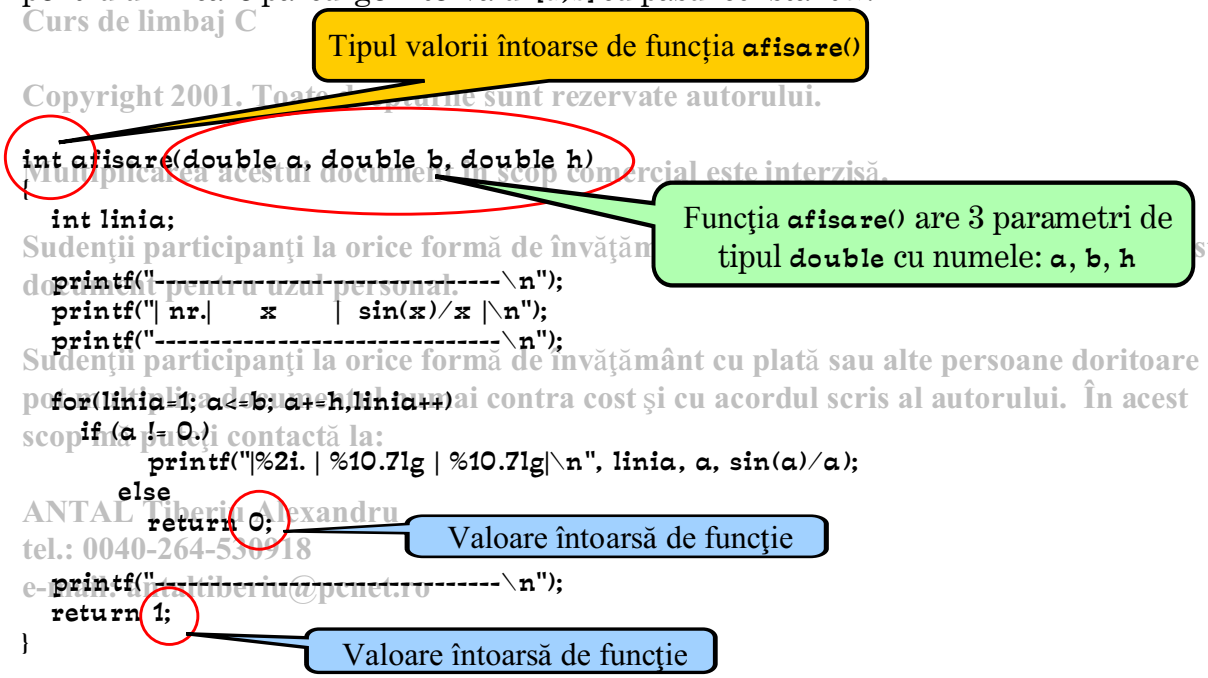

```
Formă:      tip_întors nume_functie(lista declaraţiilor de parametri)
            {
                declaraţii locale
                instrucţiuni
            }
```


Un caz special este când funcţia revine în programul apelant pentru că aceasta s-a terminat - s-a ajuns la acolada } care specifică terminarea corpului funcţiei - fără a întâlni o instrucţiune **return**. Ce fel de valoare se întoarce pentru acest caz? Este ciudat ca o funcţie, uneori să întoarcă o valoare, aleori nu. Conform definiţiei, funcţia trebuie să întoarcă valoare în ambele cazuri - cu sau fără **return** . Dacă **return** lipseşte sau nu este întâlnit, asta nu înseamnă că nu se întorce o valoare, ci că valoarea întoarsă nu este previzibilă, fiind "gunoi".

Conform **standardului**, tipul valorilor întoarse de o funcţie *poate fi*: aritmetic, structură (**struct**), reuniune (**union**), vid (**void**), poantor (**pointer**); dar *nu poate fi* o funcţie (**function**) sau un tablou (**array**).

6.2.1 Definirea unei funcţii - exemplu

Să presupunem că avem de scris o funcţie care trebuie să afişeze valoarea lui $\frac{\sin(x)}{x}$ pentru un **x** care parcurge intervalul [**a**,**b**] cu pasul constant **h**.




```
1  int afisare(double a, double b, double h)
2  {
3     int linia;
4
5     printf("-----------------------------\n");
6     printf("| nr.|     x      | sin(x)/x |\n");
7     printf("-----------------------------\n");
8
9     for(linia=1; a<=b; a+=h,linia++)
10       if (a != 0.)
11          printf("|%2i. | %10.7lg | %10.7lg|\n", linia, a, sin(a)/a);
12       else
13          return 0;
14
15    printf("-----------------------------\n");
16    return 1;
17 }
```

Pentru funcţia cu numele **afisare()** sunt specificate următoarele elemente:
Tipul întors: **int**, specificat în **linia 1**;

Numele funcţiei: `afisare`, specificat în `linia 1`. El trebuie să fie distinct de numele altor funcţii, variabile, etc. pentru ca să asigure identificarea unică a funcţiei în program;
Parametri: `double a, double b, double h`, specificaţi în `linia 1`. Fiecare parametru trebuie să aibă un `tip` şi un `nume`;
Valoarea întoarsă: `return 0`, în `linia 13` şi `return 1`, în `linia 16`. Dacă funcţia are specificat un tip al valorii întoarse aceasta va fi întoarsă prin folosirea lui `return`.
Corpul funcţiei: format din instrucţiunile cuprinse între acoladele `{, }` din `liniile 2` şi `17`.

Valorile de afişat sunt calculate în ciclul `for` din `linia 9`. Datele sunt prezentate într-o formă tabelară, capul de tabel fiind afişat în `liniile 5-7`. Dacă în urma avansului cu pasul `h` parametrul `a` ia valoarea numerică `0.`, atunci împărţirea este compromisă şi avem un caz de eroare. După terminarea afişării se revine din funcţie întorcând valoarea numerică de `0` - terminare cu eroare. Altfel, dacă terminarea este normală, revenirea din funcţia `afisare()` se face cu valoarea numerică de `1` - ea indicând terminarea afişării cu succes.

6.3 Apelul de funcţie

`Apelul` de funcţie `este în realitate un operator`. El `are ca operanzi două expresii şi are ca efect transferul controlului programului şi al argumentelor`, dacă acestea există, la secvenţa de cod cuprinsă în corpul funcţiei apelate.

Formă:	`expresie_specificator_de_funcţie (expresie_listă_argumente)`

`expresie_specificator_de_funcţie` identifică funcţia care este apelată, fiind un nume de funcţie sau o expresie care în urma evaluării specifică adresa unei funcţii. `expresie_listă_argumente` este o listă de expresii, separate prin virgulă, care constituie argumentele funcţiei apelate. Termenul de `argument` se foloseşte pentru o expresie a cărei valoare este transferată printr-un apel de funcţie. Termenul de `parametru` este folosit pentru un obiect de intrare - variabilă - într-o funcţie sau într-o declaraţie de funcţie. Termenii de `parametru actual` şi `parametru formal` sunt folosiţi uneori pentru a face aceeaşi distincţie.

6.3.1 Conversii aritmetice standard la apelul de funcţii

Pentru că avem de a face cu un operator, `tipul conversiilor realizate pentru` operanzii care alcătuiesc `argumentele funcţiei` în apel, `depinde de existenţa prototipului` funcţiei unde sunt declarate tipurile corespunzătoare argumentelor funcţiei apelate. Dacă `funcţia are prototip`, pe baza declaraţiilor de tipuri de argumente, compilatorul face verificări de tip, conversii de tip şi de număr, între argumente şi parametri. În `lipsa prototipului`, se fac numai conversii aritmetice standard ale valorilor expresiilor din lista de argumente în apelul de funcţie. Aceste conversii se realizează individual pentru fiecare argument în funcţie de tipul argumentelor. O valoare de tip `float` se va converti automat în `double`; una `char` sau `short` în `int`; una `unsigned char` sau `unsigned short` în `unsigned int`.

6.3.2 Apelul unei funcţii - exemplu

Pentru facilitarea înţelegerii, dezvolt în continuare problema propusă în paragraful 6.2.1. De această dată însă funcţia `afisare()` are o nouă definiţie. Versiunea anterioară era incorect implementată în C, fiind specifică celor care vin să lucreze în C, din alte limbaje

de programare. Fără a intra în prea multe detalii, comparaţia cu valoarea numerică `0.` era incorectă. Noua abordare foloseşte constanta **`DBL_EPSILON`** definită în fişierul antet `float.h`. **`DBL_EPSILON`** este cea mai mică valoare de tip **`double`** pentru care avem `1.0+DBL_EPSILON != 1.0`.

Această linie îi spune compilatorului cum lucrează funcţia **`afisare()`**

```
1  /* FUNCTIE1.C */
2  #include<stdio.h>
3  #include<math.h>
4  #include<float.h>
5
6  int afisare(double a, double b, double h); /* Prototip afisare() */
7  int main(void)
8  {
9      double a,b,pas;
10     printf("a = ");
11     scanf("%lf", &a);
12     printf("b = ");
13     scanf("%lf", &b);
14     printf("pas = ");
15     scanf("%lf", &pas);
16     if (!afisare(a,b,pas)) /* Apelul functiei afisare() */
17         printf("Eroare in calcule!\n\tTerminare fortata.\n");
18     else
19         printf("Terminare cu succes.\n");
20     return 0;
21 }
22
23 int afisare(double a, double b, double h)
24 {
25     int linia;
26     printf("-------------------------------\n");
27     printf("| nr.|     x     | sin(x)/x |\n");
28     printf("-------------------------------\n");
29
30     for(linia=1; a<=b; a+=h,linia++)
31         if ((a >= DBL_EPSILON) || (a <= -DBL_EPSILON))
32             printf("|%2i. | %10.7lg | %10.7lg|\n", linia, a, sin(a)/a);
33         else
34             return 0;
35     printf("-------------------------------\n");
36     return 1;
37 }
```

Compilatorul ştie că aceste expresii trebuie să fie de tipul **`double`** din prototip şi le converteşte automat dacă cumva nu sunt de tipul corespunzător.

Rezultate:

```
a = 1
b = 2
pas = .1
-------------------------------
| nr.|     x     | sin(x)/x |
-------------------------------
| 1. |         1 |  0.841471|
| 2. |       1.1 |  0.8101885|
| 3. |       1.2 |  0.7766992|
| 4. |       1.3 |  0.7411986|
```

```
a = -1
b = 1
pas = .2
-------------------------------
| nr.|     x     | sin(x)/x |
-------------------------------
| 1. |        -1 |  0.841471|
| 2. |      -0.8 |  0.8966951|
| 3. |      -0.6 |  0.9410708|
| 4. |      -0.4 |  0.9735459|
```

```
| 5. |        1.4 |  0.7038927|
| 6. |        1.5 |  0.6649967|
| 7. |        1.6 |  0.6247335|
| 8. |        1.7 |  0.5833322|
| 9. |        1.8 |  0.5410265|
|10. |        1.9 |  0.4980527|
-------------------------------

Terminare cu succes.
```

```
| 5. |       -0.2 |  0.9933467|
Eroare in calcule!
        Terminare fortata.
```

Următoarele elemente sunt esenţiale pentru apelul unei funcţii:

Prototipul: este definit în `linia 6` şi informează compilatorul despre modul în care lucrează funcţia. Astfel, funcţia cu numele `afisare`, are trei argumente de tipul `double` şi întoarce o valoare de tipul `int`.

Apelul: este porţiunea `afisare(a,b,pas)` din `linia 17`. El are ca efect transferul argumentelor şi al controlului execuţiei programului la funcţia cu numele `afisare()`. Dacă funcţia ar fi apelată prin linia `afisare(1,a,pas)`, primul argument ar fi de tipul `int`. Conform celor spuse la 6.3.1, în acest caz se face conversia automată a lui `1` la tipul `double`, adică la `1.0` pentru că acest parametru nu ar fi de tipul corect.

Valoarea întoarsă: valoarea întoarsă de funcţie este folosită pentru a decide care dintre mesajele de terminare se afişează pe ecran. Dacă terminarea se face cu eroare se revine cu valoarea `0`, `afisare()` este `falsă` pentru că nu şi-a dus treaba până la capăt şi se execută `linia 18`. Altfel, valoarea întoarsă este `1`, iar `afisare()` este `adevărată` şi se execută `linia 20` corespunzătoare lui "`afisare` nu este `falsă`". Este posibil ca valoarea întoarsă de funcţie să fie atribuită unei variabile. Este corect deci scriem `c=afisare(a,b,pas);`. Nu este obligatoriu ca valoarea întoarsă să fie folosită în funcţia apelantă. Este corect deci şi un apel de forma `afisare(a,b,pas);`. În acest caz valoarea întoarsă se va pierde.

Rolul prototipului

Folosirea prototipului duce la evitarea unor dezastre pentru că pe baza lui compilatorul poate face verificările de tip necesare la apelul funcţiei. Dacă `linia 6` nu se scrie în program, adică prototipul lipseşte, la un apel de forma `afisare(1,a,pas)` vor fi probleme serioase întrucât compilatorul presupune că toţi parametri sunt de tipul corect. Primul argument nu se converteşte automat de la tipul `int` la `double`, iar pentru că reprezentarea în RAM şi mărimea pentru stocare a celor două tipuri diferă rezultatele obţinute vor fi dubioase.

6.4 Prototipuri

Linia `int afisare(double a, double b, double h);` se numeşte prototipul funcţiei `afisare()`. Compilatorul are nevoie de câte un prototip pentru fiecare funcţie apelată în program. `Prototipul` funcţiei poartă denumirea tehnică de `declaraţia de prototip a funcţiei`. Acesta conţine trei tipuri de informaţii care trebuie cunoscute în funcţia apelantă: `numele funcţiei apelate`, `tipul întors`, `numarul`, `ordinea şi tipul tuturor argumentelor` funcţie apelate. Prezenţa prototipului anunţă compilatorul de intenţia noastră de a apela funcţia `afisare()`. Informaţiile din prototip ajută compilatorul să genereze codul corect pentru apelul de funcţie şi permit efectuarea de verificări pe marginea apelului de funcţie (de exemplu, se poate verifica dacă a fost transferat numărul corect de argumente). Dacă prototipul este omis, apelul funcţiei se va realiza folosind convenţii standard. Lista parametrilor se foloseşte pentru verificarea corespondenţei dintre argumentele funcţiei din apel şi lista parameterilor din definiţia funcţiei.

În lipsa prototipului de funcţie ...

Valoarea întoarsă: conform acestor convenţii, o funcţie fără prototip se presupune că întoarce o valoare de tipul `int`. Probleme speciale apar de exemplu la funcţiile matematice `sin()`, `cos()`, `tan()`, `exp()` etc. care întorc o valoare de tipul `double`. Fără prototipuri, valorile întoarse sunt incorecte datorită trunchierii la 2, la 4, eventual la 8 octeţi a valorii întoarse.

Argumentele funcţiei: compilatorul va presupune că numărul corect de argumente a fost transferat funcţiei în apel şi că tipurile lor sunt corecte. Când am vorbit de `rolul prototipului` am dat un exemplu în care numărul de argumente era corect, dar tipul primului parametru nu. Majoritatea compilatoarele sunt echipate cu o opţiune de avertizare pe nivele. Dacă nivelul de avertizare este scăzut (`low level warning`) compilatorul nu va afişa avertismente pentru apelul funcţiilor fără prototip. Creşterea acestui nivel va cauza apariţia unor mesaje de avertisment pentru funcţiile fără prototip.

Prototipurile funcţiilor din biblioteca standard

Prototipurile funcţiilor standard sunt scrise de cei care au scris bibiloteca şi sunt stocate în `fişiere antet` (`header files`). Programatorul trebuie doar să folosească directiva `#include` pentru fişierul antet relevant. Numele acestuia este specificat întotdeauna în documentaţia compilatorului.

Cum scriem prototipuri?

Sper că aţi observat că `prototipul unei funcţii este prima linie din definiţia ei urmată de caracterul ;`. Astfel, dacă funcţia s-a definit deja este suficient să copiem această primă linie din definiţie. Numele argumentelor din prototip este ignorat complet de compilator, astfel în loc de `int afisare(double a, double b, double h);` putem scrie `int afisare(double, double, double);`. Dacă aceste argumente au totuşi nume, acestea nu trebuie să fie identice cu cele din definiţia funcţiei. Apare întrebarea de ce să mai includem aceste nume dacă tot sunt ignorate? Pentru că orice funcţie are un rol, numele argumentelor pot furniza informaţii semnificative despre funcţionarea ei. De exemplu, în loc de `int afisare(double a, double b, double h);` aş fi putut scrie un prototip de forma `int afisare(double inceput, double sfirsit, double pas);` care ar fi lămurit pe deplin semnificaţia celor trei argumente.

Rolul lui ; în prototip

Pentru că prototipul şi prima linie din definiţia funcţiei sunt aproape identice, de unde va şti compilatorul care este prototipul şi care este definiţia? Hotărârea se ia pe baza caracterului care urmează după paranteza rotundă închisă, ")", în felul următor: caracterul "{"arată că este vorba de o definiţie, iar caracterul ";" de un prototip. Din acest motiv adăugarea unui caracter ; după prima linie a unei funcţii conduce la o eroare fatală. Dacă mai întâi se întâlneşte ; compilatorul crede că a fost un prototip. După prototip vine o acoladă "{", adică se începe un bloc, de cine anume va ţine acest bloc?

Exemple de prototipuri

Exemplele care urmează prezintă câteva prototipuri. În limbajul C, spre deosebire de limbajul Pascal, nu există noţiunea de procedură. Cuvântul cheie `void` trebuie folosit explicit, ca tip întors de o funcţie, pentru a defini un fel de "funcţie echivalentă" cu o procedură, adică una care nu întoarce o valoare. `void` folosit în lista argumentelor specifică o funcţie fără argumente. Este bine ca la specificarea unui prototip să se scrie un scurt comentariu care descrie rolul funcţiei. Iată câteva exemple:

```
/* Citeste un intreg de la tastatura,
   nu are argumente,
   intoarce o valoare de tipul int */
```

```
int citeste_int(void);

/* Emite un semnal sonor - beep,
   nu are argumente,
   nu intoarce o valoare */
void un_beep(void);

/* Calculeaza media aritmetica a trei numere intregi,
   trei argumente de tip int,
   intoarce un tip double */
double medie_ar(int, int, int);

/* Citeste un numar real de a tastatura,
   verificarea argumentelor este dezactivata,
   intoarce un tip int */
citeste_real();
```


ANTAL Tiberiu Alexandru
tel.: 0040-264-530918
e-mail: antaltiberiu@pcnet.ro

Ultimul prototip merită să fie discutat. Forma de scriere folosită produce dezactivarea verificărilor la nivelul argumentelor. Ceea ce nu înseamnă că nu se transferă argumente. În orice apel funcţiei `citeste_real()` i s-ar putea transfera unul sau mai multe argumente fără ca măcar un mesaj de avertisment să fie afişat.

6.5 Parametrii funcţiilor

`Argumentele sunt nume ale valorilor care se transferă funcţiei printr-un apel.` Parametrii sunt obiectele de intrare pe care funcţia le primeşte efectiv şi le prelucrează în corpul ei. În declaraţia de prototip între parantezele rotunde este specificată lista parametrilor funcţiei. Numele parametrilor care se folosesc în corpul funcţiei se mai numesc şi valorile transferate funcţiei. Numele acestora poate fi acelaşi sau diferit cu cele din declaraţia de prototip. Dacă în lista parametrilor există cel puţin un parametru, lista se poate termina cu o virgulă urmată de trei puncte, adică în "...". Această notaţie specifică o funcţie cu număr variabil de argumente. În apel o astfel de funcţie trebuie să aibă cel puţin atâtea argumente câţi parametri sunt înaintea virgulei. Ordinea şi tipul parametrilor trebuie să fie la fel cu cele din declaraţia funcţiei dacă aceasta există. Tipurile argumentelor, după conversii aritmetice standard, trebuie să devină compatibile cu tipul corespunzător al parametrilor.

6.6 Ordinea evaluării argumentelor

Să presupunem că avem următoarea linie de cod: `printf(" . . . ", sqr(5.0), cos(7.9));`. Poate vă pare surprinzător, dar ordinea în care cele două funcţii sunt apelate este nedefinită din punctul de vedere al standardului ANSI C. Unele compilatoare vor apela `sqr()` mai întâi, iar altele `cos()` şi ambele variante sunt corecte. Pentru cazul de mai sus nu este importantă ordinea evaluării. În cazul în care s-ar afişa ceva din interiorul fiecărei funcţii, rezultatul întors la fiecare apel ar fi corect, ordinea evaluării argumentelor nefiind însă definită. Totuşi, înainte de intrarea în funcţia apelată, toate argumentele vor fi evaluate incluzând aici şi toate efectele secundare.

6.7 Transferul prin valoare

C este un limbaj unde în locul fiecărui argument de transferat unei funcţii se transferă o copie a valorii argumentului. Acest tip de transfer de argumente în care funcţia primeşte

o copie a valorilor originale, protejate astfel contra modificărilor, poartă denumirea de **transfer prin valoare**. Funcţia poate modifica valorile parametrilor ei, care fiind copii ale argumentelor, nu duc la modificarea valorilor argumentelor. Uneori este un avantaj, alteori un dezavantaj. Pentru că o funcţie poate întoarce o singură valoare, de exemplu în cazul unei funcţii **citeste_doi_intregi()** nu se pot întoarce valorile celor doi întregi folosind o singură funcţie. Va fi necesar să scriem o funcţie **citeste_intreg()** care va fi apelată de două ori pentru a citi cei doi întregi. O metodă de eliminare a acestui inconvenient este variabila externă. Înainte de a trece la acestea doresc să menţionez că C suportă şi o metodă de **transfer prin referinţă**. Aici parametrul nu este o copie a valorii variabilei transferate, ci a adresei la care este stocată variabila în RAM. Aţi văzut deja acest mecanism în acţiune la funcţia **scanf()** care este capabilă să altereze valorile parametrilor. Totul este însă legat de operatorul **&** care urmează să fie discutat la tipul de dată structurat **poantor**.

6.8 Variabile externe sau variabile globale

Programele C sunt formate din **obiecte externe**. Acestea sunt variabile sau funcţii. Denumirea de "externe" se foloseşte pentru a le diferenţia de **obiectele** denumite **interne**. Acestea sunt argumentele funcţiilor şi variabilele declarate în interiorul funcţiilor - dacă vă amintiţi de paragraful despre definiţia funcţiilor, în corpul acestora apărea o linie de forma: **declaraţii locale**. Variabilele externe sunt definite în afara funcţiilor. Din acest motiv ele nu ţin de nici o funcţie, fiind la dispoziţia tuturor funcţiilor programului. În **standard** aceste variabile sunt descrise ca având **legare externă** pentru că toate referinţele la numele acestora, chiar şi din funcţii care au fost compilate separat, identifică obiectele cu numele respective (este posibil ca un acelaşi nume să identifice obiecte distincte). Se pot defini şi variabile externe care să fie vizibile numai în fişierul sursă în care au fost definite. Felul în care se realizează definiţia lor va fi discutat într-un paragraf următor, numit durate de existenţă. În limbajul C funcţiile sunt întotdeauna externe. Spre deosebire de limbajul Pascal, nu este posibil să se definească în interiorul unei funcţii o altă funcţie. Pentru că variabilele externe sunt global accesibile, adică pot fi utilizate din întregul program, ele permit implementarea unei strategii de comunicaţie între funcţii care ar trebui să întoarcă mai mult de o valoare. În exemplul care urmează o astfel de funcţie este **citeste_2_reali()**. Aceasta stochează datele citite în două variabile globale, **deimpartit** şi **impartitor**, de tipul **float**.

```
1  /* EXTERN.C */
2  #include<conio.h>
3  #include<stdio.h>
4  #include<float.h>
5
6  /* Prototipurile functiilor din program */
7  void citeste_2_reali(void);
8  float calculeaza_citul(float deimpartit, float impartitor);
9  void afiseaza_citul(float cit);
10
11 const float INFINIT=FLT_MAX;
12
13 float deimpartit, impartitor; /* doua variabile externe - globale */
14
15 /* functia main() - punctul de intrare in program */
16 int main(void)
```

```
{
        float cit; /* o variabila locala */
        do {
                        citeste_2_reali();
                        cit=calculeaza_citul(deimpartit,impartitor);
                        afiseaza_citul(cit);
        } while (getch() != 'i');

        return 0;
}
/* sfarsitul functiei main() */

/* definitiile functiilor */
void citeste_2_reali()
{
        printf("Baga 2 numere, deimpartit si impartitor: ");
        scanf("%f %f",&deimpartit,&impartitor);
}

float calculeaza_citul(float deimpartit, float impartitor)
{
        if (impartitor == 0.0F)
                return(INFINIT);
        else
                return(deimpartit/impartitor);
}

void afiseaza_citul(float cit)
{
        if (cit == INFINIT)
                printf("Citul este nedefinit\n");
        else
                printf("Citul este %f\n",cit);
}
```

Rezultate:

```
Baga 2 numere, deimpartit si impartitor: 1 2
Citul este 0.500000
Baga 2 numere, deimpartit si impartitor: -1 34.57
Citul este -0.028927
Baga 2 numere, deimpartit si impartitor: 1 0
Citul este nedefinit
```

Aici s-a introdus i pentru terminare

Este bine ca prototipurile funcţiilor să se scrie la începutul programului. Începând cu acea linie de program şi până la terminarea lui, funcţiile pot fi apelate prin numele lor oriunde în program şi pot fi făcute verificări de tip pe marginea fiecărui apel.

Limbajul C permite ca variabilele externe şi funcţiile să fie organizate în mai multe fişiere sursă C distincte care se pot compila împreună sau separat pentru a crea un program executabil. Dacă doriţi, unele funcţii pot fi stocate în biblioteci, caz în care pot fi refolosite în mai multe programe. Problema care se pune este cum se va reorganiza un program existent în unul cu mai multe fişiere? O modalitate de reorganizare a programului anterior în 5 fişiere sursă C este:

```
/* CIT.H */
```

```
#include<float.h>

void citeste_2_reali(void);
float calculeaza_citul(float deimpartit, float impartitor);
void afiseaza_citul(float cit);

const float INFINIT = FLT_MAX; /* definitia constantei INFINIT */
float deimpartit, impartitor; /* definitiile variabilelor */
```

```
/* MAIN.C */
#include<conio.h>
#include "cit.h"

int main(void)
{
        float cit; /* o variabila locala */
        do {
                citeste_2_reali();
                cit=calculeaza_citul(deimpartit,impartitor);
                afiseaza_citul(cit);
        } while (getch() != 'i');

        return 0;
}
```

```
/* CIT2REAL.C */
#include<stdio.h>
extern float deimpartit, impartitor; /*declaratiile variabilelor externe */

void citeste_2_reali()
{
        printf("Baga 2 numere, deimpartit si impartitor: ");
        scanf("%f %f",&deimpartit,&impartitor);
}
```

```
/* CALCIT.C */
extern const float INFINIT; /* declaratia constantei INFINIT */

float calculeaza_citul(float deimpartit, float impartitor)
{
        if (impartitor == 0.0F)
                return(INFINIT);
        else
                return(deimpartit/impartitor);
}
```

```
/* AFICIT.C */
#include<stdio.h>
extern const float INFINIT; /* declaratia constantei INFINIT */

void afiseaza_citul(float cit)
{
        if (cit == INFINIT)
```

```
8         printf("Citul este nedefinit\n");
9     else
10        printf("Citul este %f\n",cit);
11 }
```

Cele 5 fişiere sursă sunt: **CIT.H**, **MAIN.C**, **CIT2REAL.C**, **CALCIT.C** şi **AFICIT.C**. Cele care au extensia C pot fi compilate independent unul de celălat. Fişierul **CIT.H** este antetul în care am stocat definiţiile constantelor şi variabilelor externe, împreună cu prototipurile funcţiilor din program. Pentru înţelegerea criteriilor folosite la reorganizarea programului trebuie cunoscută noţiunea de vizibilitate a variabilelor C.

La începutul cursului am prezentat succint preprocesorul şi directiva **#include**. Forma folosită în **linia 3** din **MAIN.C** este o variantă specială care permite încărcarea conţinutului fişierului **CIT.H**, din directorul curent în programul sursă C unde acesta apare. Lămuriri suplimentare vor fi date în capitolul directivelor de procesare.

6.8.1 Vizibilitatea (scope)

La declararea unei variabile se specifică automat şi **vizibilitatea** ei. Aceasta depinde de locul în care s-a făcut declaraţia variabilei în program şi **determină porţiunile de program din care acea variabilă poate fi folosită**, adică este vizibilă. Un nume este vizibil, deci poate fi utilizat numai în porţiunile de program care fac parte din zona de vizibilitate. Vizibilitatea unui nume poate fi limitată pentru a creşte restricţiile la nivelul unui fişier, al unei funcţii, al unui bloc, sau al unui prototip de funcţie în care acesta figurează. Următoarele categorii de vizibilităţi sunt posibile:

- **bloc**: un bloc este o porţiune de program cuprinsă între acolade { ... }. În C se mai numeşte şi **vizibilitate locală** întrucât ea corespunde corpului funcţiei în care s-a făcut declaraţia. O astfel de variabilă este vizibilă din locul în care s-a făcut declaraţia ei şi până la acolada de închidere a blocului "}" în care s-a realizat declaraţia;
- **parametri de funcţie**: au aceeaşi vizibilitate cu variabilele declarate în interiorul funcţiilor şi astfel corespund vizibilităţii de tipul bloc;
- **prototip de funcţie**: variabilele declarate în interiorul unui prototip de funcţie au o vizibilitate specială. Aceasta începe din punctul corespunzător declaraţiei parametrului şi ţine până la paranteza rotundă ")" care închide lista parametrilor funcţiei;
- **fişier**: variabilele cu această vizibilitate sunt declarate în afara tuturor blocurilor. Aceste nume sunt vizibile, începând cu punctul în care s-au declarat, până la terminarea fişierului sursă al declaraţiei.

6.8.2 Declaraţia şi definiţia

Vizibilitatea unei variabile externe sau a unei funcţii este cea de tip fişier. Dacă o variabilă externă este referită în program înainte de a fi definită într-un program sursă distinct sau în cel curent este obligatorie folosirea cuvântului cheie **extern**. Legat de variabilele externe, se uzitează doi termeni: declaraţia şi definiţia. O **declaraţie** este un anunţ legat de proprietăţile variabilei - în principal ale tipului acesteia. O **definiţie** adaugă anunţului anterior şi alocarea spaţiului în RAM pentru stocarea acelei variabile. Pentru programul anterior, **liniile 8** şi **9** ale fişierului sursă **CIT.H** sunt definiţii pentru constanta **INFINIT** şi pentru variabilele **deimpartit** şi **impartitor**. Tot aici se face şi alocarea spaţiului pentru stocarea acestor obiecte în RAM. Liniile care conţin **extern const float INFINIT;** şi **extern float deimpartit, impartitor;** sunt numai declaraţii ale constantei şi variabilelor respective

- stocarea lor în RAM s-a făcut deja ca urmare a definiţiilor lor. Este obligatoriu să existe o definiţie a unei variabile externe într-unul din fişierele sursă ale programului. Toate celelalte fişiere pot conţine numai declaraţii externe pentru accesul la aceasta.

În cazul funcţiilor, problema este diferită. Distincţia între o declaraţie de variabilă externă şi o definiţie de variabilă externă se face prin prezenţa sau absenţa cuvântului cheie **`extern`**. La funcţii, definiţia este formată din antetul funcţiei - prima linie a acesteia - împreună cu corpul ei. Declaraţia funcţiei se face prin prototipul ei. După cum am mai spus - vezi 6.6, o declaraţie externă este singura variantă posibilă în C pentru funcţii. Din acest motiv, cuvântul cheie **`extern`** este opţional în declaraţia de prototip. Astfel **`void citeste_2_reali(void);`** şi **`extern void citeste_2_reali(void);`** sunt acelaşi lucru.

6.9 Stiva

Stiva este o regiune din RAM gestionată după politica **`Ultimul Introdus este Primul Extras`** (**`L`**ast **`I`**n **`F`**irst **`O`**ut sau prescurtat, **`LIFO`**). Imaginaţi-vă o stivă de cărţi sub forma: Conceptual, stiva limbajului C este asemenea acestor cărţi. **`Introducerea unei cărţi noi`**

Figura 4 - O stivă (de cărţi)

în stivă se face numai la nivelul cărţii celei mai de sus - cea pusă ultima în stivă, aceasta fiind chiar vârful stivei. Altfel apare pericolul ca înteaga stivă să se prăbuşească. **`Extragerea unei cărţi`** respectă aceeaşi procedură: numai cartea cea mai de sus va putea fi luată fără să apară pericolul prăbuşirii stivei. Conform celor spuse, unei stive i se poate adăuga un articol nou numai la vârful acesteia şi i se poate extrage un articol numai de la vârful ei. Dacă aşezarea cărţilor permite să se vadă autorul şi titlul accesul la ele este simplu, atâta timp cât introducerea şi extragerea cărţilor se face sub controlul riguros al procedurilor prezentate. De exemplu, dacă dorim să extragem cea de a treia carte de la vârful stivei, se extrag pe rând primele trei cărţi, apoi se pun la loc ultimele două.

6.9.1 Stiva şi limbajul C

`Limbajul C utilizează o zonă de RAM` cu politica menţionată **`denumită`** chiar **`stivă`** pentru:

- **`stocarea valorilor variabilelor locale`** (care sunt declarate în interiorul funcţiilor);
- **`stocarea valorilor transferate`** prin mecanismul parametrilor funcţiei apelate;
- **`stocarea adresei de revenire`** în funcţia apelantă.

Operaţia de introducere pe stivă se numeşte în engleză **`push`**, iar cea de extragere **`pop`**. Când o valoare de variabilă se transferă ca parametru al unei funcţii, o copie a valorii respective se introduce pe stivă. Funcţia extrage de pe stivă această valoare copie pe post de parametru. Pentru că funcţia poate accesa numai valoarea parametrului, spaţiul folosit pentru stocarea originalului fiind alocat într-un alt bloc de funcţie, valoarea originală nu se poate modifica în corpul funcţiei. Operaţia de rezervare a unei porţiuni din RAM pentru

stocarea unui obiect se numeşte **alocare**. Dacă funcţia are declarate variabile locale, alocarea spaţiului pentru acestea se face tot pe stivă. Funcţia are responsabilitatea ca la terminarea ei să elibereze automat spaţiul alocat acestor variabile. La revenire în funcţia apelantă, funcţia apelată are sarcina de a distruge şi spaţiul alocat pe stivă pentru parametrii ei. Cum ştie însă C unde anume să se revină în funcţia apelantă? La apelul funcţiei, C trebuie să reţină unde anume era înainte de intrarea în funcţia apelată. În acest scop, el va stoca tot pe stivă adresa locaţiei curente care va fi adresa de revenire.

Mecanismul transferului de parametri prin stivă se poate descrie astfel:

① funcţia apelantă introduce pe stivă parametrii în ordinea lor inversă de apariţie în lista de parametri [**push(parametri)**];
② se stochează pe stivă adresa de revenire în funcţia apelantă, apoi se apelează funcţia;
③ funcţia apelată îşi citeşte parametrii de pe stivă;
④ funcţia apelată introduce pe stivă variabilele locale [**push(variabile locale)**];
⑤ la revenire, funcţia apelată extrage variabilele locale [**pop(variabile locale)**] şi sare înapoi la funcţia apelantă (pe baza adresei care a fost stocată pe stivă la ②);
⑥ funcţia apelantă extrage parametrii de pe stivă [**pop(parametri)**];
⑦ valoarea întoarsă de funcţia apelată este tratată - dacă este cazul.

6.9.2 Exemplu de lucru cu stiva

```
1  /* ARIA.C */
2  #include<stdio.h>
3
4  double aria(double, double);
5
6  int main(void)
7  {
8        double h,r,a;
9
10       printf("inaltimea cilindrului = ");
11       scanf("%lf", &h);
12       printf("raza cilindrului = ");
13       scanf("%lf", &r);
14
15       a=aria(r,h);
16       printf("aria cilindrului este: %lg\n",a);
17
18       return 0;
19 }
20
21 double aria(double raza, double inaltimea)
22 {
23       double pi=3.1415926535;
24
25       return pi*raza*raza*inaltimea;
26 }
```

Rezultate:

```
inaltimea cilindrului = 2
raza cilindrului = 3
aria cilindului este: 56.5487
```

Programul îşi începe execuţia cu funcţia **main()**. Pentru că **r**, **h** şi **a** sunt variabile locale,

se alocă spaţiul pentru acestea pe stivă. Când se apelează **aria()**, cu argumentele **r** şi **h**, valorile acestor variabile se copiază pe stivă. Se apelează funcţia. Argumentul **r** se află pe poziţia parametrului **raza**, iar argumentul **h** se află pe poziţia parametrului **inaltimea**. Din acest motiv, când îşi citeşte valorile parametrilor de pe stivă, **aria()** va stoca valoarea lui **r** în **raza** şi cea a lui **h** în **inaltimea**. Pentru că funcţia are o variabilă locală cu numele **pi** se alocă spaţiul pe stivă pentru ea şi se iniţializează cu valoarea "**3.1415926535**". Se calculează valoarea ariei cu expresia din **linia 25** şi se revine în funcţia apelantă (**main()**), datorită lui **return** care produce terminarea funcţiei apelate şi revenirea din ea. Compilatoarele au strategii diferite de întoarcere a valorilor la revenirea din funcţii. Unele întorc valoarea pe stivă, altele în regiştri. Înainte de terminarea funcţiei apelate, ea este responsabilă pentru "distrugerea" lui **pi** prin extragerea lui de pe stivă. La revenirea în **main()**, cei doi parametri **raza** şi **inaltimea** sunt distruşi şi ei. Valoarea întoarsă este salvată într-un registru, de aici copiindu-se în variabila **a**. După afişarea rezultatului pe ecran urmează **return 0**. Înainte de terminarea funcţiei **main()** se distruge spaţiul alocat pe stivă pentru **r**, **h** şi **a**. Apoi valoarea "**0**" se încarcă într-un registru pentru a fi transferată sistemului de operare.

6.10 Organizarea memoriei pentru programele în execuţie

Când un program este în curs de execuţie memoria alocată lui este divizată în mai multe zone de RAM numite şi segmente:

Segmentul de cod (CODE SEGMENT): aceasta este porţiunea de RAM în care sunt stocate instrucţiunile maşină corespunzătoare funcţiilor din programele C - **main()** de exemplu - sau cele extrase din biblioteci - de exemplu **printf()**. Prin definiţie, zona segmentului de cod este protejată contra modificărilor pe timpul execuţiei programului, altfel ar fi posibil să scriem un program care se modifică pe sine.

Stiva(STACK): o porţiune de RAM în care sunt stocate toate variabilele locale. Conţinutul acestei zone şi dimensiunea ei se modifică pe timpul execuţiei programului, depinzând de apelurile şi de revenirile din funcţiile care alcătuiesc programul. În unele limbaje, stiva are o dimensiune maximă fixă pe timpul execuţiei unui program. Vor apărea erori ca urmare a încercării de a extrage articole de pe stiva vidă sau ca urmare a introducerii în stiva plină (datorită modului de implementare) a unui articol pentru care nu mai este loc. Ultimul caz este mai des întâlnit şi se numeşte **depăşire de stivă** (**stack overflow**) având ca efect suprascrierea unor locaţii de memorie adiacente zonei de stivă, rezultând erori a căror depanare este dificilă.

Segmentul de date (DATA SEGMENT): este o zonă de RAM în care se stochează variabilele globale. Din moment ce variabilele globale sunt întotdeauna stocate acolo şi au o dimensiune fixă, spre deosebire de cele locale care se creează şi se distrug automat, dimensiunea totală a segmentului de date nu se schimbă pe timpul execuţiei programului.

HEAP-ul: este o zonă de RAM a cărui dimensiune poate varia în timpul execuţie programului. În limbajul C, dimensiunea lui este controlată prin patru funcţii de alocare dinamică a memoriei **malloc()**, **calloc()**, **realloc()** şi **free()**. Pe heap, porţiuni de memorie denumite blocuri sunt alocate şi eliberate într-o ordine arbitrară, iar şablonul de alocare împreună cu dimensiunea lui vor fi cunoscute numai în timpul execuţiei programului. Funcţiile pentru manipularea heap-ului vor fi discutate într-un capitol următor.

6.11 Durata de existenţă sau persistenţa (storage class) variabilelor

În funcţie de segmentul în care sunt stocate, avem **`variabile globale`**, **`locale`** şi **`registru`**. Despre cele **`registru`** nu am discutat până acum explicit, însă dacă vă amitiţi, în capitolul operatorilor am folosit uneori exprimarea "valoarea expresiei se stochează într-un registru". În limbajul C există cuvinte cheie prin care se poate controla segmentul în care se alocă spaţiul pentru stocarea variabilelor. În majoritatea cărţilor de C acest subiect este tratat sub denumirea de **`durate de existenţă`**.

6.11.1 Durata de existenţă `auto`

Am spus că implicit, spaţiul pentru stocarea variabilelor locale este alocat pe stivă. Când funcţia este apelată, variabilele locale sunt stocate pe stivă. La terminarea funcţiei, spaţiul folosit pentru aceste variabile este eliberat - se mai zice că stiva este descărcată. Deoarece procedurile de alocare şi eliberare a spaţiului din RAM pentru aceste variabile decurg automat, durata lor de existenţa (modul în care compilatorul alocă spaţiul de stocare în RAM pentru variabile) se întreţine automat, de unde şi numele de **`auto`**. Folosirea cuvântului cheie **`auto`** este deci inutilă dacă, de exemplu, în **ARIA.C** modificăm **`linia 8`** de la **`double h,r,a;`** la **`auto double h,r,a;`** efectul va fi că aceste variabile vor deveni automate - cum de altfel erau deja.

Iniţializarea variabilelor **`auto`**

Deşi compilatorul asigură gestionarea pe stivă a spaţiului alocat acestor variabile, acestea nu se iniţializează implicit. Este obligatoriu ca utilizatorul să facă explicit iniţializarea şi să ţină minte că valorile stocate în aceste variabile se pierd - datorită descărcării automate a stivei - la revenirea din funcţie. Din acest motiv se mai spune că între două apeluri ale funcţiei valorile variabilelor locale se pierd. În realitate nu se poate ca o variabilă să nu conţină o valoare. Acesta este şi cazul valabil pentru variabilele **`auto`**, dar valoarea pe care o conţin nu este previzibilă, adică nu ne putem baza pe valoarea lor iniţială.

Performanţe

Dacă o funcţie care conţine foarte multe variabile locale este apelată de multe ori din diferite porţiuni ale unui program, pentru fiecare apel spaţiul variabilelor locale va fi alocat respectiv descărcat de pe stivă la terminarea funcţiei. Această procedură este consumatoare de timp, iar programul va rula încet datorită strategiei de programare adoptate. Probabil soluţia într-un astfel de caz este ca unele variabile să fie globale.

6.11.2 Durata de existenţă `static`

Din punctul de vedere al segmentului în care se stochează, spaţiul alocat variabilelor **`static`** este segmentul de date. Acesta este segmentul în care sunt stocate şi variabilele globale. Dacă o variabilă locală este declarată **`static`** (statică), spaţiul pentru aceasta nu se mai alocă pe stivă, ci pe segmentul de date.

Iniţializarea variabilelor **`static`**

Variabilele **`static`** sunt **`iniţializate implicit cu valoarea 0`** la începutul execuţiei programului. Acestea vor "reţine" valorile stocate în ele şi între

apelurile funcţiilor în care s-a făcut declararea lor întrucât mărimea segmentului de date este fixă pe tot parcursul execuţiei programului (spaţiul alocat lor nu se distruge). Un exemplu cu declararea unei variabile statice este prezentat în continuare:

```
int test(void)
{
    int i; /* variabila locala auto */
    static int Contor; /* variabila statica, Contor este initializata cu 0 */
    ...
}
```


Performanţe

Pentru aceste variabile nu există faze de creare şi distrugere care se reiau în funcţie de apelurile din program, motiv pentru care programele cu astfel de variabile rulează mai rapid decât cele cu variabile **auto**.

Vizibilitatea variabilelor static

Variabilele cu durata de existenţă **static** au o vizibilitate limitată la blocul în care au fost declarate. Dacă este vorba de variabile statice externe, acestea nu pot fi accesate din alte fişiere sursă, asigurând astfel o modalitate de ascundere a datelor. Din acest motiv putem spune că variabilele statice au o vizibilitate locală. Iată un exemplu în acest sens:

```
/* STATIC.C */
#include<stdio.h>

void f1(void);
void f2(void);
void f3(void);

int i; /* un i global */

int main(void)
{
        int i=3; /* un i local auto */

        printf("in main() i=%i\n",i);
        f1();
        f2();
        f3();

        return 0;
}

void f1(void)
{
        static int i=5; /* i local static */
        printf("in f1() i=%i\n",i);
}

void f2(void)
{
        static int i=7; /* alt i local static */
        printf("in f2() i=%i\n",i);
}

void f3(void)
{
        /* aici se afiseaza valoare i-ului global */
```

```
37          printf("in f3() i=%i \n",i);
38  }
```

Rezultate:

```
in main() i=3
in f1() i=5
in f2() i=7
in f3() i=0
```

Dintre variabilele "i" cele din liniile 8, 24, 30 sunt permanente, iar cea din linia 12 este temporară. Precum vedeţi, toate valorile stocate sunt distincte. Deşi numele acestor variabile este acelaşi, ele sunt permanent separate. De asemenea, valorile celor trei "i"-uri sunt şi inaccesibile între ele.

Funcţiile `static`

În condiţii obişnuite, numele de funcţii sunt globale şi vizibile în toate modulele unui program. Dacă o funcţie este declarată `static`, numele ei va deveni invizibil în afara modulului în care s-a declarat.

6.11.3 Durata de existenţă register

O variabilă declarată ca `register` anunţă compilatorul ca va fi des utilizată în program. Din acest motiv ar fi bine să fie stocată într-un registru al UCP pentru ca programul să fie cât mai mic şi cât mai rapid în rulare. Cuvântul cheie este moştenit din C K&R când nu existau optimizatoare de cod. Acesta este motivul pentru care numeroase posibilităţi de optimizare au fost incluse ca parte a limbajului. O declaraţie `register` este de forma:

```
void f(register unsigned j, register short s)
{
    register int i;
    register char ch;
    ...
}
```

şi poate fi aplicată numai variabilelor `auto` sau parametrilor de funcţii.

Iniţializarea variabilelor `register`

Variabilele `register` nu sunt iniţializate implicit, din care cauză programatorul este nevoit să le iniţializeze. Numai în acest fel valoarea lor de plecare va fi previzibilă.

Restricţii de utilizare

Declaraţia `register` este luată în seamă de compilator numai dacă are un registru liber în UCP. Tipurile de date care pot fi stocate într-un registru al UPC sunt dependente şi de tipul acestuia. Numărul de regiştri ai unui UCP este şi el dependent de tipul lui. Trebuie să menţionez tot aici că variabilele stocate în regiştrii UCP nu au adresă, motiv pentru care nu pot participa în expresii care fac calcule cu această adresă - vezi poantorii.

6.12 Tipuri de legare

Noţiunea de legare determină modalitaţile de comunicare între funcţiile unor module distincte ale unui program prin apel şi prin intermediul variabilelor externe.

Vizibilitatea poate fi privită în două contexte distincte. Unul `lexical`, uneori numit şi

vizibilitate statică pentru că este fixată şi la momentul compilării corespunde porţiunilor de program din care caracteristicile obiectului sunt cunoscute. După cum aţi văzut, în programul **STATIC.C**, acelaşi nume de identificator este declarat de mai multe ori având vizibilităţi distincte sau aceleaşi vizibilităţi, reuşind însă să refere corect identificatorul în cauză. Aceasta pentru că vizibilitatea lexicală permite ca acelaşi nume să fie folosit în vizibilităţi diferite fără nici o interferenţă, chiar dacă vizibilitatea lui este aceeaşi, cu condiţia ca acesta să fie unic în spaţiul numelor din care face parte.

Celălalt context al vizibilităţii este asociat funcţiilor şi variabilelor externe, prin care se derulează **comunicaţia între modulele separat compilate** ale programului. Există trei tipuri de legare: **internă**, **externă** şi **nespecificată**. Dacă declaraţia unui obiect cu vizibilitate de tip **fişier** conţine cuvântul cheie **static**, atunci identificatorul are **legare internă**. În interiorul unui modul, fiecare utilizare a identificatorului cu legare internă reprezintă un unic identificator de obiect sau funcţie. Dacă prima declaraţie cu vizibilitate de tip **fişier** a identificatorului nu foloseşte durata de existenţă **static** obiectul are **legare externă** şi este echivalent cu specificarea duratei de existenţă **extern**. Un nume de identificator cu legare externă specifică aceeaşi funcţie sau obiect de tipul dată, asemenea oricărei alte declaraţii cu acelaşi nume şi cu legare externă. Cele două declaraţii pot fi în acelaşi modul sau în module distincte. Dacă obiectul sau funcţia are durată de existenţă fişier, atunci devine vizibil în toate modulele programului. Fiecare obiect, indiferent de legare, trebuie să aibă o singură definiţie. Obiectele cu legare externă vor avea o singură definiţie la nivelul tuturor modulelor programului. Obiectele cu legare internă, fiind interne fiecărui modul, trebuie să aibă câte o definiţie separată în fiecare modul.

Dacă la declararea unui identificator vizibilitatea lui este de tip **bloc** şi nu include durata de existenţă **extern**, identificatorul nu are legare şi este unic în funcţie.

Următorii identificatori nu au legare:

① un identificator declarat diferit de orice altceva decât un obiect sau funcţie;
② un identificator declarat parametru de funcţie;
③ un identificator cu vizibilitate bloc pentru un obiect declarat fără cuvântul **extern**.

Dacă un identificator nu are legare, redeclararea lui la acelaşi nivel de vizibilitate generează o eroare de redefinire.

6.12.1 Legarea şi modulele de program - un exemplu comentat

Cele prezentate până acum vor fi reluate sistematizat pentru a înţelege exemplul care va fi prezentat în continuare, deci am spus că variabilele **extern** sunt vizibile la nivel de fişier. Sunt definite în afara oricărei funcţii şi pot fi accesate de majoritatea funcţiilor. Funcţiile pot fi definite numai la nivel **extern**, deci nu pot fi imbricate. Implicit, toate referinţele la variabilele sau funcţiile **extern** sunt referinţe la acelaşi obiect, deci au legare externă. Utilizarea lui **static** duce la modificarea acestei convenţii implicite.

Declaraţiile de variabile la nivelul **extern** sunt fie definiţii de variabile (**declaraţii de definiţii**), fie referinţe la variabile definite altundeva (**declaraţii de referenţiere**). O declaraţie de variabilă externă (**extern**) care face şi iniţializarea variabilei se numeşte declaraţie de definiţie a variabilei.

Următoarele reguli sunt valabile pentru **static**:

- variabilele declarate în afara tuturor blocurilor fără cuvântul rezervat static îşi păstrează valorile în tot programul. Pentru a restrânge accesul acestora la unele module de programe particulare se foloseşte static, ceea ce le va conferi "legare internă". Pentru a le face globale, se omite specificarea tipului de durată de existenţă sau se specifică cel extern, caz în care legarea va fi externă;
- o variabilă externă se poate defini o singură dată într-un program. Se poate defini o altă variabilă cu acelaşi nume şi durata de existenţă static într-un alt modul de program. Din moment ce definiţiile static sunt vizibile numai în modulul în care s-au făcut, nu vor exista conflicte. Aceasta este o metodă utilă de ascundere a numelor de identificatori care se partajează între funcţii ale unui singur modul de program şi nu sunt vizibile pentru alte module;
- static poate fi utilizat şi în cazul numelor de funcţii, caz în care numele funcţiei este făcut invizibil în afara modulului de program în care a fost declarat.

Următoarele reguli sunt valabile pentru **extern**:

- specificatorul de durată de existenţă extern declară o referinţă la o variabilă definită într-un alt loc. Se poate folosi o declaraţie extern pentru a face o definiţie dintr-un alt modul vizibil sau pentru a face vizibilă o variabilă înainte de definirea ei în cadrul aceluiaşi modul;
- pentru ca o referinţă externă să fie validă, variabila la care se face referinţă trebuie să fie definită numai o singură dată, la nivel extern. Această definiţie poate fi în oricare din modulele programului.

```
1  /*************************************************
2  *                                                *
3  *          Fisierul sursa C (modulul) 1.c        *
4  *                                                *
5  *************************************************/
6
7  #include<stdio.h>
8  extern int i; /* declaratie de referentiere a lui i, definit mai jos */
9
10 void urmator(void); /* prototipul functiei urmator() */
11 void alta(void);    /* prototipul functiei alta() din fisierul sursa 2.c */
12
13 main()
14 {
15         i++;
16         printf("%d\n",i);  /* i este egal cu 4*/
17         urmator();
18 }
19
20 int i =3; /*definitia lui i sau declaratia de definitie a lui i */
21
22 void urmator(void)
23 {
24         i++;
25         printf("%d\n",i); /* i este 5 */
26         alta();
27 }
```

```
/*************************************************
*                                                *
*        Fisierul sursa C (modulul) 2.c          *
*                                                *
*************************************************/

#include <stdio.h>
extern int i; /* declaratie de referentiere la i din fisierul 1.c */
void alta(void)
{
    i++;
    printf("%d\n",i); /* i este 6 */
}
```

În exemplul de mai sus, sunt prezentate două fişiere sursă (sau module de program) C, `1.c` şi `2.c`. Pentru crearea unui program executabil se va folosi un proiect, atât în Borland C cât şi în Microsoft C, care va conţine cele două nume de fişiere. Cele două fişiere sursă conţin în total 3 declaraţii externe ale lui `i`. Numai una dintre ele este o declaraţie de definiţie şi anume `int i=3;` (`linia 20` din `1.c`) care defineşte variabila globală `i` şi o iniţializează cu valoarea `3`. Declaraţia de referenţiere la începutul lui `1.c` (`linia 8`) face variabila globală vizibilă înainte de apariţia definiţiei ei în fişier. Declaraţia de referenţiere din `linia 8` a modulului `2.c` face variabila vizibilă în modulul `2.c`. Dacă nu se specifică o definiţie pentru variabila `i` într-unul dintre modulele de program, compilatorul presupune că este o declaraţie de referenţiere `extern int i` şi că o declaraţie de definiţie de forma `int i = 0` apare undeva într-un alt modul al programului.

Toate cele 3 funcţii, `main()`, `urmator()` şi `alta()`, execută aceeaşi operaţie: incrementează valoarea lui `i`, apoi o afişează pe ecran. Valorile afişate sunt `4`, `5`, şi `6`.

6.12.2 Legarea și duratele de existență ale variabilelor

Fie programul:

```
/* LEGDUREX.C */
#include<stdio.h>
int i=1;
void alta(void);

int main()
{
   extern int i;/* Referinta la i-ul din linia 3*/
   static int a; /*Valoarea initiala e 0 si vizibila numai in main() */
   register int b = 0; /* b va fi stocta intr-un registru */
   int c = 0;/* c are implicit durata de existenta auto */
   printf("%d\n%d\n%d\n%d\n",i,a,b,c);
/* Valorile afisate sunt 1, 0, 0, 0 */
   alta();
   return 0;
```

```
16 }
17
18 void alta(void)
19 {
20    static int *ext_i=&i; /* ext_i e globala si atribuita unui poantor */
21    int i=16; /* i este redefinit si i-ul global nu mai este vizibil */
22    static int a=2; /* acest a este vizibil numai în functia alta() */
23    a+=2;
24    printf("%d\n%d\n%d\n", i,a,*ext_i);
25 /* Valorile afisate sunt 16, 4 si 1*/
26 }
```


În exemplul de mai sus, variabila `i` este definită la nivel extern cu valoarea iniţială `1` (`linia 3`). O declaraţie `extern` este folosită în funcţia `main()`, pentru a declara o referinţă la variabila externă `i`. Variabila `a` statică este iniţializată automat cu valoarea 0 pentru că valoarea de iniţializare este omisă. Funcţia `printf()` va afişa valorile `1`, `0`, `0` şi `0` pe coloană.

Noţiunea de poantor (pointer în engleză) nu a fost încă discutată. Pentru moment este suficient să ştiţi că un poantor stochează adresa unei variabile. În funcţia `alta()`, adresa variabilei globale `i` se foloseşte pentru a iniţializa poantorul `static extern_i`. Metoda de iniţializare funcţionează pentru că variabila globală are o durată de existenţă de tipul `static`, în sensul că adresa din RAM la care este stocată nu se schimbă pe timpul execuţiei programului pentru că variabila se alocă în segmentul de date . În continuare, variabila `i` se redefineşte ca variabilă locală cu valoarea iniţială de `16` (`linia 20`). Această redefinire nu afectează valoarea stocată în `i` extern, aceasta fiind ascunsă, prin folosirea aceluiaşi nume pentru o variabilă locală. Valoarea globală este acum accesibilă indirect prin poantorul `extern_i`. Încercarea de a da adresa variabilei `auto i` unui poantor nu merge pentru că adresa va fi diferită la fiecare intrare în bloc. Variabila `a` este declarată `static` şi iniţializată cu valoarea `2` (`linia 22`). Acest `a` nu intră în conflict cu `a`-ul din `main()` din moment ce variabilele `static` sunt vizibile la nivel intern, adică numai în interiorul blocului în care au fost declarate.

Variabila `a` este mărită cu `2`, rezultând valoarea `4`. Dacă funcţia `alta()`, ar fi apelată din nou în acelaşi program, valoarea iniţială pentru `a` ar fi 4 din moment ce variabilele `static` îşi păstrează valorile pe timpul în care programul iese şi reintră în blocul în care au fost declarate.

6.13 Recursivitatea

Limbajul C permite utilizarea recursivităţii. **`O funcţie care se apelează pe ea însăşi se numeşte recursivă`**. Dacă **`apelul`** nu se face **`direct`** din corpul funcţiei, ci **`indirect`**, prin una sau mai multe funcţii, atunci grupul acestor funcţii se numeşte "mutual recursiv". Dacă o astfel de funcţie se va apela pe sine în mod neîntrerupt, programul nu se va termina niciodată. Pentru evitarea acestei situaţii, funcţia realizează un test bazat pe parametrii ei în care verifică un caz iniţial - o condiţie din care poate reveni cu o valoare fără a se apela pe sine.

Exemplul clasic de funcţie recursivă este factorialul:

```
0!=1
n!=n*(n-1)!
```

sau, ca să ne apropiem puţin de scrierea în limbajul C:

factorial(0)=1
factorial(n)=n*factorial(n-1) sau $$factorial(n)=\begin{cases}1, & n=0\\ n*factorial(n-1), & n\geq 1\end{cases}$$

Programul C corespunzător este:


```
1  /* FACTORIA.C */
2  #include<stdio.h>
3
4  unsigned long factorial(unsigned int);
5
6  int main(void)
7  {
8          unsigned int n;
9
10         printf("n = ");
11         scanf("%u", &n);
12         printf("factorial(%u) = %lu\n", n ,factorial(n));
13
14         return 0;
15 }
16
17 unsigned long factorial(unsigned int n)
18 {
19         if (n == 0) return 1L; /* aici e mai sanatos sa scriem n <=0 */
20         else return n*factorial(n-1);
21 }
```


Rezultat:

```
n = 7
factorial(7) = 5040
```

Funcţia **factorial()** are un singur parametru, pe **n**, care se va stoca pe stivă. La fiecare apel recursiv pe stivă se va stoca noua valoare a lui **n** independent de cea anterioară. Iniţial **n** ia valoarea **7**, o copie a lui **n** se stochează pe stivă, apoi la primul apel recursiv al lui **factorial()** argumentul va lua valoarea **6**, o copie a valorii noului argument - valoarea **6** - se va stoca din nou pe stivă şi procedura de apel recursiv va continua până ce argumentul ia valoarea **0** - **linia 19**. Pentru acest caz apelul recursiv nu se reia, ci se revine din funcţie cu valoarea numerică **1**. Terminarea apelului recursiv determină revenirea în **linia 20** unde se poate calcula valoarea lui **factorial(1) = 1*factorial(0)=1*1=1.** Apoi se revine - din cauza lui **return** - cu o nouă valoare, cea a lui **factorial(1)** în aceeaşi linie şi se calculează **factorial(2) = 2*factorial(1)=2*1=2.** În final, după ce se execută toate revenirile, se obţine valoarea **lui factorial(7) = 7*factorial(6)=7*6*5*4*3*2*1=5040.**

Avantaje şi dezavantaje ale recursivităţii

Recursivitatea este o metodă de a cuprinde infinitul în finit. Programele recursive vor fi din acest motiv compacte. Din cauza că utilizează masiv lucrul cu stiva acestea vor rula încet şi pot duce la apariţia erorii numite "depăşire de stivă". Recursivitatea este convenabilă pentru prelucrarea unor structuri de date recursive cum sunt de exemplu listele sau arborii.

Evitarea recursivităţii

În limbajele imperative (C, Pascal, BASIC, FORTRAN) - unde programatorul descrie explicit secvenţele de paşi ce trebuie urmaţi pentru a obţine rezultatul - iteraţia este de preferat în locul

recursivităţii, cel puţin din motive de viteză. Se numeşte iteraţie repetarea unei secvenţe de instrucţiuni. Iteraţiile sunt caracterizate printr-o mulţime de condiţii iniţiale, un pas de iteraţie şi o condiţie de terminare. Implementată iterativ, funcţia `factorial()` este:

```
1 unsigned long factorial(unsigned int n)
2 {
3         unsigned long p,i;
4
5         for (p=1L,i=2L;i<=n;i++)
6                 p*=i;
7
8         return p;
9 }
```


Ciclul `for` grupează condiţiile iniţiale în `p=1L,i=2L`, pasul de iteraţie în `i++` şi condiţia de terminare în `i<=n`.

Conf. dr. ing. ANTAL Tiberiu Alexandru
Universitatea Tenhică din Cluj-Napoca
Facultatea Construcţii de Maşini
Catedra de Mecanică şi Programare
Curs de limbaj C

Copyright 2001. Toate drepturile sunt rezervate autorului.

Multiplicarea acestui document în scop comercial este interzisă.

Sudenţii participanţi la orice formă de învăţământ superior bugetar pot multiplica acest document pentru uzul personal.

Sudenţii participanţi la orice formă de învăţământ cu plată sau alte persoane doritoare pot multiplica documentul numai contra cost şi cu acordul scris al autorului. În acest scop mă puteţi contactă la:

ANTAL Tiberiu Alexandru
tel.: 0040-264-530918
e-mail: antaltiberiu@pcnet.ro

Universitatea Tehnică din Cluj-Napoca
Facultatea Construcţii de Maşini
Catedra de Mecanică şi Programare
Curs de limbaj C

7 Preprocesarea

Programul preprocesor este apelat de compilator pentru realizarea prelucrărilor de macrosubstituţie, compilare condiţionată şi includerea de fişiere, înainte de compilarea programului sursă C.

Conform celor spuse, faza de compilare este precedată de cea numită **preprocesare**. Comunicaţia cu programul preprocesor se derulează prin **instrucţiuni** sau **directive de preprocesare**. Acestea încep cu semnul diez, "**#**". Sintaxa acestor linii este independentă de restul limbajului de programare. Liniile pot să apară oriunde în textul sursă, efectul lor poate fi local sau global.

NOTA: Nu se pune punct și virgulă (;), în mod "mecanic", după o directivă pentru că aceasta nu este o instrucțiune a limbajului C.

Programul preprocesor are rolul de a elimina comentariile din programul sursă şi de a efectua numeroase substituţii de texte în acesta pe baza directivelor de preprocesare. Rezultatul acţiunilor de mai sus este un nou program sursă, invizibil de obicei pentru utilizator şi care va fi transferat programului compilator. De regulă, directivele sunt plasate în primele linii ale programului sursă. Ele pot fi scrise începând cu oricare linie, dar din motive de consistenţă, se preferă scrierea lor la începutul programului din prima coloană a textului sursă. Directivele de preprocesare cauzează modificarea programului sursă, însă aceste modificări durează numai atâta timp cât ţine faza de compilare a programului sursă. Preprocesarea se derulează în mai multe faze succesive, dar distincte logic, care depind de implementarea programului preprocesor. În continuare prezint câteva din fazele preprocesării, împreună cu prelucările caracteristice fazelor.

Lipirea liniilor

Liniile de program sursă care se termină în caracterul backslash "\" şi sunt urmate de trecere la o linie nouă (caracterul **newline**), care se obţine prin apăsarea tastei <**Enter**> în textul sursă C, se îmbină pentru a forma o singură linie a programului sursă.

Directiva #define

Directiva **#define** permite definirea de macrocomenzi pentru înlocuirea unui şir de caractere cu un altul. Se foloseşte pentru definirea unor constante numerice în vederea creşterii lizibilităţii programului.

```
#define PI 3.151592653
#define TIP_CERC 0
```

Forma:	#define IDENTIFICATOR1 şir2

unde **IDENTIFICATOR1** este un cuvânt fără spaţii. **şir2** este separat de **IDENTIFICATOR1** prin cel puţin un spaţiu şi poate fi orice caracter, cuvânt sau propoziţie; de asemenea,

poate conţine orice număr de spaţii. Directiva **#define** are ca efect înlocuirea tuturor apariţiilor lui **IDENTIFICATOR1** din cadrul modulului curent cu **şir2** începând cu linia următoare directivei. Există şi o formă mai "savantă" a lui **#define**, însă pe aceasta am s-o prezint prin intermediul exemplelor.

```
1  /* DEFINE1.C */
2  #include <stdio.h>
3
4  #define INCEPUT  5 /* Valoarea de inceput a ciclului */
5  #define SFARSIT 10 /* Valoarea de sfarsit a ciclului */
6  #define MAX(A,B) ((A)>(B)?(A):(B)) /* macro definitia lui MAX */
7  #define MIN(A,B) ((A)>(B)?(B):(A)) /* macro definitia lui MIN */
8
9  int main(void)
10 {
11    int i, min, max;
12    int comp = 7;
13
14    for (i = INCEPUT ; i <= SFARSIT ; i++)
15    {
16       max = MAX(i, comp);
17       min = MIN(i, comp);
18       printf("i=%2i, comp=%2i, max este %2i si min este %2i\n",i,comp,max,\ min);
19    }
20
21    return 0;
22 }
```

Rezultate:

```
i= 5, comp= 7, max este  7 si min este  5
i= 6, comp= 7, max este  7 si min este  6
i= 7, comp= 7, max este  7 si min este  7
i= 8, comp= 7, max este  8 si min este  7
i= 9, comp= 7, max este  9 si min este  7
i=10, comp= 7, max este 10 si min este  7
```

În exemplul anterior am folosit noţiunea de macro. O macrodefiniţie sau, mai pe scurt, un **macro** este o formă specială de **#define** capabil cel puţin în aparenţă să realizeze o decizie logică sau să implementeze o funcţie matematică sub un nume unic. La definirea unui macro este necesar să nu existe spaţii între numele lui şi paranteza deschisă după care se specifică parametrii. Voi folosi **linia 6** din programul de mai sus pentru a explica "funcţionarea" unui macro. În orice loc din textul sursă în care preprocesorul găseşte cuvântul **MAX** urmat de un grup de paranteze, va aştepta specificarea a doi parametri care vor fi substituiţi conform specificaţiilor din a doua parte a macrodefiniţiei. Astfel, primul parametru va substitui fiecare **A** din a doua parte a macrodefiniţiei, iar al doilea parametru va substitui fiecare apariţie a lui **B** în cea de a doua parte din macrodefiniţie. Când se ajunge la **linia 16**, fiecare **A** se va substitui cu **i**, iar fiecare **B** cu **comp**. Astfel, **linia 16** care va fi transferată compilatorului este:

```
max = ((i) > (comp)?(i):(comp))
```

NOTA: Comentariile împreună cu şirurile de caractere rămân întotdeauna neafectate de substituţiile cauzate de macrodefiniţii.

Să trecem în continuare la un exemplu de macro care va da chix elegant.

```
/* DEFINE2.C */
#include <stdio.h>
#define GRESIT(A) A*A       /* Macro gresit pentru patrat  */
#define PATRAT(A) (A)*(A)    /* Macro Ok pentru patrat      */
#define ADUNA_CHIX(A) (A)+(A) /* Asa adunarea nu va merge ...*/
#define ADUNA_OK(A) ((A)+(A)) /* Asa da ...                 */
#define START 1
#define STOP  3

int main(void)
{
   int i, val_ct=3;

   for (i = START; i <= STOP; i++)
   {
        printf("Patratul lui %2i este %2i\n",i+val_ct, PATRAT(i+val_ct));
        printf("\tPatratul gresit a lui %2i este %3i\n",i+val_ct, GRESIT(i+val_ct));
   }

   printf("\nAcum testam macro-ul de adunare\n");
   for (i = START; i <= STOP; i++)
   {
        printf("Macro de adunare gresit = %2i\
, si acum cel Ok =  %2i\n",5*ADUNA_CHIX(i), 5*ADUNA_OK(i));
   }

   return 0;
}
```


Rezultate:

```
Patratul lui  4 este 16
        Patratul gresit a lui  4 este   7
Patratul lui  5 este 25
        Patratul gresit a lui  5 este  11
Patratul lui  6 este 36
        Patratul gresit a lui  6 este  15

Acum testam macro-ul de adunare
Macro de adunare gresit =  6, si acum cel Ok =  10
Macro de adunare gresit = 12, si acum cel Ok =  20
Macro de adunare gresit = 18, si acum cel Ok =  30
```

Avem două macrodefiniţii greşite în **liniile 3** şi **5**. Pentru cel din **linia 3**, în **linia 17**, când **i** ia valoarea **1**, iar **val_ct, 5,** se calculează suma lor prin **i+val_ct**, adică **1+5=6**. Pentru cazul macrodefiniţiei **PATRAT**, valorile se grupează astfel: **(1+5)*(1+5)=6*6=36**. În varianta **GRESIT** avem **1+5*1+5=11**. Deci, parantezele rotunde sunt necesare pentru gruparea corectă a parametrilor, dar contează câte sunt şi unde le punem. Pentru **linia 5**, şi ea greşită, am folosit "stilul" de paranteze care dă rezultatele corecte la produs, însă la adunare nu se poate lucra la fel. Pentru **5*ADUNA_CHIX(i)** cu **i=1**, obţinem **5*1+1=5+1=6**. Ceea ce dorem să obţinem este **5*(1+1)=5*2=10**, care este răspunsul corect obţinut cu **ADUNA_OK** ca urmare a parantezelor puse în jurul întregii expresii. Un pic de studiu individual al macrodefiniţiilor din cele două exemple prezentate sper că va clarifica modul în care se pune problema. În general, programatorii experimentaţi pun paranteze în jurul fiecărui argument de macro şi paranteze adiţionale în jurul întregii expresii.

O linie de forma **#undef IDENTIFICATOR1** face ca definiţia lui **IDENTIFICATOR1** să fie

uitată de preprocesor. Este corect să se aplice **#undef** asupra unui identificator care nu a fost definit.

Directiva **#include**

Directiva **#include** inserează conţinutul unui fişier în programul sursă. Practic, linia directivei se va înlocui prin conţinutul întregului fişier. De obicei fişierul conţine declaraţii de identificatori şi/sau definiţii de funcţii care sunt grupate într-un singur modul pentru a facilita un anumit tip de operaţii, cum ar fi cele de intrare/ieşire, prelucrare a şirurilor etc. Acelaşi fişier poate fi inclus în mai multe module ale aceluiaşi program sau ale unor programe diferite.

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

Forma: **#include<numefişier> sau #include"numefişier"**

Această directivă nu face mai mult cu programul decât ar face un simplu editor de texte; editoarele de texte sunt şi ele capabile de operaţia de includere a conţinutului unui fişier stocat pe disc în conţinutul fişierului curent. **numefişier** trebuie să fie numele unui fişier text ASCII stocat pe disc. Prima formă asigură căutarea lui **numefişier** în locuri definite dependent de implementarea particulară a preprocesorului. Cea de-a două formă va căuta mai întâi pe **numefişier** în acelaşi loc cu programul sursă, apoi va trece la căutările specifice primei forme.

Directive de compilare condiţionată

#if, **#ifdef**, **#ifndef**, **#else**, **#endif** sunt directive pentru compilare condiţionată. Acestea permit numai compilarea unor porţiuni din programul sursă. Se poate scrie astfel un program general, portabil sub mai multe sisteme de operare, de exemplu DOS şi UNIX, porţiunile specifice sistemului de operare fiind compilate, în funcţie de valori luate de anumite constante. Voi prezenta câteva dintre directivele de compilare condiţionată prin intermediul exemplelor.

În programul următor - **IFDEF.C**, **OPTIUNE_1** apare într-un **#define** fără argumente. Aceasta este o formă particulară de **#define** care duce la crearea identificatorului **OPTIUNE_1**. Dacă ulterior se testează existenţa lui se poate lua o decizie pe baza existenţei sau inexistenţei lui. Pentru că **OPTIUNE_1** este definit în **linia 4**, când preprocesorul ajunge la **linia 16**, va include în programul sursă tot ceea ce există între **liniile 16** şi **19**. **ifdef** se traduce "dacă s-a definit", iar **endif** marchează terminarea grupului de instrucţiuni care vor fi incluse în programul sursă la existenţa lui **OPTIUNE_1**. Pentru a nu avea definit identificatorul **OPTIUNE_1** este suficient să transformăm **linia 4** într-un comentariu prin modificarea ei la forma **/* #define OPTIUNE_1 */**, apoi se recompilează, linkeditează şi se relansează în execuţie programul.

```
1  /* IFDEF.C */
2  #include <stdio.h>
3
4  #define OPTIUNE_1 /* Se def. ident. pentru controlul preprocesorului */
5  #ifdef OPTIUNE_1
6     int cont = 17; /* Ac. linie exista doar daca s-a def. OPTIUNE_1 */
7  #endif
8
9  int main(void)
10 {
11    int i;
```

```
12      for (i = 0 ; i < 3 ; i++)
13      {
14          printf("In ciclu, index = %d", i);
15  Curs de limbaj C
16  #ifdef OPTIUNE_1
17          /* Aceasta linie exista sau nu, condiţionata de OPTIUNE_1 */
18      printf(" si cont = %d", cont);
19  #endif
20          printf("\n");
21      }
22
23      return 0;
24  }
25  #undef OPTIUNE_1
```

Dacă programul este rulat în forma de mai sus se obţin rezultatele:

```
In ciclu, index = 0 si cont = 17
In ciclu, index = 1 si cont = 17
In ciclu, index = 2 si cont = 17
```

Fără definiţia lui **OPTIUNE_1** din linia 4 se obţine:

```
In ciclu, index = 0
In ciclu, index = 1
In ciclu, index = 2
```

Exemplul care urmează ilustrează utilizarea directivelor de preprocesare pentru includerea unei porţiuni de cod în programul sursă dacă un identificator nu este definit. Directiva **ifndef** se citeşte "dacă nu s-a definit - if not defined". Se folosesc doi identificatori **OPTIUNE_1** şi **AFISARE_DATE**, iar în funcţie de existenţa lor se pot obţine 3 variante de răspunsuri. Dacă exemplul anterior era pur didactic şi simplu, acesta este tot didactic, dar puţin mai complex. Identificatorul **OPTIUNE_1** are rolul inversat din programul anterior, iar **AFISARE_DATE** se foloseşte pentru a valida afişarea dacă acesta este definit. Daca **AFISARE_DATE** nu este definit se va afişa un mesaj specific acestui caz.

```
1   /* IFNDEF.C */
2   #include <stdio.h>
3   #define OPTIUNE_1 /* Se defineste id. pentru controlul preprocesorului */
4   #define AFISARE_DATE  /* Daca id. este definit se va afisa */
5
6   #ifndef OPTIUNE_1
7       int cont = 17; /* Linia exista numai daca OPTIUNE_1 nu este definita */
8   #endif
9
10  int main(void)
11  {
12      int i;
13
14  #ifndef AFISARE_DATE
15      printf("Nu se vor afisa rezultate cu aceasta versiune a programului\n");
16  #endif
17      for (i = 0 ; i < 3 ; i++)
18      {
19  #ifdef AFISARE_DATE
20          printf("In ciclu, Index = %i", i);
21
22
```

```
23  #ifndef OPTIUNE_1
24        printf("cont = %i", cont); /* Linia se va afisa conditionat */
25  #endif
26
27        printf("\n");
28  #endif
29     }
30
31     return 0;
   }
```

Pentru cazul când există cei doi identificatori se afişează rezultatele:

```
In ciclu, Index = 0
In ciclu, Index = 1
In ciclu, Index = 2
```

Celelalte două variante de rezultate sper să le găsiţi voi. Sunt numai 3 tipuri de rezultate care se vor afişa, deşi există 4 stări distincte care pot fi obţinute prin gruparea stărilor de existenţă sau inexistenţă a celor doi identificatori.

Directiva `#line`

Directiva **`#line`** spune preprocesorului că pentru programul C generat, numărul intern de linii şi numele fişierului C să fie modificate. Compilatorul foloseşte aceste numere de linii şi numele de fişier pentru afişarea erorilor găsite în faza de compilare.

Formă: `#line constantă "nume de fişier"`

De exemplu, prin **`#line 123 "test1.c"`**, numărul intern de linie este modificat la **`123`** iar numele fişierului la **`test1.c`**.

Directive `#pragma`

Orice compilator are un set de opţiuni dependente de UCP şi sistemul de operare sub care se rulează. De exemplu, pentru unele programe datele trebuie să fie stocate la anumite adrese în RAM. Alteori dorim să controlăm modul în care funcţiile primesc parametri. Directiva **`#pragma`** asigură aceste facilităţi, dar argumentul ei este de obicei diferit pentru fiecare combinaţie de UCP şi sistem de operare. Pentru cunoaşterea acestora va trebui să consultaţi documentaţia compilatorului pe care îl folosiţi.

Nume predefinite

Există un grup de identificatori care în urma substituţiei produc informaţii speciale. Aceştia nu se pot redefini sau "uita" cu **`#undefine`**. Câţiva dintre aceştia se află în tablelul care urmează:

Nume identificator	Semnificaţie
`__LINE__`	O constantă în baza zecimală ce conţine numărul de ordine a liniei curente din codul sursă.
`__FILE__`	Un şir ce conţine numele fişierului în curs de compilare.
`__DATE__`	Un şir care conţine data compilării sub forma `Mmm dd yyyy`.
`__TIME__`	Un şir care conţine ora compilării sub forma `hh:mm:ss`.

Prezint în continuare un exemplu de folosire a acestor nume:

```
1  /* PREDEF.C */
2
3  #include<stdio.h>
4
5  int main(void)
6  {
7
8          printf("Linia de program %d,\ndin fisierul sursa %s,\ncompilat\
9  la data %s, ora %s\n", __LINE__, __FILE__, __DATE__, __TIME__);
10
11         return 0;
12 }
```

Rezultate:

```
Linia de program 9,
din fisierul sursa C:\Visual C++\prefef.c,
compilat la data Feb  3 2000, ora 16:22:29
```

Conf. dr. ing. ANTAL Tiberiu Alexandru
Universitatea Tenhică din Cluj-Napoca
Facultatea Construcţii de Maşini
Catedra de Mecanică şi Programare
Curs de limbaj C

Copyright 2001. Toate drepturile sunt rezervate autorului.

Multiplicarea acestui document în scop comercial este interzisă.

Sudenţii participanţi la orice formă de învăţământ superior bugetar pot multiplica acest document pentru uzul personal.

Sudenţii participanţi la orice formă de învăţământ cu plată sau alte persoane doritoare pot multiplica documentul numai contra cost şi cu acordul scris al autorului. În acest scop mă puteţi contacta la:

ANTAL Tiberiu Alexandru
tel.: 0040-264-530918
e-mail: antaltiberiu@pcnet.ro

Universitatea Tenhică din Cluj-Napoca
Facultatea Construcţii de Maşini
Catedra de Mecanică şi Programare
Curs de limbaj C

8 Poantori

Principial, variabila de tipul poantor sau, mai pe scurt, **`poantorul`** (pointer-ul în engleză) **`stochează adresa unei locaţii din RAM`**. Poantorul poate fi tipizat, adică prin tipul lui va indica tipul obiectului la care poantează. Poantorii pot fi priviţi asemenea unor salturi care asigură legături dintr-o parte a programului în alta. Tony Hoare, în colecţia lui de eseuri "Hints on Programming Language Design", 1973, Prentice-Hall spunea: **"`Introducerea poantorilor în limbajele de nivel înalt a fost un pas înapoi după care este posibil să nu ne mai revenim niciodată`"**. În C, nu trebuie să ne facem griji datorită afirmaţiei de mai sus, cel puţin pentru că K&R nu îl numesc limbaj de nivel înalt, ci mai degrabă un limbaj de asamblare de nivel înalt. Pentru că C este un limbaj destul de jos este imposibil să se scrie în el ceva serios fără poantori. De exemplu, este imposibil să alterăm valorile argumentelor unei funcţii aducând modificări asupra parametrilor în corpul funcţiei, cu cele discutate până acum. Numai folosind poantori se pot crea variabile noi în timpul execuţiei programului. C permite programului să aloce o cantitate de RAM pe heap, întorcând o adresă care poate fi stocată într-un poantor. Această tehnică poartă denumirea de **`alocare dinamică a memoriei`**.

RAM-ul şi poantorii

În primul capitol, am spus că RAM-ul este o secvenţă de locaţii, fiecare locaţie având o adresă unică şi un conţinut unic, care se poate modifica. Cazul tipic de organizare a locaţiei este grupul de 8 biţi, care se mai numeşte şi octet. Putem spune că RAM-ul este un "vector" $M = (L_1, L_2, ..., L_k, ..., L_n)$, unde L_k este o locaţie formată dintr-un grup de 8 biţi, iar $_k$ este adresa locaţiei. Ştiţi că tipul **`char`** se stochează pe un octet, iar tipul **`long`** pe 4 octeţi consecutivi. La nivelul RAM-ului, în cazul comun, poantorul se stochează într-un grup de 4 (eventual 2) locaţii succesive (vezi în *Figura 5* pe L_1, L_2, L_3, L_4). În *Figura 5* se reprezintă grafic legătura creată prin poantarea la o variabilă **`a`** de tipul **`char`** de către un poantor **`pta`**.

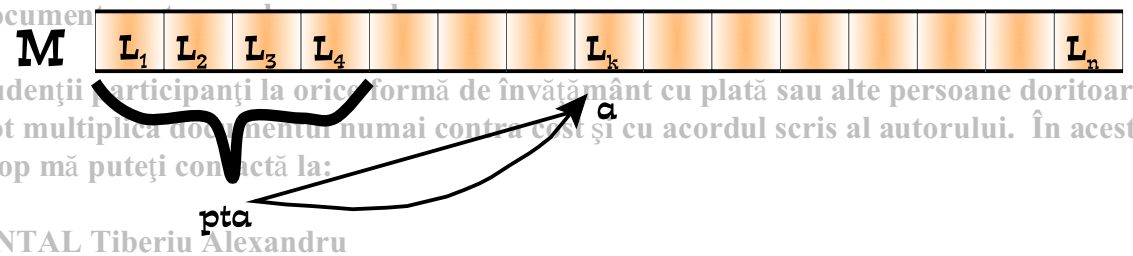


Figura 5 - Model de reprezentare a poantorului în RAM

Când un poantor stochează adresa unui obiect, prin modificarea conţinutului poantorului - a adresei stocate în el - putem poanta la diferite obiecte din RAM. Astfel, apare posibilitatea manipulării (atribuire, citire, modificare) informaţiilor stocate la adrese diferite cu ajutorul unei singure variabile, aceea de tipul poantor.

Poantorii în ANSI C

Datorită asemănării poantorilor cu instrucţiunea **goto** este posibil să se facă legături incorecte sau cu zone de RAM interzise. Deci, din neatenţie sau lipsă de experienţă, un poantor va putea poanta "incorect". Era cazul ca o oarecare disciplină să fie introdusă şi pentru variabila poantor. Aceasta este marea contribuţie a standardului ANSI C. El defineşte regulile de lucru cu variabilele poantor. Acestea au fost preluate din practică, de la programatorii profesionişti, iar compilatoarele C bune obligă la folosirea acestora.

8.1 Declararea poantorilor

Un poantor se declară în C prin precedarea numelui de variabilă cu caracterul asterisc "*". Acesta informează compilatorul că dorim să declarăm o variabilă poantor şi să aloce spaţiul corespunzător numărului de octeţi necesari pentru a stoca o adresă de memorie. Dacă prin linia:

```
char a;
```

se declară o variabilă a de tipul întreg **char**, atunci prin linia:

```
char *pta;
```

se declară o variabilă poantor la un tip **char** cu numele **pta**. Din punct de vedere al declaraţiei, poziţia caracterului * este nesemnificativă atât timp cât se află între tipul şi numele variabilei poantor. Astfel, declaraţia de mai sus se poate scrie şi aşa:

```
char* pta;
```

Se pot declara mai mulţi poantori în aceeaşi linie de program astfel:

```
1 char *pta, *ptb, pt1;
2 int* ptx1, x2, *ptx3;
```

În **linia 1**, **pta** şi **ptb** sunt poantori la tipul **char**, dar **pt1** este o variabilă de tipul **char** pentru că * lipseşte. În **linia 2**, **ptx1** şi **ptx** sunt poantori la tipul **int**, iar **x2** este o variabilă întreagă.

Tipul poantorului

La fel cum o variabilă are un tip ce specifică atributele acesteia în program, şi poantorul are un tip prin care compilatorul este anunţat asupra numărului de octeţi şi a atributelor datelor care sunt valide începând de la adresa poantată de poantor.

8.2 Operatorul adresă "&"

Operatorul adresă "&" l-am folosit în funcţia **scanf()**. El are ca rezultat adresa unei variabile stocate pe stivă sau în zona segmentului de date. Tehnic vorbind **&** întoarce adresa operandului. Operandul poate fi un nume de funcţie sau un nume de obiect, iar rezultatul este un poantor la operand.

```
1 /* PT1.C */
2 #include<stdio.h>
3
4 int a, *pta; /* a si pta sunt variabile globale */
```

```
5
6   int main(void)
7   {
8           int *pt1a; /* pt1a este variabila locala */
9           pta=&a;    /* se atribuie adresa lui a poantorului pta */
10          a=50;      /* se atribuie valoarea 50 variabilei a */
11          pt1a=pta;  /* se atribuie adresa stocata în pta lui pt1a */
12
13          printf("adresa lui a: %p\n", &a);
14          printf("continutul lui a: %i\n\n", a);
15
16          printf("continutul lui pta: %p\n", pta);
17          printf("valoarea la care poanteaza pta: %i\n\n", *pta);
18
19          printf("continutul lui pt1a: %p\n", pt1a);
20          printf("valoarea la care poanteaza pt1a: %i\n\n", *pt1a);
21
22          printf("valorile numerice sunt: %i %i %i\n", a, *pta, *pt1a);
23
24          *pt1a=22; /* aici se modifica valoarea lui a */
25
26          printf("valorile numerice sunt: %i %i %i\n", a, *pta, *pt1a);
27
28          return 0;
29  }
```

Rezultate:

```
adresa lui a: 004237A4
continutul lui a: 50

continutul lui pta: 004237A4
valoarea la care poanteaza pta: 50

continutul lui pt1a: 004237A4
valoarea la care poanteaza pt1a: 50

valorile numerice sunt: 50 50 50
valorile numerice sunt: 22 22 22
```

În programul anterior `a` este o variabilă întreagă, iar `pta` este un poantor la întreg. În aceste condiţii `&a` are ca rezultat adresa variabilei `a` din segmentul de date. Această adresă, prin operatorul de atribuire, se copiază - vezi `linia 9` - în poantorul `pta`. Din rezultatele afişate de program se observă că poantorii stochează tot numere, însă acestea sunt intepretate ca şi adrese. Prin "tradiţie", adresele se scriu în notaţie hexazecimală, ceea ce ajută la distingerea lor de valorile numerice "normale".

Afişarea poantorilor

Valoarea unui poantor se poate afişa folosind în `printf()` specificatorul de format `%p`.

Reprezentarea spaţiului de date

În programul **PT1.C** se declară 3 variabile: `a` - variabilă de tipul întreg, `pta` - variabilă poantor la întreg globală, `pt1a` - variabilă poantor la întreg locală. Acesta din urmă stochează adresa variabilei de tipul `int`. `pta` fiind un

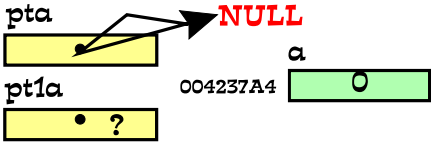


Figura 6

poantor global este inițializat cu o valoare specială numită **`NULL`**, despre care voi discuta pe larg într-un paragraf următor. Variabila **`a`** fiind globală este **`automat inițializată`** cu valoarea **`0`**. Pentru a ușura înțelegerea problemei, prezint în *Figura 6* spațiul de date al programului anterior.

Pentru **`pt1a`** spațiul se alocă pe stivă, iar acolo unde s-a făcut această alocare sunt stocate câteva valori imprevizibile. Întrucât acestea sunt lipsite de utilitate pentru programator, prefer să zic că variabilele "nu au nici o valoare", iar în reprezentarea spațiului de date am folosit "**`?`**" pentru a simboliza situația. Variabila **`a`** are stocată în ea valoarea **`0`**.

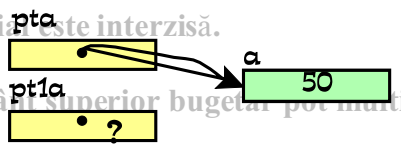


Figura 7

În urma execuției **`liniilor 9`** și **`10`**, **`pta`** va poanta la **`a`**, iar în **`a`** se va stoca valoarea **`50`**, deci reprezentarea spațiului datelor devine cea din *Figura 7*. Din acest moment avem un poantor la variabilă întreagă **`a`** și o valoarea stocată în ea, care va putea fi manipulată, fie prin numele variabilei **`a`**, fie prin poantorul la aceasta **`pta`**, după cum se va vedea în continuare. Variabila **`pt1a`** este și ea tot o variabilă poantor la un tip **`int`**. Prin atribuirea din **`linia 11`**, adresa din **`pta`** se copiază și în **`pt1a`**, deci **`pt1a`** va poanta în același loc cu **`pta`**. Situația este ilustrată în *Figura 8*.

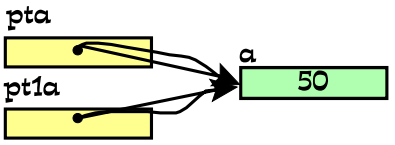


Figura 8

Reguli de atribuire a adreselor

Un poantor poate poanta numai la o variabilă de același tip cu tipul poantorului folosit în declarația acestuia. Motivul acestei interdicții stă în faptul că prin intermediul unui poantor este posibil să avem acces la valoarea numerică de la adresa poantata de poantor. Tipul poantorului specifică cum anume va fi tratată valoarea de la acea adresă. Ar fi tragic ca prin intermediul poantorilor să fie permisă, de exemplu, tratarea unei valori de tipul **`int`** ca una de tipul **`double`**. Un compilator bun trebuie să sancționeze orice tentativă de acest fel.

8.3 Operatorul de indirectare, "*"

`Operatorul de indirectare` "*" este într-un fel opusul celui adresă. Dacă "**`&`**" are ca rezultat adresa unei variabile, operatorul "*" folosește această adresă stocată într-o variabilă poantor pentru accesarea conținutului locației de memorie corespunzătoare. Operatorul de indirectare "*" accesează o valoare numerică indirect - de aici și numele de indirectare - prin intermediul unui poantor. Operandul trebuie să fie o valoare de poantor. Rezultatul operației este valoarea adresată de operand, adică valoarea de la adresa la care poantează operandul. Tipul rezultatului este tipul pe care operandul îl adresează. Se mai spune că operatorul "*" realizează **`dereferențierea`** poantorului, adică îl covertește într-o **`l-value`**.

În programul **PT1.C**, pentru că **`pta`** poantează la **`a`**, folosirea indirectării, adică a formei de scriere ***`pta`**, permite manipularea valorii întregi stocate în variabila **`a`**. În orice loc din program în care este permisă utilizarea lui **`a`**, începând cu **`linia 10`**, este permisă și utilizarea lui ***`pta`**. Semnificația lor este identică atât timp cât **`pta`** poanteaza la **`a`**. În **`linia 11`**, **`pt1a`** este făcut să poanteze la aceeași adresă cu **`pta`**, deci de acum și **`pt1a`** poantează la **`a`** și ***`pt1a`** se poate folosi în locul lui **`a`** pentru că sunt identice. **`printf()`** din **`linia 22`**, în care

se folosesc toate cele 3 modalităţi de identificare a variabilei `a`, afişează de 3 ori aceeaşi valoare. Deşi sunt declarate 3 variabile, în realitate spaţiul folosit pentru stocarea valorii întregi este unul singur, corespunzător variabilei `a`.

Modificarea valorii poantate de poantor

Până acum în programul prezentat am extras valoarea numerică poantată de poantori. Este posibilă însă şi modificarea acestei valori sub controlul operatorului de atribuire. Aceasta se ilustrează în `linia 24` în care i se atribuie valoarea `22` variabilei `a`, prin `*pt1a`. `printf()` din `linia 26` cauzează afişarea a noii valori, de `22`, de trei ori. Avem deci trei nume alternative pentru o singură variabilă pe `a`, `*pta` şi `*pt1a`. În *Figura 9* se prezintă spaţiul de date pentru această situaţie.

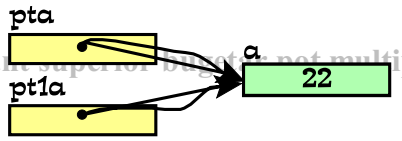


Figura 9

O eroare comună

Programul care urmează conţine o eroare - în jargonul programatorilor această eroare se numeşte ploşniţă sau `bug` în engleză.

```
1  /* PTR2.C */
2
3  #include<stdio.h>
4
5  int main(void)
6  {
7          int b, *ptb;
8
9          b=13;
10         printf("b= %i\n");
11
12         *ptb=7;
13         printf("b= %i\n");
14
15         return 0;
16 }
```

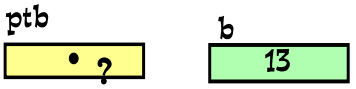


Figura 10

Starea spaţiului de date a programului, când se ajunge la `linia 12`, se prezintă în *Figura 10*. Probabil speraţi ca prin linia `*ptb=7;` să se modifice valoarea stocată în variabila `b`. Din păcate, linia `ptb=&b;` lipseşte şi pentru că `ptb` este o variabilă locală `auto`, va fi iniţializată cu o valoare aleatoare. Presupunând ca tipul `int` se stochează pe 4 octeţi, `linia 12` va suprascrie 4 octeţi din RAM începând cu adresa stocată în `ptb`. Este foarte puţin probabil ca cei 4 octeţi să fie chiar cei corespunzători lui `b`. Practic, scrierea se poate face oriunde, inclusiv în segmentul de cod, caz în care programul va "crăpa" subit. Scrierea în segmentul de date, în stivă şi pe heap este permisă, însă va avea ca efect modificarea la valori aleatoare a unor obiecte care vor duce cel puţin la rezulate eronate. Este posibil să apară şi un caz mai deosebit de scriere în afara zonelor de adresă puse la dispoziţia programului de către sistemul de operare. Dacă sistemul de operare lucrează cu protecţie (UNIX sau Windows NT) programul este ucis înainte de a apuca să facă ceva rău. Sub vechiul MS-DOS, el va corupe o parte din propriul lui cod, al altui program sau cel al DOS-ului. Rezultate "ciudate" vor fi observate în programul actual sau în altele, după caz. În `Windows`, o astfel de eroare produce terminarea aplicaţiei cu mesajul "`General Protection Fault`".

Iniţializarea poantorilor

Cea mai sigură metodă de iniţializare a unui poantor este iniţializarea în linia declaraţiei lui. Astfel, pentru exemplul anterior, în loc `int b, *ptb;` se poate scrie `int b, *ptb=&b;`.

8.4 Ce este NULL?

Am văzut deja că există posibilitatea definirii unor constante folosind directiva `#define` a preprocesorului. `NULL` este o astfel de valoare definită în mai multe fişiere antet (de exemplu în `stdio.h`). Ea indică faptul că poantorul care ia această valoare specială nu poantează adresa unui obiect C. Dacă declaraţia unei variabile poantor este globală, compilatoarele ANSI vor iniţializa variabila poantor cu valoarea specială `NULL`. Secvenţa de biţi corespunzătoare lui `NULL` nu este neapărat valoarea numerică zero, fiind dependentă de compilatorul specific pe care lucrăm. Pentru a face codurile sursă compatibile între diferite compilatoare a fost necesară reprezentarea poantorului `NULL` printr-un macro. `NULL` este de fapt numele acestui macro. Setarea unui poantor folosind acest macro se face printr-o atribuire de forma `ptb = NULL`. De asemenea, această valoare specială poate fi testată în expresii pentru a verifica dacă un poantor este sau nu iniţializat.

```
1  /* PTR3.C */
2  #include<stdio.h>
3
4  int *ptb;
5  int main(void)
6  {
7          int b;
8          /* int *ptb=NULL; */
9          b=13;
10         printf("b= %i\n",b);
11
12         if (ptb == NULL)
13                 printf("poantorul ptb este invalid!\n");
14         else
15         {
16                 *ptb=7;
17                 printf("b= %i\n",b);
18         }
19
20         return 0;
21 }
```

În programul anterior, `ptb` este o variabilă globală, motiv pentru care este iniţializată automat cu valoarea `NULL`. Dacă ar fi fost locală, iniţializarea trebuia făcută explicit de noi ca şi în `linia 8`. În cazul când iniţializarea nu se face nici automat, nici explicit, avem o problemă, pentru că nu avem de unde să ştim dacă valoarea poantată este una reală sau "gunoi", detecţia cu un `if`, ca în exemplul anterior, neputându-ne salva de la dezastru.

8.5 Poantorii şi argumentele funcţiilor

În programul `EXTERN.C` din paragraful 6.6 am prezentat un program care citea două numere reale, apoi calcula şi afişa cătul lor. Implementarea dată folosea variabile globale. Implementarea care urmează foloseşte variabile locale şi transferul de parametri prin

poantori. Am spus deja la apelul prin valoare că funcţia apelată nu are cum să modifice valorile variabilelor pe post de argumente din funcţia apelantă pentru că se transferă numai o copie a lor prin stivă. Soluţia este să apeleze funcţia cu poantori la parametri. Majoritatea programatorilor aleg transferul valorilor de obiect prin adresă şi nu prin valoare, din motive de creştere a vitezei de execuţie a programului. De asemenea, metoda de transfer este singura care permite modificarea valorilor variabilelor ale căror adrese s-au transferat ca argumente din interiorul funcţiei care s-a apelat.

```
/* APELPTR.C */
#include<conio.h>
#include<stdio.h>
#include<float.h>

/*Prototipurile functiilor din program */
void citeste_2_reali(float *deimpartit, float *impartitor);
float calculeaza_citul(float deimpartit, float impartitor);
void afiseaza_citul(float cit);

const float INFINIT = FLT_MAX;

/* functia main() - punctul de intrare in program*/
int main(void)
{
        float cit,deimpartit,impartitor; /* 3 variabile locale */
        do {
                        citeste_2_reali(&deimpartit,&impartitor);
                        cit=calculeaza_citul(deimpartit,impartitor);
                        afiseaza_citul(cit);
        } while (getch() != 'i');

        return 0;
}
/*sfarsitul functiei main()*/

/* definitiile functiilor */
void citeste_2_reali(float *deimpartit, float *impartitor)
{
        printf("Baga 2 numere, deimpartit si impartitor: ");
        scanf("%f %f",deimpartit,impartitor);
}

float calculeaza_citul(float deimpartit, float impartitor)
{
        if (impartitor == 0.0F)
                return(INFINIT);
        else
                return(deimpartit/impartitor);
}

void afiseaza_citul(float cit)
{
        if (cit == INFINIT)
                printf("Citul este nedefinit\n");
        else
                printf("Citul este %f\n",cit);
}
```

Liniile modificate faţă de implementarea din programul **EXTERN.C** s-au marcat prin culoarea gri a fondului. În **linia 18**, prin **citeste_2_reali(&deimpartit,&impartitor);** se apelează funcţia **citeste_2_reali()** cu două argumente ce sunt adresele variabilelor **deimpartit** şi **impartitor**. Evident, declaraţia funcţiei a fost şi ea modificată de la **void citeste_2_reali();** la **void citeste_2_reali(float *deimpartit, float *impartitor);**. Se observă că cei doi parametri sunt poantori la tipul **float**, ca urmare orice modificare a lor în corpul funcţiei **citeste_2_reali()** se va reflecta şi asupra valorilor variabilelor **deimpartit** şi **impartitor** din funcţia apelantă **main()**. La revenirea din **citeste_2_reali()** valorile variabielor **deimpartit** şi **impartitor** vor fi modificate datorită funcţiei **scanf()**. Aceasta, după primul argument are ca argumente de intrare adresele variabilelor. Din acest motiv, de obicei, numele variabilelor în care se vor citi valori trebuie să fie precedate de operatorul **&**. Aici nu este cazul pentru că aceste variabile stochează ele însele adrese, fiind poantori.

8.6 Funcţii cu număr variabil de parametri

Unele funcţii C cum ar fi **printf()** acceptă un număr variabil de argumente pe lângă un număr fixat de argumente. Declaraţia de funcţie unde ultimul argument din lista argumentelor este o virgulă urmată de suită de 3 puncte "..." va lucra cu un număr variabil de argumente. Este necesar să existe cel puţin un argument fix, chiar dacă acesta nu va fi folosit ulterior, pentru a putea accesa lista argumentelor variabile.

Varianta portabilă de acces la argumentele acestor funcţii s-a implementat în C prin intermediul unor macrodefiniţii ale căror nume încep cu **va_...**. Acestea sunt folosite pentru a parcurge lista argumentelor când funcţia apelată nu ştie numărul şi tipul acestora. Prototipurile funcţiilor utilizate în acest scop sunt:

```
tip va_arg(va_list arg_ptr, tip);
void va_end(va_list arg_ptr);
void va_start(va_list arg_ptr, prev_param);
```

iar fişierul de prototip este **"stdarg.h"**.

va_list este un tablou care ţine informaţia necesară lui **va_arg** şi **va_end**. Când funcţia apelată foloseşte un număr variabil de argumente în ea se declară o variabilă **arg_ptr** de tipul **va_list**.

va_start este un macro care cere 2 parametri, **arg_ptr** şi **prev_param** ce vor fi explicaţi în continuare.

va_arg este un macro care devine o expresie de acelaşi tip şi valoare cu următorul argument de transferat. La prima utilizare a lui **arg_ptr** întoarce primul argument din listă. **prev_param** este numele parametrului obligatoriu precedând imediat primul argument opţional din listă. Fiecare utilizare succesivă întoarce următorul argument al listei. Aceasta se face prin dereferenţierea lui **prev_param** şi apoi prin incrementarea lui pentru a poanta la următorul parametru. **va_arg** foloseşte **tip** pentru a realiza dereferenţierea şi localizarea următorului argument. La fiecare apel succesiv a lui **va_arg** se modifică **prev_param** pentu a poanta la următorul parametru din listă.

va_end ajută funcţia apelată să revină la normal. **va_end** poate modifica pe **prev_param** aşa încât să nu mai poată fi folosită decât după reapelarea lui **va_arg**.

va_end va fi apelată după ce **va_arg** a citit toţi parametrii, altfel programul putând să o ia razna. Programele care urmează ilustrează modul de lucru cu aceste macrodefiniţii:

```
/* VALST1.C */
#include<stdio.h>
#include<stdarg.h>
#include<limits.h>

void minim(char *format, ...)
{
        int min;
        va_list arg_ptr;
        int arg;

        va_start(arg_ptr, format);
        min=va_arg(arg_ptr, int);
        printf("%d ", min);

        while ((arg = va_arg(arg_ptr, int)) != INT_MAX)
        {
                printf("%i ", arg);
                if (min > arg) min=arg;
        }

        va_end(arg_ptr);
        printf(format, min);
}

int main(void)
{
        minim("\nMinimul este: %i\n",1,12,2,3,4,5,3,INT_MAX);

        return 0;
}
```


Rezultate:

```
1 12 2 3 4 5 3
Minimul este: 1
```

Pentru programul următor, funcţional în varianta prezentată mai jos, vă propun să încercaţi să-l faceţi să lucreze cu valori **float** în loc de **double**.

```
/* VALST2.C */
#include<stdio.h>
#include<stdarg.h>

double medar(int n, double *max, ...)
{
        double s, arg;
        int i;
        va_list arg_ptr;

        va_start(arg_ptr, max);

        *max=va_arg(arg_ptr, double);
        printf("%5.3f ", *max);
```

```
16      for (i=1;i<n;i++)
17      {
18              s+= (arg = va_arg(arg_ptr, double));
19              printf("%5.3f ", arg);
20              if (*max < arg) *max=arg;
21      }
22
23      va_end(arg_ptr);
24      return s/n;
25  }
26
27  int main(void)
28  {
29      double fmax;
30      printf("\nMedia aritmetica este: %10.5f\n", medar(4, &fmax, 1.0, 4.0,3.0, -4.0));
31      printf("Maximul este: %f\n", fmax);
32
33      return 0;
34  }
```

Rezultate:

```
1.000 4.000 3.000 -4.000
Media aritmetica este:    0.75000
Maximul este: 4.000000
```

Când argumente de tip **char** sunt transferate ca argumente variabile ele se convertesc automat la tipul **int**. Similar, argumentele de tip **float** se convertesc automat la tipul **double** în cazul argumentelor variabile.

8.7 Poantori la poantori

C permite declararea de poantori la orice tip, inclusiv declararea unui poantor la un poantor la un tip oarecare.

```
1   /* PTRPTR.C */
2   #include<stdio.h>
3
4   int main(void)
5   {
6       int a=7;
7       int *pta;
8       int **ptpta;
9
10      pta=&a;
11      ptpta=&pta;
12      printf("%p     a: [%i]\n",&a,a);
13      printf("%p   pta: [%p] -> [%i]\n",&pta,pta,*pta);
14      printf("%p ptpta: [%p] -> [%i]\n",&ptpta,ptpta,**ptpta);
15
16      return(0);
17
18  }
```


Rezultate:

```
0065FDF4     a: [7]
0065FDF0   pta: [0065FDF4] -> [7]
```

```
0065FDEC ptpta: [0065FDF0] -> [7]
```

În programul anterior `a` este o variabilă de tipul `int`, `pta` este un poantor la un tip `int`, iar `ptpta` este un poantor la un poantor de tipul `int`. Spaţiul datelor, în starea finală, se prezintă în *Figura 11*. În desen mărimea locaţiilor simbolizate sub forma dreptunghiurilor este aceeaşi. Am presupus că atât tipul `int` cât şi poantorii se reprezintă pe 4 octeţi, iar unui grup de 4 octeţi îi corespune câte un dreptunghi.

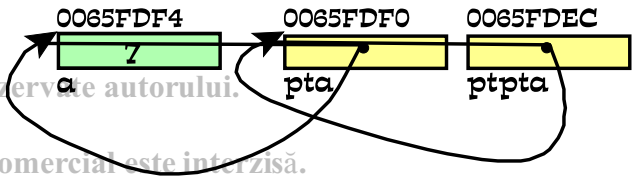


Figura 11

Deasupra fiecărui dreptunghi stă scrisă adresa de început a primului dintre cei 4 octeţi consecutivi, folosiţi la stocarea respectivei variabile.

8.8 Poantori la funcţii

În C, numele unei funcţii este un poantor constant la începutul codului funcţiei. Pentru că numele funcţiei este definit ca un poantor la funcţia respectivă, numele funcţiei se poate atribui unei variabile poantor la funcţie. O variabilă poantor la funcţie poate fi transferată ca parametru unei alte funcţii şi poate fi folosită într-o funcţie pentru apelarea funcţiei la care poantează. Modul declaraţiei variabilei poantor la funcţie se prezintă în continuare.

Formă:	**tip (*variabila_poantor_la_funcţie) (listă de parametri)**

Nu este permisă efectuarea de operaţii aritmetice cu o variabilă poantor la funcţie. De asemenea ea poate "lua valoarea" în urma unei atribuiri, numai a unei funcţii cu acelaşi număr de parametri şi cu acelaţi tip de valoare întoarsă cu cele specificate în declaraţia ei. Un exemplu se prezintă în programul următor:

```
1  /* PTRFNC.C */
2  #include <stdio.h>
3  #include <ctype.h>
4  #include <conio.h>
5  char iesire(int c);
6  char fisiere(int c);
7  char editare(int c);
8  char altcaracter(int c);
9
10 /* ptrfunctie se declara ca poantor la o functie ce primeste ca parametru de intrare
11 o valoare int si întoarce o valoare char */
12 char (*ptrfunctie)(int);
13
14 main()
15 {
16       int ch;
17       for(;;)
18          {
19                ch=toupper(getch());
20                switch (ch) {
21                      case 'I': ptrfunctie=iesire; break;
22                      case 'F': ptrfunctie=fisiere; break;
23                      case 'E': ptrfunctie=editare; break;
24                      default: ptrfunctie=altcaracter;
```

```
25          }
26                  /* aici se apeleaza functia prin poantorul la ea */
27                  if ('I' == (ptrfunctie)(ch)) goto treispe;
28          }
29
30  treispe: printf("Gata, ai IESIT !\n");
31  }
32
33  char iesire(int c)
34  {
35          printf("\ai apasat %c si esti in functia: IESIRE\n",c);
36          return c;
37  }
38
39  char fisiere(int c)
40  {
41          printf("ai apasat %c si esti in functia: FISIERE\n",c);
42          return c;
43  }
44
45  char editare(int c)
46  {
47          printf("ai apasat %c si esti in functia: EDITARE\n",c);
48          return c;
49  }
50
51  char altcaracter(int c)
52  {
53          printf("ai apasat %c si NU STIU ce doresti ... \n",c);
54          return c;
55  }
```


Aici linia `char (*ptrfunctie)(int)` declară un poantor cu numele `ptrfunctie` la o funcţie care are un parametru de intrare `int` şi întoarce o valoare `char`. Toate funcţiile care sunt apelate prin acest poantor la funcţie întorc tipul `char` şi un singur parametru de intrare de tipul `int`.

O variabilă poantor la o funcţie ascunde o facilitate foarte puternică de programare. Nu recomand această tehnică de lucru începătorilor, dar am menţionat-o întrucât conceptul intimidează, fiind privit ca ceva dificil de abordat. După cum vedeţi nu este chiar aşa!

Universitatea Tenhică din Cluj-Napoca
Facultatea Construcţii de Maşini
Catedra de Mecanică şi Programare
Curs de limbaj C

9 Tablouri

Tabloul este o colecţie de date identice ca tip, distincte însă prin indice. Numărul de dimensiuni ale tabloului depinde de limbaj, dar în general, nu este limitat. O variabilă obişnuită - un scalar - poate fi privită ca un tablou cu dimensiunea zero. Un tablou cu o singură dimensiune este cunoscut sub numele de "vector", iar unul cu două dimensiuni sub denumirea de "matrice". În majoritatea limbajelor imperative, o referinţă la un element de tablou se scrie sub forma **a[i,j]** unde, **a** este **numele tabloului**, iar **i** şi **j** sunt **indici**. Elementele tabloului sunt de obicei stocate în locaţii de memorie consecutive. Limbajele diferă de obicei prin metoda de stocare a datelor pe rânduri continue sau pe coloane continue.

Tablourile sunt folosite pentru stocarea unui grup de valori care se vor accesa într-o ordine imprevizibilă (de exemplu, o structură de dată numită listă se pretează mult mai bine pentru stocarea unor valori care se accesează secvenţial).

9.1 Declararea tablourilor

Un tablou se declară folsind un **tip**, un **nume** de variabilă şi o listă de constante cuprinse între paranteze dreptunghiulare care definesc **dimensiunile** tabloului.

Formă: `tip nume[dimensiune1][dimensiune2] ... [dimensiuneN];`

Pentru a declara un tablou cu **o singură dimensiune** de **7** elemente de tipul **int**, cu numele **a**, scriem:

```
int a[7];
```

În C este o regulă că spaţiul pentru cele **7** elemente ale tabloului **a** se vor aloca consecutiv în memoria fizică. Astfel, putem spune de exemplu că al doilea element este adiacent cu al treilea, al patrulea cu al cincelea ş.a.m.d.

Pentru a declara un tablou cu **2 dimensiuni**, prima de **5**, a doua de **3** de tipul **float**, cu numele **m**, scriem:

```
float m[5][3];
```

C poate lucra cu orice tipuri de elemente, inclusiv structuri sau reuniuni (care vor fi discutate în capitolele următoare), cu excepţia tipului **void**. Este obligatoriu ca între parantezele dreptunghiulare să fie scrisă o expresie constantă, întreagă şi pozitivă. În locul declaraţiei anterioare a lui **m** se poate scrie:

```
#define D1 5
#define D2 3
float m[D1][D2];
```

Nu sunt însă corecte următoarele forme de declaraţie:

```
int d1=5;                          const int c1=5;
int d2=3                    şi     const int c2=3
float m[d1][d2];                   float m[c1][c2];
```


Deşi `c1` şi `c2` sunt evident constante, compilatorul nu va accepta această formă de scriere. Între parantezele drepte pot să apară numai expresii constante (termenii acestor expresii sunt valori numerice constante).

Iniţializarea tablourilor

Un tablou este format din elemente. Iniţializarea tabloului presupune iniţializarea fiecărui element de tablou. Sintactic, definiţia iniţializatorului de tablou este recursivă şi are forma:

ANTAL Tiberiu Alexandru
tel.: 0040-264-530918
e-mail: antaltiberiu@pcnet.ro

```
<iniţializator>::= expresie constantă | <{listă de iniţializare}> | <{listă de iniţializare,}>
<listă de iniţializare>::= <iniţializator> | <listă de iniţializare>, <iniţializator>
```

listă de iniţializare este o listă **iniţializator** separată prin virgule. Fiecare **iniţializator** din listă este, fie o expresie constantă, fie o altă **listă de iniţializare**. Astfel, listele pot fi cuprinse unele în altele.

Dacă tabloul are o singură dimensiune, se scrie o singură listă cu valorile de iniţializare separate prin virgule.


```
int a[7] ={1,-4,2,28,7,67,99};
```

Sub această formă de scriere toate cele `7 elemente` ale tabloului `unidimensional a` sunt iniţializate. Ordinea valorilor din listă este şi cea de atribuire la elementele de tablou. Astfel, primul element va lua valoarea `1`, al doilea `-4`, etc. Dacă numărul de valori este mai mic decât cel al elementelor de tablou, iniţializarea se face la fel, adică în ordinea valorilor din listă până când se epuizează toate valorile listei. Elementele rămase neiniţalizate vor fi iniţializate automat cu valoarea `0`.


```
int b[7]={1};
```


Mai sus, primul element din `b` este iniţializat cu valoarea `1`, celelalte cu valoarea `0`. Un alt caz este când numărul de elemente ale tabloului nu sunt specificate, dar sunt specificate un număr oarecare de valori pentru iniţializare. În acest caz compilatorul va fixa numărul de elemente ale tabloului la cel al valorilor de iniţializare, atribuirea acestora la elementele de tablou facându-se în ordinea apariţiei în listă.

ANTAL Tiberiu Alexandru
tel.: 0040-264-530918
e-mail: antaltiberiu@pcnet.ro

```
int c={1,2,3,4};
```

Mai sus `c` este un tablou `unidimensional` cu `4 elemente`, primul este iniţializat cu valoarea `1`, al doilea cu valoarea `2`, etc.

Iniţializarea unui tablou bidimensional se face astfel:

```
int m[3][2]= {
                {2, 8},
                {0, 7},
                {6, 7,},
             };
```

Tabloul `m` are **două dimensiuni**, **prima** cu **3 elemente**, a **doua** cu **2**. El corespunde unei **matrice** cu **3 linii** şi **2 coloane**. Se iniţializează **prima linie** cu **2** respectiv cu **8**, a **doua** cu **0** şi **7** etc. Observaţi că iniţializatorii din **linia 3** au o virgulă după ultima expresie constantă. Aceste virgule sunt permise, dar nu obligatorii şi separă o expresie constantă de alta. Cele care separă o listă (**listă de iniţializare**) de alta sau un iniţializator (**iniţializator**) de altul sunt însă obligatorii.

Limitări ale iniţializărilor

Nu se poate specifica un factor de repetiţie pentru iniţializări sau iniţializa un element "din mijloc", fără iniţializarea elementelor anterioare.

9.2 Accesul la elementele de tablou

Numerotarea elementelor de tablou începe de la zero.

```
1  /* TAB1.C */
2  #include <stdio.h>
3  int main(void)
4  {
5          int a[7], d1, i;
6
7          d1=sizeof a/sizeof a[0]; /* d1 = 7 nr. elementelor de tablou */
8
9  /* elementele primesc valori intr-o ordine oarecare */
10         a[0]=5; a[2]=13; a[5]=67; a[6]=28; a[1+2]=3; a[3/2]=1; a[2*2]=7;
11
12         printf("Elementele tabloului a sunt: ");
13
14         for (i=0;i<d1;i++)
15                 printf("%-4i", a[i]);
16
17         printf("\n   Indicii tabloului a sunt: ");
18
19         for (i=0;i<d1;i++)
20                 printf("%-4i", i);
21
22         printf("\n  Adresele elementelor sunt: ");
23
24         for (i=0;i<d1;i++)
25                 printf("%-9p", &a[i]);
26         printf ("\n");
27
28         return 0;
29 }
```

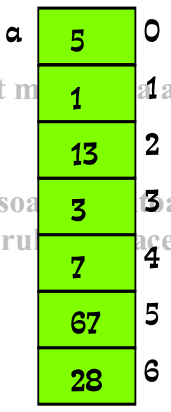


Figura 12

Rezultate:

```
Elementele tabloului a sunt: 5    1    13   3    7    67   28
   Indicii tabloului a sunt: 0    1    2    3    4    5    6
  Adresele elementelor sunt: 0065FDDC 0065FDE0 0065FDE4 0065FDE8 0065FDEC 0065FDF0 0065FDF4
```

Observaţi că adresele elementelor de tablou sunt consecutive. Primul element are indicele `zero`, al doilea `unu`, ultimul indice ia valoarea numărului total de elemente ale tabloului minus o unitate. În *Figura 12* se prezintă spaţiul de date al programului corespunzător variabilei tablou `a`. Tabloul are `o dimensiune` şi `7 elemente`. Conform celor spuse, indicii pot varia de la `0` la `6=7-1`. În C nu se verifică dacă indicele se încadrează în domeniul permis. Este simplu să scriem de exemplu `a[7]=0` care are un indice incorect. Pentru acest caz, orice ar fi stocat în locaţiile de memorie următoare lui `a[6]` s-ar suprascrie cu valoarea lui `a[7]`, valorile numerice stocate acolo prin intermediul altor variabile devenind corupte.

9.3 Operatorul de indexare al tablourilor

Formă:	**expresie1[expresie2]**

O `expresie1` urmată de operatorul `[expresie2]` specifică indexarea unui tablou. Una dintre expresii trebuie să fie un poantor sau un tip tablou, adică să fie declarată sub una dintre formele: `tip*`, `tip[ ]`. Cealaltă trebuie să fie de tip întreg. Forma de utilizare comună este cea în care expresia de tip întreg este între parantezele drepte.

Operatorul adresă este comutativ, deci `expresie1[expresie2]` şi `expresie2[expresie1]` sunt echivalente pentru că rezultatul operatorului de indexare se calculează prin definiţie cu expresia: `*((expresie2)+(expresie1))`.

```
int a[10],a2;
a2=a[1];
a2=1[a];
```

Primul element al taboului este `0`. Astfel, domeniul indicelui pentru un `tablou` cu `o dimensiune` şi `n` elemente este de la `tablou[0]` până la `tablou[n-1]`.

9.4 Conversia numelor de tablouri în poantori

Dacă tipul unei expresii sau subexpresii este "tablou cu elemente de tipul T", valoarea acestei expresii se va descompune într-un poantor la primul element din tablou, iar tipul expresiei va fi poantor la tipul T. Această conversie nu se va realiza dacă expresia este operandul următorilor operatori: operatorul adresă (`&`), operatorul de incrementare (`++`), operatorul de decrementare (`--`) sau operatorul `sizeof`.

```
1  /* TAB2.C */
2  #include<stdio.h>
3
4  int main(void)
5  {
6          float x[10];
7          float *ptx, *pt1x;
8
9          pt1x=&x[0];
```

```
10      ptx=x;
11
12      printf("adresa primului element (x[0]) din tabloul x este: %p\n", &x[0]);
13      printf("ptx= %p si pt1x =%p", ptx, pt1x);
14
15      return 0;
16  }
```

Rezultate:

```
adresa primului element (x[0]) din tabloul x este: 0065FDD0
ptx= 0065FDD0 si pt1x =0065FDD0
```

În *Figura 13* se prezintă spaţiul de date al programului **TAB2.C**. Observaţi că adresa primului element de tablou se poate găsi prin două metode, fie prin aplicarea operatorului adresă asupra primului element de tablou, care se scrie **&x[0]**, fie prin aplicarea aceluiaşi operator asupra numelui tabloului, care se scrie **&x**.

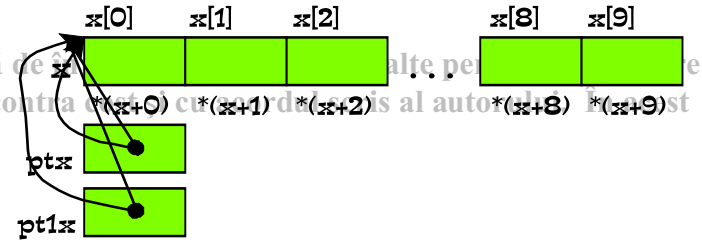


Figura 13

Adresa primului element se converteşte într-un poantor constant la primul element al tabloului, motiv pentru care valoarea lui nu se poate modifica. Orice încercare de modificare a acestei adrese este sancţionată de compilator, lucru normal din moment ce dacă această adresă ar putea fi modificată, compilatorul ar putea "uita" adresa de început a tabloului. Ca urmare a conversiei prezentate la operatorul de indexare se poate afirma mai mult. În C numele unui tablou poate fi folosit ca un poantor şi invers, se poate indexa un poantor ca un tablou. Dacă **x** este un tablou unidimensional, următoarele expresii sunt echivalente: **(x + i) == &(x[i]), *(x + i) == x[i]**

Diferenţe între tablou şi poantor la tablou

Există câteva diferenţe între un poantor la un tablou (**ptx**, **pt1x**) şi un tablou (**x[D1]**). Prima este alocarea de memorie. La declararea unui tablou, compilatorul alocă spaţiul de memorie specificat prin numărul de elemente ale dimensiunii automat. La declararea unui poantor la un tablou, utilizatorul trebuie să aloce explicit spaţiul pe heap prin folosirea uneia dintre funcţiile **malloc()** sau **calloc()**, eventual să atribuie poantorului o adresă a unui spaţiu deja alocat. A doua diferenţă este aceea că numele tabloului devine un poantor constant la primul element de tablou. Fiind poantor constant, valoarea la care poantează nu se poate modifica, în concluzie nu se pot efectua manipulări ale respectivei adrese. De exemplu, nu se poate scrie **x++**, însă se poate scrie **ptx++**.

9.5 Calculul poziției și a valorii celui de al i-elea element de tablou

Elementul **i**- al tabloului **x** (vezi *Figura 13*) este accesat prin expresia **x[i]**. Pentru tabloul **x unidimensional** de reali - elemente de tipul **float** - procedura de calcul este următoarea:

1. Valoarea întreagă a lui **i** este înmulţită cu numărul de octeţi folosiţi pentru stocarea tipului **float**. Această valoare calculată va reprezenta a **i**-a poziţie a tipului **float** faţă de adresa de început a lui **x** şi se numeşte offset sau, pe românește, deplasament .

2. Această valoare se adună la valoarea poantorului **x**, pentru a forma o adresă care

se află la `i` poziţii `float` faţă de adresa lui `x`.

3. Se aplică operatorul de indirectare (*) asupra acestei adrese, valoarea rezultată fiind valoarea elementului al `i`-lea din `x`.

Conform celor spuse putem scrie `x[i] = *(x+i*sizeof(float))`.

`x[0]` reprezintă valoarea primului element din `x`, din moment ce offsetul corespunzător lui `x` este `0`. O expresie de forma `x[5]` referă elementul cu offsetul `5` din `x`, adică al `6`-lea element din tablou.

9.6 Aritmetica poantorilor

Puterea limbajul C vine din modul consistent şi sistematic de integrare într-un tot al tablourilor, poantorilor şi al aritmeticii adreselor. Operaţii valide cu poantori sunt: atribuirea între poantori de acelaşi tip; adunarea sau scăderea unui poantor la un întreg; adunarea, scăderea şi comparaţia între elementele aceluiaşi tablou; atribuirea şi comparaţia cu `NULL`.

Adunarea dintre un poantor şi un întreg

Este corectă adunarea unui poantor la un element de tablou cu un întreg. Adunarea lor decurge astfel: valoarea întreagă este convertită într-un offset prin înmulţirea acesteia cu dimensiunea elementului de tablou la care poantează poantorul, apoi se face suma dintre valoarea stocată în poantor şi offset. Suma va fi un poantor de acelaşi tip cu poantorul original. El va poanta la un obiect al aceluiaşi tablou, pe baza valorii de offset calculate. Dacă `ptr` este un poantor la un element de tabloul de tipul `T`, atunci `ptr+1` va poanta la următorul element din tabloul de tipul `T`. Dacă suma rezultată este în afara limitelor tabloului, rezultatul este nedefinit.

```
1  /* TAB3.C */
2  #include<stdio.h>
3
4  int main(void)
5  {
6     int x[5]={1,2,3,4,5};
7
8     int *ptr;
9
10    ptr=x;
11    printf("ptr    = %p, *ptr= %i, x[0] =%i\n", ptr, *ptr, x[0]);
12
13    ptr++;
14    printf("ptr++  = %p, *ptr= %i, x[1] =%i\n", ptr, *ptr, x[1]);
15
16    ptr+=3;
17    printf("ptr+=3 = %p, *ptr= %i, x[4] =%i\n", ptr, *ptr, x[4]);
18
19    return 0;
20 }
```

Rezultate:

```
ptr    = 0065FDE4, *ptr= 1, x[0] =1
ptr++  = 0065FDE8, *ptr= 2, x[1] =2
ptr+=3 = 0065FDF4, *ptr= 5, x[4] =5
```

În *Figura 14* se prezintă spaţiul de date al programului anterior (**TAB3.C**). Poantorul **ptr** este iniţializat cu adresa primului element al tabloului **unidimensional x** cu **5 elemente** de tipul **int**. **printf()** din **linia 11** va afişa **1** pentru valoarea stocată la adresa poantată de **ptr**. La incrementarea poantorului, adresa stocată în acesta nu va fi **0065FDE5**, ci **0065FDE8**. Compilatorul ştie că **ptr** este un poantor la tipul **int**, care se stochează pe 4 octeţi (**sizeof (int) = 4**). Astfel, va aduna **4** la adresa **0065FDE4**, pentru incrementare. Putem spune că la acest tip de adunare, valoarea numărului întreg adunat la adresa stocată într-un poantor se "scalează" prin dimensiunea obiectului la care poantează acesta.

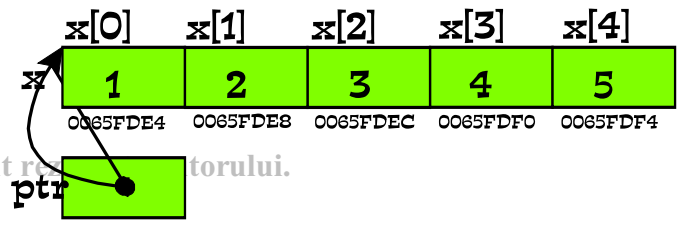


Figura 14

Scăderea dintre un poantor şi un întreg

Este corectă scăderea unui întreg dintr-un poantor. Aceasta se va realiza în aceleaşi condiţii cu adunarea.

Adunarea şi scăderea a doi poantori

Este corectă şi adunarea sau scăderea a doi poantori la obiecte de acelaşi tip. Rezultatul este un întreg cu semn reprezentând valoarea offsetului dintre cele două obiecte poantor. Poantorii consecutivi diferă prin valoarea 1. Valoarea este nedefinită dacă poantorii nu poantează la acelaşi obiect tablou.

```
1  /* TAB4.C */
2  #include<stdio.h>
3
4  int main(void)
5  {
6        double v[5]={1.1,2.2,3.3,4.4,5.5};
7        double *ptr1, *ptr2;
8
9
10       ptr1=v+1;
11       ptr2=v+4;
12
13       printf("ptr1 = %p, *ptr1= %lf, v[0] =%lf\n", ptr1, *ptr1, v[0]);
14       printf("ptr2 = %p, *ptr2= %lf, v[4] =%lf\n", ptr2, *ptr2, v[4]);
15       printf("ptr2-ptr1 = %i\n", ptr2-ptr1);
16
17       return 0;
18 }
```

Rezultate:

```
ptr1 = 0065FDD8, *ptr1= 2.200000, v[0] =1.100000
ptr2 = 0065FDF0, *ptr2= 5.500000, v[4] =5.500000
ptr2-ptr1 = 3
```

În *Figura 15* se prezintă spaţiul datelor pentru programul anterior (**TAB4.C**). Compilatorul realizează scăderea scalând valoarea găsită conform tipului la care poantează poantorii **ptr1 = 0065FDD8** şi **ptr2 = 0065FDF0**, deci **ptr2-ptr1=24**. În urma scalării acestei valori prin dimensiunea tipului la care poantează poantori (tipul **double** se

stochează pe 8 octeţi, adică `sizeof(double)` = `8`) se obţine `24/8` = `3`.

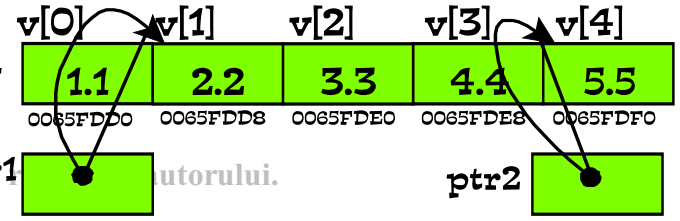


Figura 15

Compararea poantorilor

Operaţia de comparaţie dintre doi poantori se poate realiza numai dacă aceştia poantează la acelaşi tablou. În afara acestui caz, rezultatul comparaţiei este nedefinit. Este corectă comparaţia unui poantor cu valoarea `NULL`. Un caz special de comparaţie corectă este când adresa primului element după ultimul din tablou este folosită într-o comparaţie sau într-o expresie aritmetică.

Consistenţa aritmeticii adreselor

Operaţiile aritmetice cu poantori ţin cont de tipul de dată la care poantează poantorul. Toate manipulările de poantori iau automat în considerare tipul la care poantează poantorul. De exemplu, dacă incrementăm un poantor la tipul `char`, noua adresă este mai mare cu un octet (o unitate), iar dacă se incrementează un poantor la `float`, noua adresă va fi cu 4 octeţi - spaţiul de RAM folosit pentru stocarea unui tip `float` - mai mare.

9.7 Tablouri multidimensionale

Un tablou se poate construi din elemente de tip aritmetic, poantor, structură, reuniune sau un alt tablou. Pentru ultimul caz se obţin tablouri multidimensionale. Declaraţia unui tablou de elemente `float` cu numele `m`, de `două dimensiuni` şi cu `numerele de elemente` `3`, respectiv `5` se scrie:

```
#define D1 3
#define D2 5
float m[D1][D2];
```

`m` este un tablou format din `3` articole de tipul tablou, fiecare astfel de articol este la rândul lui un tablou cu `5` elemente. Astfel, oricare din expresiile `m`, `m[i]` sau `m[i][j]` sunt corecte. Primele două au tipul tablou, respectiv `m` este un tablou cu `3` elemente de tipul tablou, cu elementul de tip tablou de `5` elemente `float`, iar `m[i]` este un tablou cu elementele de tipul tablou cu `5` elemente `float`. Ultima expresie are tipul `float` şi reprezintă valoarea numerică a elementului în cauză. Operaţiile de evaluare a valorilor elementelor de tablouri se aplică şi aici. Astfel, dacă `e1` este un tablou, iar `e2` un indice, `e1[e2]` este echivalentă cu `*((e1)+(e2))`, deci se poate scrie `m[i][j]` = `*(m[i]+j)`. Asociativitatea operatorului [ ] este de la stânga la dreapta, deci `m[i][j]` = `*((m[j]) + (i*sizeof(m[j])))`. Regulile de evaluare ale acestei expresii ţin de aritmetica poantorilor. `m[j]` este un poantor la care se va aduna o valoare întreagă. În adunarea `*((m[j]) + (i*sizeof(m[j])))`, `sizeof(m[j])` este mărimea unui tablou de reali (`float`) cu numărul de elemente `D2`. Se poate scrie astfel: `*((m[j]) + i*sizeof(m[j]))` = `*(m[j] + i*D2*sizeof(float))`. În final, expresia `*(m[j] + i*D2*sizeof(float))` se converteşte în `*((m+j*sizeof(float))+i*D2*sizeof(float))`. Întrucât `m[j]` este un poantor, operatorul de indirectare nu se aplică pentru calculul adresei acestuia.

După cum am spus, C face automat toate aceste transformări şi calcule. Observaţi că prima dimensiune a tabloului (`D1`) nu participă la calculele pentru determinarea poziţiei unui element. Din acest motiv, când tablourile vor fi transmise ca parametri de funcţii, prima dimensiune poate fi omisă. Prima oară va fi evaluată subexpresia cuprinsă între

parantezele cele mai interioare, care ţine de **j**, apoi va fi evaluată subexpresia care ţine de **i**. Ultimul indice, **j**, este evaluat în expresie înaintea celuilat, **i**. Din acest motiv se zice că el variază mai repede. Elementele corespunzătoare ultimului indice vor fi stocate în zone continue de RAM. Dacă se face corespondenţa cu elementele unei matrice, $\mathbf{m}_{i,j}$, unde **i** este rândul şi **j** coloana, se poate spune că tabloul cu două dimensiuni este stocat pe rânduri.


```
1  /* TAB5.C */
2  #include<stdio.h>
3  #define D1 3
4  #define D2 2
5  #define TIP int
6  TIP m[D1][D2]= {{1,2},
7                  {3,4},
8                  {5,6}
9                  };
10 int main(void)
11 {
12       int i,j;
13
14       char *ptr;
15
16       for(i=0;i<D1;i++)
17       {
18             for(j=0;j<D2;j++)
19                   printf("%4i ",m[i][j]);
20             printf("\n");
21       }
22
23       printf("\n");
24       for(i=0;i<D1;i++)
25       {
26             for(j=0;j<D2;j++)
27                   printf("%4i ",*(*(m+i)+j));
28             printf("\n");
29       }
30
31       printf("\n");
32       ptr=m;
33       for(i=0;i<D1;i++)
34       {
35             for(j=0;j<D2;j++)
36                   printf("%4i ",*(ptr + i*D2*sizeof(TIP) + j*sizeof(TIP)));
37             printf("\n");
38       }
39
40       return 0;
41 }
42
43
```

Rezultate:

```
   1    2
   3    4
   5    6

   1    2
   3    4
   5    6
```

Linia 21 prezintă modul de acces clasic - prin operatorul de indexare - la valorile elementelor unui tablou bidimensional. **Linia 29** prezintă un mod de acces care foloseşte formulele de conversie ale adreselor de tablou în poantori, iar **linia 38** implementează formulele folosite de C pentru calculul poziţiei în RAM a unui element de tablou.

Voi numi în continuare tabloul cu două dimensiuni matrice. În programare, matricea este organizată pe **rânduri** şi **coloane**. În *Figura 16* se prezintă organizarea conceptuală a matricei **m[3][2]** cu **3 rânduri** şi **2 coloane**. Reprezentarea în RAM a matricei este dată în *Figura 17*. Prin declaraţia **m[3][2], m** devine un poantor la un tablou ale cărui elemente sunt **m[0], m[1]** şi **m[2]**. La rândul lui, **m[0]** este un poantor la un tablou ale cărui elemente sunt **m[0][0]** şi **m[0][1]**. În general, un element **m[i][j]** se poate accesa prin formula: **m[i][j]= *(m[i]+j)**. Cunoscând că **m[i]** se descompune în ***(m+i)** se mai poate scrie **m[i][j]= *(*(m + i)+j)**. Concluzionând, toate expresiile care urmează vor referi valoarea aceluiaşi element $\mathbf{m}_{i,j}$ de tipul **TIP** din matricea $\mathbf{M}_{D1xD2}$:

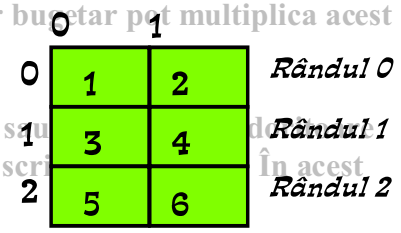


Figura 16

```
m[i][j]
*(m[i]+j)
*(*(m+i)+j)
*(m+i*D2*sizeof(TIP)+j*sizeof(TIP))
```


Dacă tabloul este tridimensional va avea 3 indici. Asociativitatea operatorului [ ] este de la stânga la dreapta. Pentru calcularea adresei la care este stocată valoarea selectată prin indici se evaluează **nume[indice1]**. Adresa care rezultă din adunarea adresei lui **nume** cu offsetul **indice1** formează o expresie poantor, apoi offsetul lui **indice2** se adună la expresia anterioară pentru a forma o nouă expresie poantor, continuându-se până la epuizarea indicilor. Operatorul de indirectare (*) se aplică numai după evaluarea ultimului indice. Fie declaraţiile:

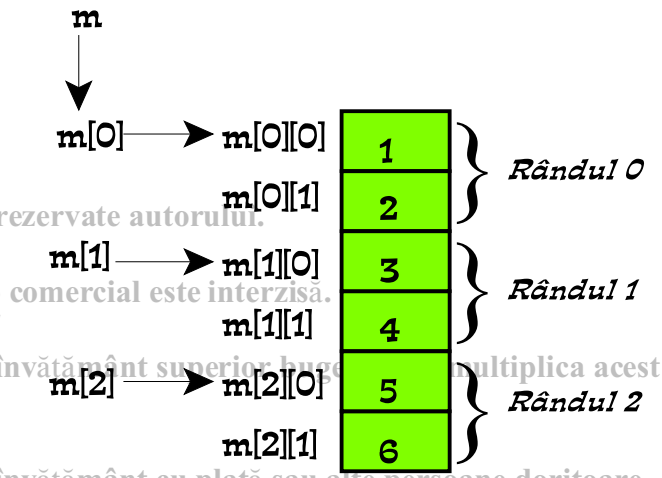


Figura 17


```
#define D1 3
#define D2 4
#define D3 5
int T3d[D1][D2][D3];
int i,*ptrT3d,(*ptrprfT3d)[D3];
```

În exemplul de mai sus, s-a declarat un tablou multidimensional **T3d** de 3 elemente, fiecare element fiind un tablou de **4 x 5 (D2 x D3)** valori de tipul **int**. O referinţă la un element al unui astfel de tablou apare în expresia **i=T3d[0][0][1];**

Generalizând cele discutate mai sus se poate spune că tablourile multidimensionale sunt stocate pe linii, deci ultimul indice variază cel mai repede. În consecinţă, expresia `T3d[0][0][2]` va referi următorul element de tablou din RAM, adică pe cel de al 3-lea.

Găsirea valorii lui `T3d[2][1][3]` se face astfel:

1. Primul indice `r` este multiplicat cu dimensiunea unui tablou `int` de `4 x 5 (D2 X D3)` elemente. Acest rezultat va poanta la cel de al `3`-lea tablou `4 x 5`.
2. Cel de al doilea indice `1` este multiplicat cu dimensiunea unui tablou `int` de `5 (D3)` elemente şi adunat la adresa reprezentată de `T3d[2]`.
3. Fiecare element al tabloului de `5` elemente este de tipul `int`, deci indicele final, `3`, este multiplicat cu lungimea lui `int` înainte de adunarea lui la `T3d[2][1]`.
4. Operatorul de indirectare este aplicat valorii poantor, rezultatul fiind elementul `int` de la acea adresă.

Formula de calcul finală pentru accesul la valoarea `T3d[i][j][k]`, este:

```
*(((T3d + i*D3*D2*sizeof(int)) + j*D2*sizeof(int)) + k*sizeof(int));
```

Câteva cazuri pentru care operatorul de indirectare nu mai este aplicat sunt:

```
ptrT3d=T3d[2][3];
ptrptrT3d=T3d[1];
```

În primul caz, expresia `T3d[2][3]` este o referinţă validă la tabloul tridimensional `T3d`, referindu-se un tablou unidimesional cu `D3` elemente. Pentru că valoarea poantorului adresează un tablou, operatorul de indirectare nu este aplicat. Similar, rezultatul `T3d[1]` este o valoare poantor la un tablou de `2` dimensiuni (`D2 x D3`).

9.8 Transferul tablourilor şi funcţiilor ca parametri de funcţie

Dacă parametrul unei funcţii este `tablou de tipul T`, declaraţia este astfel transformată încât se va citi "`poantor la tipul T`". Dacă un parametru este declarat ca `funcţie ce întoarce tipul T`, aceasta se transformă astfel încât se va citi "`poantor la funcţie ce întoarce tipul T`". Funcţia va putea întoarce o valoare de orice tip aritmetic, structură, reuniune, poantor sau `void` dar nu de tipul funcţie sau tipul tablou.

9.8.1 Transferul unui tablou unidimensional

Dacă avem un tablou unidimensional declarat astfel: `int a[5]`, el se transferă unei funcţii cu numele `media_aritmetica()`, scriind `media_aritmetica(a)`. Compilatorul va trata numele tabloului `a` conform celor discutate până acum, ca pe un poantor constant la primul element al tabloului. Deci tabloul nu se transferă prin valoare, ci funcţiei i se va transfera un poantor. Din acest motiv, din interiorul funcţiei se vor putea modifica valorile stocate în tabloul din funcţia apelantă. Dacă funcţia `media_aritmetica()` va avea doi parametri, tabloul `a` şi numărul de elemente ale tabloului, următoarele forme de declaraţie sunt corecte:

```
double media_aritmetica(int a[ ], int nr_elemente);
```

sau

```
double media_aritmetica(int *a, int nr_elemente);
```

Pentru că tot ceea ce primeşte funcţia este un poantor, ea nu are de unde să ştie numărul de elemente ale tabloului. Astfel, este necesar al doilea parametru, `nr_elemente`, care va trebui să fie numărul de elemente ale tabloului.


```
1  /* TAB6.C */
2  #include<stdio.h>
3
4  double media_aritmetica(int [ ], int);
5
6  int main(void)
7  {
8          int vector[5]={1,2,3,4,5};
9          double medar;
10
11         medar=media_aritmetica(vector,5);
12
13         printf("Media aritmetica este: %lg\n", medar);
14
15         return 0;
16 }
17
18 double media_aritmetica(int a[ ], int nr_elemente)
19 {
20         int i;
21
22         double total=0.0;
23
24         for(i=0; i<nr_elemente; i++)
25                 total+= a[i];
26
27         return total/nr_elemente;
28 }
```


Rezultate:

```
Media aritmetica este: 3
```


În programul prezentat `tabloul unidimensional` de `5 elemente` numit `vector` este argumentul funcţiei `media_aritmetica()` în `linia 11`. Parametrul `a` primeşte ca valoare adresa primului element din `vector`, adică pe `&vector[0]`. În interiorul funcţiei `a[i]` este modalitatea de acces la elementele tabloului, `a[0] = 1`, `a[1] = 2` etc. Al doilea parametru este numărul de elemente ale vectorului care asigură oprirea indicelui `i` la ultimul element valid de tablou. `a[0]` este primul element, iar ultimul este `a[5]` pentru că atunci când `i` va lua valoarea 6 se iese din ciclul `for`. Datorită apelului prin adresă, orice modificare a valorilor lui `a[i]` se va reflecta asupra lui `vector[i]`.

9.8.2 Transferul unui tablou unidimensional prin poantor

Funcţia `main()` ramâne identică cu cea anterioară.

```
1  /* TAB7.C */
2  #include<stdio.h>
3  double media_aritmetica(int *, int);
4
5  int main(void)
6  {
7          int a[5]={1,2,3,4,5};
8          double medar;
9
10         medar=media_aritmetica(a,5);
11
12         printf("Media aritmetica este: %lg\n", medar);
13
14         return 0;
15 }
16
17 double media_aritmetica(int *ptr, int nr_elemente)
18 {
19         int *ptr_capat=ptr + nr_elemente;
20
21         double total=0.0;
22
23         while (ptr < ptr_capat)
24                         total+= *ptr++;
25
26         return total/nr_elemente;
27 }
```

În `linia 10`, în urma apelului se copiază adresa elementului `0` în `ptr` şi valoarea numerică `5` în variabila `nr_elemente`. Iniţializarea din `linia 19` va atribui poantorului `ptr_capat` adresa `ptr+nr_elemente*sizeof(int)`. Pentru simplitate, presupunem că adresa lui `ptr` în zecimal este `1000`. Astfel, adresa stocată în `ptr_capat` este `ptr_capat = 1000+5*4=1020`. Această adresă este dincolo de domeniul alocat tabloului `a` însă conform celor spuse la comparaţia poantorilor, este corect să o folosim. Din acest motiv se scrie `ptr < ptr_capat` şi nu `ptr <= ptr_capat`. Instrucţiunea din `linia 24` face adunarea valorilor elementelor poantate de `ptr`. `total` este iniţializată cu valoarea `0`, iar valoarea poantată de poantor care este `1` se adună şi ea la `total`, care devine `1`. Apoi poantorul se incrementează şi adresa lui `prtr` devine `1000+1*4`, adică `1004`. Cum relaţia `1004 < 1024` este `adevărată`, ciclul continuă şi valoarea de `2` se adună la valoarea curentă a lui `total`, aceasta devenind `1+2 =3`. Poantorul este incrementat la adresa `1008`, etc. După ce s-au adunat cele `5` numere, poantorul va poanta la adresa `1020`. Aceasta este adresa stocată în `ptr_capat` motiv pentru care ciclul nu se reia. Valoarea întoarsă este media aritmetică, adică suma elementelor împărţită la numărul de elemente.

9.8.3 Operatorii * şi ++ la poantori

Operatorul `++` are o precedenţă mai mare decât operatorul `*`. Dacă `++` se realizează prima oară, de ce se adună prima oară valoarea şi numai apoi se incrementează poantorul? Pentru a înţelege rezultatul să revenim la modul de lucru al operatorului `++` postfix. Fie declaraţia `int a=3,b;`, dacă scriem `b=a++;` valoarea lui `a` este salvată într-un registru. `a` se incrementează devenind `4`, apoi valoarea din registru este atribuită lui `b`. Deci incrementarea se face înaintea atribuirii. Dacă `x` este o variabilă întreagă, iar `p` un poantor la un tip întreg, în cazul unei expresii de forma `x=*p++`, adresa din `p` se salvează într-un registru, apoi `p` se incrementează. Valoarea adresei salvate în registru este folosită cu operatorul `*` pentru extragerea valorii poantate de la adresa neincrementată.

`(*p)++`

În această formă conţinutul adresei din `p` se salvează într-un registru. Conţinutul locaţiei poantate de `p` se va incrementa, dar poantorul `p` va continua să poanteze la aceeaşi locaţie. Expresia va creşte cu `1` valoarea stocată în locaţia poantată de `p`, fără a modifica însă adresa stocată în poantorul `p`. O astfel de expresie, dacă poantează la un tablou, garantează incrementarea continuă a elementului de tablou poantat de poantor.

`*++p`

Aici operatorul `++` este prefixat, deci nu se foloseşte registrul. Valoarea stocată în `p` se incrementează, apoi se accesează valoarea stocată la această nouă adresă. O astfel de expresie garantează ratarea tratării primului element dintr-un tablou.

9.8.4 Transferul unui tablou bidimensional

Dacă tabloul este bidimensional, este obligatoriu ca declaraţia de parametru să conţină numărul de coloane al parametrului tablou. Numărul de linii este irelevant, din moment ce se va transfera un poantor la un tablou de rânduri, unde fiecare rând este un tablou cu `D2` elemente de tipul `T`. Ca exemplu, voi prezenta un program care face suma şi produsul elementelor nenule dintr-o matrice $A_{m \times n}$.

```
1  /* TAB8.C */
2  #include<stdio.h>
3  #define R 7
4  #define C 10
5
6  void CitesteMatrice(int [ ][C], int, int, char);
7  void ScrieMatrice(int [ ][C], int, int, char);
8  void SumProdMatrice(int [ ][C], int , int );
9  int CitesteNrElem(int *, int, char);
10
11 int main(void)
12 {
13       int a[R][C], n, m;
14
15       if (CitesteNrElem(&n,R,'r')) return 0;
16       if (CitesteNrElem(&m,C,'c')) return 0;
17       if (m<1) return(0);
18
19       CitesteMatrice(a,n,m,'a');
20       ScrieMatrice(a,n,m,'a');
21       SumProdMatrice(a,n,m);
22
23       printf("Suma: %i\nProdusul: %i\n", a[0][0], a[0][1]);
24
25       return 0;
26 }
27
28
29 void CitesteMatrice(int m[ ][C], int r, int c, char nume)
30 {
31       int i,j;
32       for(i=0;i<r;i++)
33             for(j=0;j<c;j++)
34             {
35                   printf("%c[%i][%i] = ",nume,i,j);
36                   scanf("%d", &m[i][j]);
```

```
}
}
void ScrieMatrice(int m[ ][C], int r, int c, char nume)
{
        int i,j;
        printf("Matricea %c este:\n", nume);
        for(i=0;i<r;i++)
        {
                for(j=0;j<c;j++)
                        printf("%i ", m[i][j]);
                printf("\n");
        }
}
int CitesteNrElem(int *x, int d, char nume)
{
        printf("%c= ", nume);
        scanf("%i", x);
        if (*x > d)
        {
                printf("\nEroare, numarul de elemente este prea mare\n");
                return 1;
        }

        return 0;
}

void SumProdMatrice(int m[ ][C], int r , int c)
{
        int i,j;
        int suma = 0, produs = 1;
        for(i=0;i<r;i++)
                for(j=0;j<c;j++)
                {
                        suma+=m[i][j];
                        if (m[i][j] != 0) produs*=m[i][j];
                }
        m[0][0]=suma;
        m[0][1]=produs;
}
```

Rezultate:

```
r= 2
c= 4
a[0][0] = 1
a[0][1] = 2
a[0][2] = 3
a[0][3] = 0
a[1][0] = 0
a[1][1] = 4
a[1][2] = 5
a[1][3] = 6
Matricea a este:
1 2 3 0
0 4 5 6
```

```
Suma: 21
Produsul: 720
```

Funcţiile care manipulează matricea sunt **CitesteMatrice()**, **ScrieMatrice()** şi **SumProdMatrice()**. Parametrul tablou este declarat sub forma: **int m[][C]**. El poate fi însă declarat şi sub una din formele: **int m[R][C]** sau **int (*m)[C]**. Pentru ultima formă, parantezele rotunde sunt obligatorii pentru că **[]** are precedenţă mai ridicată decât *. Dacă aş fi scris numai **int *m[C]**, **[]** dimensiunea **C** s-ar lega de numele **m**, deci **m[C]** va fi un tablou, apoi * s-ar lega de tipul **int**, adică elementele tabloului ar fi poantori la tipul **int**. Pentru înţelegerea detaliată a modului în care C interpretează declaraţiile complexe vă recomand citirea ANEXEI 2.

9.8.5 Care notaţie o folosim?

În programul care urmează, atât forma **a[i]** cât şi ***ptr** sunt folosite pentru accesul la elementele unui tablou unidimensional. Care dintre notaţii este mai bună?

```
1  /* TAB9.C */
2  #include<stdio.h>
3
4  int main(void)
5  {
6          float x[5]={1.1F,2.2F,3.3F,4.4F,5.5F};
7          float *ptr=x;
8
9          printf("x[2]= %g\n", x[2]);
10         printf("2[x]= %g\n", 2[x]);
11         printf("*(x+2)= %g\n\n", *(x+2));
12
13         printf("ptr[2]= %g\n", ptr[2]);
14         printf("*(ptr+2)= %g\n", *(ptr+2));
15
16         return 0;
17 }
```

Rezultate:

```
x[2]= 3.3
2[x]= 3.3
*(x+2)= 3.3

ptr[2]= 3.3
*(ptr+2)= 3.3
```

Înainte de a răspunde la întrebare, observaţi **linia 10** din programul **TAB9.C**. Forma de scriere este corectă, astfel că **x[2]** şi **2[x]** accesează valoarea aceluiaşi element de tablou. Asta pentru că operatorul **[]** se descompune într-o adunare, iar adunarea este comutativă. Răspunsul la întrebare este folosirea notaţiei cât mai uşor de înţeles în contextul problemei care se rezolvă.

9.9 Şiruri de caractere

În C **nu există un tip specific pentru şirul de caractere**. El **se declară ca un tablou unidimensional de caractere**. Se poate lucra şi cu un poantor la un tip **char**. Dacă şirul este declarat ca un tablou de caractere, spaţiul pentru stocarea elementelor de şir este

alocat automat. Dacă şirul se declară ca un poantor la caracter, nu se alocă spaţiu pentru elementele lui, utilizatorul fiind nevoit să-l aloce explicit cu **malloc()**, **calloc()** sau să-i atribuie adresa unui şir deja existent. Un caracter special este folosit pentru marcarea terminării şirului. Acest caracter se numeşte **null** şi se scrie ca '\0'. Nu trebuie să confundaţi acest **null** cu **NULL** discutat la poantori. Acel **NULL** marca o valoarea invalidă de poantor şi putea fi definită în orice fel. Pentru caracterul **null** este garantată valoarea **0**. Scrierea de '\0' este preferabilă unui simplu **0** deoarece compilatorul, conform celor discutate la constante, atribuie tipul **int** lui **0** şi **char** lui '\0'. Diferenţa dintre tipuri este spaţiul folosit pentru stocare, **int** se stochează pe 2 sau 4 octeţi, iar **char** pe unul singur. Pentru stocarea unui zero este suficient un singur octet.


```
1  /* TAB10.C */
2  #include <stdio.h>
3
4  int main(void)
5  {
6    char nume[5]; /* Declaratia variabilei sir de caractere */
7
8    nume[0] = 'T';
9    nume[1] = 'i';
10   nume[2] = 'b';
11   nume[3] = 'i';
12   nume[4] = '\0';  /* Caracterul null - marcheaza terminarea textului */
13
14   printf("Numele este: %s\n", nume);
15   printf("Al 3-lea caracter este: %c\n", nume[2]);
16   printf("O parte din nume este: %s\n", &nume[1]);
17
18   return 0;
19 }
```


Rezultate:

```
Numele este: Tibi
Al 3-lea caracter este: b
O parte din nume este: ibi
```


Iniţializarea şirurilor

În **linia 6**, **nume** este un **tablou unidimensional** cu **5 elemente** de tipul **char**. Putem spune că avem 5 variabile de tipul **char** cu numele **nume[0]**, **nume[1]**, **nume[2]**, **nume[3]**, **nume[4]**. Variabila **nume** este un şir de caractere ce poate stoca cel mult 5 caractere, dar pentru că este nevoie şi de caracterul **null**, ce marchează terminarea şirului, mai rămân numai 4 caractere utilizabile. Pentru a stoca date în şir se folosesc 5 instrucţiuni de atribuire, fiecare atribuind un caracter alfabetic la câte unul dintre caracterele şirului. În final, ultimul caracter din şir este valoarea numerică 0 care indică terminarea şirului. După iniţializarea şirului de caractere acesta va fi afişat pe ecran în **linia 14**.

Caracterul de conversie **%s** din **printf()** se foloseşte la afişarea şirurilor de caractere. Afişarea începe cu primul caracter al şirului şi se termină la întâlnirea caracterului **null**. Observaţi că în **printf()** doar numele şirului trebuie specificat, fără nici un indice, pentru că suntem interesaţi ca afişarea să se facă începând cu primul caracter. Pentru că toate cele discutate la tablouri rămân valabile şi aici, **nume** se referă la întregul şir, iar **nume[i]** la un singur caracter din şir.

Din exemplul anterior aţi putea deduce că utilizarea şirurilor este greoaie din moment ce trebuie iniţializat individual fiecare element de şir. Realitatea este însă mai simplă, iar

exemplul a avut numai rolul de a prezenta un mod de tratare al elementelor de şir identic cu metodele generale de iniţializare prezentate la tablouri. Următoarele **metode de iniţializare** pot fi practicate în cazul unui şir de caractere:

```
1  char nume[7] = {'V','a','s','i','l','e','\0'};
2  char prenume[9]="Gheorghe";
3  char titlu[ ]="doctor inginer";
```

În **linia 1**, folosind o metodă de iniţializare specifică tuturor tablourilor, compilatorul nu este suficient de deştept ca să se prindă că este vorba de un şir, motiv pentru care trebuie să punem cu mâna terminatorul de şir. Lăsarea lui afară ar fi dus la un tablou corect iniţializat, dar la un şir de caractere neterminat. În **linia 2**, variabila **prenume** este iniţializată cu un şir de caractere. Conform celor prezentate în paragraful numit "Elemente de C" un şir este o secvenţă de caractere cuprinsă între ghilimele. Aici, compilatorul "se prinde" că se lucrează cu un şir şi adaugă automat la sfârşitul lui caracterul **null**. **Linia 3** prezintă cazul în care numărul de elemente ale şirului se deduce de către compilator, el fiind numărul de caractere din şir plus unu, pentru terminatorul de şir care din nou se adaugă automat.

Afişarea unor porţiuni ale şirului

În **linia 15**, **printf()** ilustrează modul de afişare al unui singur caracter din şir utilizând caracterul de conversie **%c** şi specificând caracterul particular din variabila **nume** pe care dorim s-o afişăm. Ultimul **printf()** ilustrează modul în care se poate afişa o porţiune din şir prin specificarea punctului de început şi utilizând indicele. După cum ştiţi deja, operatorul **&** întoarce adresa lui **nume[1]**.

9.9.1 Atribuirea în cazul de şiruri

O eroare frecventă este scrierea unei atribuiri de forma **nume="Ion";**. Aici, **nume** stochează adresa primului caracter din şir, adică adresa lui **'I'**. Această adresă este constantă, modul de lucru corect este folosirea funcţiei **strcpy()** din biblioteca standard.

```
1  /* TAB11.C */
2  #include <stdio.h>
3  #include <string.h>
4
5  int main(void)
6  {
7    char nume1[12], nume2[12], maxim[25];
8    char titlu[ ]="Acesta este un titlu.";
9
10   strcpy(nume1, "Ioana");  /* atribuirea unui sir la o variabila */
11   strcpy(nume2, "Tibi");
12   printf("    %s\n\n", titlu);
13   printf("Nume 1 este: %s\n", nume1);
14   printf("Nume 2 este: %s\n", nume2);
15
16   if(strcmp(nume1, nume2) > 0)  /* comparatia a 2 siruri */
17     strcpy(maxim, nume1);
18   else
19     strcpy(maxim, nume2);
20
21   printf("Cel mai mare nume in sens alfabetic este: %s\n", maxim);
22
23 }
```

```
24    strcpy(maxim, nume1);
25    strcat(maxim, " "); /* concatenarea a doua siruri */
26    strcat(maxim, nume2);
27    printf("Cele doua nume sunt: %s\n", maxim);
28
29    return 0;
30 }
```

Rezultate:

```
     Acesta este un titlu.
Nume 1 este: Ioana
Nume 2 este: Tibi
Cel mai mare nume in sens alfabetic este: Tibi
Cele doua nume sunt: Ioana Tibi
```

Funcţia de bibliotecă **strcpy()** este declarată în fişierul antet **string.h** (vezi **linia 3** din programul anterior) iar descrierea ei este dată în continuare:

strcpy - copiază un şir de caractere

Prototip: **char *strcpy(char *şirDestinaţie, const char *şirSursă);**

Fişier prototip: **<string.h>**

Valoare întoarsă: întoarce şirul destinaţie. Nu se poate afla din valoarea întoarsă dacă s-a realizat copierea fără eroare.

Parametri: **şirDestinaţie** este şirul în care se face copierea;
şirSursă este şirul din care se face copierea; el trebuie să fie terminat în caracterul **null**.

Remarcă: Funcţia **strcpy()** copiază caracter cu caracter, inclusiv caracterul terminator de şir, din **şirSursă** în **şirDestinaţie**. Oprirea copierii se face după întâlnirea caracterului **null**. Lipsa lui duce la un comportament nedefinit în funcţionare, prezenţa lui **null** fiind singura metodă de verificare a atingerii limitei lui **şirSursă**. **şirDestinaţie** trebuie să fie suficient de mare pentru a stoca tot conţinutul lui **şirSursă**. Altfel, foarte probabil, din cauza că zona de RAM destinată lui **şirDestinaţie** este depăşită, programul va "crăpa".

9.9.2 Sortarea alfabetică a şirurilor

Funcţia care permite compararea a două şiruri este **strcmp()** (**"string compare"** - vezi **linia 17**).

strcmp - compară două şiruri de caractere

Prototip: **int strcmp(const char *şir1, const char *şir2);**

Fişier prototip: **<string.h>**

Valoare întoarsă: funcţia compară pe **şir1** cu **şir2** şi întoarce un rezultat în funcţie de relaţia dintre cele două şiruri, astfel:

Valoare întoarsă	Relaţia între şir1 şi şir2
< 0	**şir1 < şir2**
0	**şir1 == şir2**

> 0	**şir1 > şir2**

Parametri: **şir1** şi **şir2** sunt şiruri terminate în **null** care vor fi comparate.

Din descriere se observă că **strcmp()** va întoarce o valoare **>0** dacă primul şir este mai mare decât al doilea, **zero** dacă au aceeaşi lungime şi aceleaşi caractere şi o valoare **<0** dacă primul şir este mai mic decât al doilea. Unul dintre şiruri, în funcţie de rezultat, este copiat în variabila şir **maxim** prin **linia 18** sau **20**, apoi cel mai mare nume, în sensul alfabetic, se afişează în **linia 22**. Nu trebuie să fiţi surprinşi că **Tibi** va fi mai mare pentru că lungimea nu contează, fiind semnificativă numai poziţia în alfabet. Trebuie să ştiţi că rezultatul mai depinde şi de modul în care au fost scrise numele, adică cu litere mari sau mici. Există funcţii care permit transformarea unui şir oarecare într-unul numai cu majuscule sau numai cu litere mici.

9.9.3 Concatenarea şirurilor

Liniile 24-27 ilustrează o altă funcţie standard, **strcat()** (“**string concatenation**”). Această funcţie adaugă caracterele din al doilea şir la capătul primului şir, având grijă să facă ajustarea caracterului **null**. Astfel, **nume1** se copiază în **maxim**, apoi două spaţii se concatenează la **maxim** şi în final **nume2** se concatenează la combinaţia anterioară. Rezultatul afişează numele celor două şiruri stocate în variabila şir **maxim.**

Şirurile sunt utile şi simplu de utilizat, dar necesită unele precauţii. Este o eroare să copiaţi un şir în altul care este declarat mai scurt decât sursa, deoarece compilatorul va realiza copierea şi va suprascrie o porţiune din program sau din date. Pentru că nu există nici o metodă de a avertiza programatorul despre acest dezastru, e bine să fiţi cu băgare de seamă.

Pentru a găsi toate funcţiile de lucru cu şiruri, consultaţi documentaţia compilatorului (sau sistemul lui de ajutor). Există funcţii care copiază cel mult un număr fixat de caractere, altele adaugă caractere în faţa, la mijlocul sau la capătul unui şir. Desigur, pot şi scoate caractere de oriunde din şir. Merită citită această documentaţie pentru că, în unele cazuri, funcţiile peste care daţi vă vor simplifica mult treaba.

9.10 Poantori la şiruri

Durata de existenţă a şirului, tehnic denumit **constantă şir**, este **static**. Compilatorul va stoca în segmentul de date toate şirurile care nu iniţializează un tablou de caractere. De exemplu, la o iniţializare de forma **char sir[ ]="Salut!";** caracterele din şirul **"Salut!"** şi variabila **sir** ar putea fi stocate pe stivă. Pentru exemplul care urmează variabila poantor la caracter **ptrSir** este stocată pe stivă, dar şirul **"Sir de caractere cu poantor"** este întotdeauna stocat în segmentul de date.

```
1  /* TAB12.C */
2  #include <stdio.h>
3
4  int main(void)
5  {
6          char *ptrSir;
7
8          ptrSir ="Sir de caractere cu poantor";
9          puts(ptrSir);
```

```
10
11      return 0;
12  }
```

Rezultate:

```
Sir de caractere cu poantor
```


Funcţia `puts()` primeşte ca parametru de intrare adresa primului caracter din şir. Ea verifică dacă apare caracterul `null`; dacă da, îşi întrerupe execuţia. Altfel, afişează caracterul curent pe ecran, adună `1` la adresa lui, verifică dacă caracterul curent este `null` etc. Observaţi că operaţiile de prelucrare a şirurilor sunt dependente de caracterul `null`. Acesta este motivul pentru care şirurile sunt cunoscute şi sub numele de şiruri cu terminator `null`, iar în unele documentaţii mai apar şi sub numele de şiruri ASCII-Z. Această metodă de stocare permite depăşirea limitelor de stocare fixe. Practic, şirul poate fi oricât de lung dacă încape în RAM. În loc de `puts()` se poate folosi `printf()` pentru afişarea şirului scriind `printf("%s",ptrSir);`. Aici `"%s"` e un şir constant stocat în segmentul de date.

ANTAL Tiberiu Alexandru
tel.: 0040-264-530918
e-mail: antaltiberiu@pcnet.ro

Caracterele unui şir constant se stochează în ordinea lor de apariţie în locaţii de memorie consecutive, iar caracterul '`\0`' este adăugat automat la sfârşitul lui.

Optimizări la stocarea şirurilor

Conform standardelor ANSI şi ISO compilatoarele pot să facă optimizări la stocarea şirurilor. Compilatorul poate să nu stocheze două şiruri, dintre care unul este un subşir al celuilalt, la adresele de memorie diferite.

9.10.1 Consideraţii despre creşterea vitezei programelor

Fie programul:

```
1   /* SIRVIT1.C */
2   #include<conio.h>
3   #include<string.h>
4
5   int main(void)
6   {
7           char sirtablou[ ]="Salut bade!\n";
8           int i,lungime;
9
10          lungime=strlen(sirtablou);
11          for (i=0; i < lungime; i++)
12                  putch(sirtablou[i]);
13
14          return 0;
15  }
```


ANTAL Tiberiu Alexandru
tel.: 0040-264-530918
e-mail: antaltiberiu@pcnet.ro

Rezultat:

```
Salut bade!
```

El va afişa şirul `"Salut bade!\n"`, caracter cu caracter, prin folosirea funcţiei `putch()`. Înainte de fiecare incrementare a lui `i` se va afişa caracterul `sirtablou[i]`. La fiecare referire a unui element de tablou, locaţia de memorie la care se află elementul se calculează după formula `&sirtablou[0] + i *sizeof(char)`. Altfel spus, C începe să calculeze

adresa de bază a lui **sirtablou** sau **&sirtabolu[0]**, apoi determină offset-ul **i*sizeof(char)** şi adună cele două valori pentru a obţine adresa elementului. În acest caz **sizeof(char)** dă **1** şi înmulţirea este simplu de efectuat, dar dacă tabloul ar fi avut elemente de tipul **float** sau **double** lucrurile s-ar fi complicat. La fiecare referire de element de tablou, este necesară realizarea unei înmulţiri, operaţie care este costisitioare ca timp de execuţie. Să aducem programul de mai sus la forma:

```
1  /* SIRVIT2.C */
2  #include<conio.h>
3  #include<string.h>
4
5  int main(void)
6  {
7          char sirtablou[ ]="Salut bade!\n";
8          char *psir=sirtablou;
9          int i,lungime;
10
11         lungime=strlen(psir);
12
13         for (i=0; i < lungime; i++)
14                 putch(*psir++);
15
16         return 0;
17 }
```

În **linia 8** s-a adăugat un poantor **psir** care este iniţializat în linia de declaraţie. Rezultatul va fi identic cu cel al programului anterior, dar la fiecare reluare a ciclului se incrementează **psir** şi se afişează ***psir**. Altfel spus, deplasarea în şir se face prin incrementarea unui poantor. Când compilatorul întâlneşte expresia **psir++**, o aduce la forma **psir+sizeof(char)**. O altă îmbunătăţire este eliminarea determinării adresei elementului de bază. Cum înmulţirile sunt operaţii încete, prin această metodă am economisit timp. Incrementarea sau decrementarea unui poantor, pentru parcurgerea unui tablou, este întotdeauna mai rapidă decât generarea completă a adresei unui element, avantaj de care putem beneficia numai dacă folosim notaţia de poantor în locul celei de tablou.

9.10.2 Tablouri de poantori la şiruri

Ştiţi deja că în C ne putem referi la un element de tablou în mai multe moduri. Fie declaraţiile:

```
1  char tsir[5][11];
2  char *ptsir[5];
```

În **linia 1**, **tsir** este un tablou bidimensional cu elemente de tipul **char**, iar în **linia 2**, ***ptsir** este un tablou de poantori la **char**. Sintactic, referirile de forma **tsir[i][j]** şi **ptsir[i][j]** sunt corecte şi echivalente. **tsir** este însă un tablou pentru care se alocă **5x11** locaţii de tipul **char**, iar formula de calcul a poziţiei unui element **tsir[r][c]** este **c+r*11**. În cazul lui **ptsir** se alocă spaţiu numai pentru un tablou de **5** poantori. Aceştia nu sunt iniţializaţi să poanteze undeva, motiv pentru care utilizatorul trebuie să-i iniţializeze explicit în momentul declaraţiei sau prin instrucţiuni ale programului. Presupunând că fiecare poantor poantează la un şir de **11** caractere, se vor aloca **5x11** locaţii pentru caractere şi **5** locaţii pentru poantori. Deci, din punctul de vedere al spaţiului de memorie alocat, cea de a doua metodă este mai costisitoare. Atunci de ce să o folosim? Avantajul celei de a doua metode

este că şirurile pot avea lungimi diferite. Fie declaraţia:

```
char *mesaj_err[5]={"an gresit","luna gresita", "zi in [0,31]", "eroare necunoscuta",\
"eroare fatala"};
```

S-a iniţializat un tablou **mesaj_err** de **5** elemente de tipul poantor la **char**. Observaţi că nu este necesar ca şirurile la care poantează elementele tabloului **mesaj_err** să aibă aceeaşi lungime. În *Figura 19* se prezintă spaţiul de date pentru declaraţia de mai sus.

Dacă declaraţia ar fi fost de forma:


```
char tmesaje[5][19]={"an gresit","luna gresita", "zi in [0,31]", "eroare necunoscuta",\
"eroare fatala"};
```


spaţiul de date era cel din *Figura 18*. Şirul cu lungimea cea mai mare este "**eroare necunoscuta**" şi are **18** caractere plus terminatorul de şir. Înseamnă în total **19** caractere pentru stocarea lui. **tmesaje** este un **tablou** de **5 elemente** de tipul **tablou** de **19 elemente** de tipul **char**. Elementele tabloului de **5** elemente au toate acelaşi tip, şi anume tablou de **19** elemente de tipul **char**. Indiferent dacă spaţiul de **19** caractere este ocupat sau nu, se alocă **19** caractere pentru un element **tmesaje[i]**.

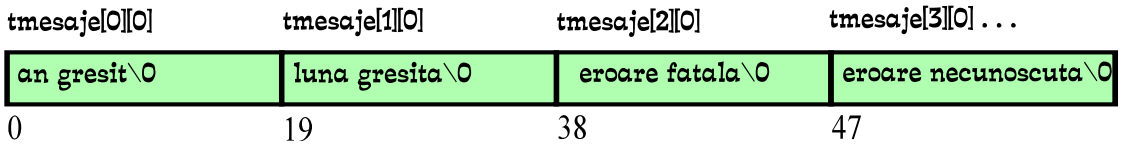


Figura 18

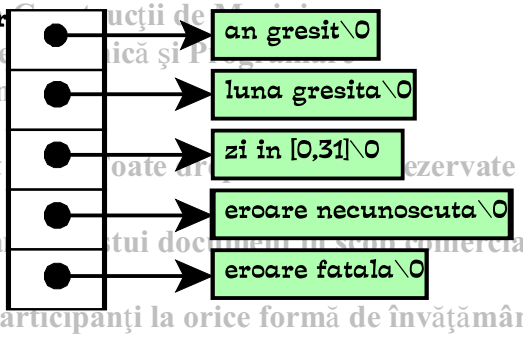


Figura 19

Universitatea Tenhică din Cluj-Napoca
Facultatea Construcţii de Maşini
Catedra de Mecanică şi Programare
Curs de limbaj C

Conf. dr. ing. ANTAL Tiberiu Alexandru
Universitatea Tenhică din Cluj-Napoca
Facultatea Construcţii de Maşini
Catedra de Mecanică şi Programare
Curs de limbaj C
tel.: 0040-264-530911
e-mail: antaltiberiu@pcnet.ro

10 Structuri

Structura, numită înregistrare (**record**) în limbajele Ada şi Pascal sau tuplă (**tuple**) în programarea funcţională, este tipul de dată cel mai general al limbajului C. Structura este formată din membri, fiecare membru putând avea un tip diferit de ceilalţi membri. Pentru că tipul unui membru este arbitrar este posibil ca acesta să fie el însuşi un tip structurat, ca de exemplu tablou sau structură. Tipul de dată structură permite gruparea unor tipuri de date, distincte sau nu, sub un singur nume.

Modificări aduse de standardizare

În ANSI C sunt definite noţiunile de atribuire a structurilor, transferul de parametri de tipul structură funcţiilor şi întoarcerea unei valori de tipul structură de către funcţie. Majoritatea compilatoarelor asigurau aceste facilităţi, fără însă ca mecanismul de lucru să fie unitar şi definit precis.

O declaraţie de structură specifică un nou tip de dată şi o secvenţă de variabile, numite membri sau câmpuri ale structurii, care pot avea tipuri distincte. Formele declaraţiei de structură sunt:

```
struct [tip] { lista-membri } [identificatori];
```

sau

```
struct tip identificatori;
```

Un identificator opţional **tip** dă nume tipului structurii şi va putea fi utilizat ulterior la referirea tipului structură. În C este necesară utilizarea explicită a cuvântului rezervat **struct** pentru a declara o variabilă de tipul structură. **identificatori** sunt variabile opţionale de tipul **struct** care ţin sub numele lor întreaga secvenţă a membrilor din **lista-membri** definiţi de acel tip structură. Declaraţia unui tip structură este numai un şablon pentru declaraţii ulterioare ale unor variabile de tipul structură. Numele din **lista-membri** nu pot să apară de două ori în aceeaşi structură.

Declararea tipurilor de structuri

În continuare prezint modalitatea declarării unor tipuri de structuri. Cuvântul **struct** va fi urmat de numele **tip**. Acest nume de tip permite identificarea şablonului, fără a realiza alocarea de memorie la crearea tipului structură.

```
1  struct Data
2  {
3          int zi;
4          int luna;
5          int an;
```

```
};

struct Membru_familie
{
        char numepersoana[50];
        int varsta;
};
struct Suma_platita
{
        struct Data data_platii;
        long suma;
        char motiv[255];
};

struct Cheltuieli_familie
{
        struct Membru_familie cine;
        struct Suma_platita cit[2];
        struct Cheltuieli_familie *ptrCheltuieli;
};
```

Tipul structură cu numele de **Data** este format din trei întregi. Avantajul faţă de folosirea unui tablou este că membrii au nume sugestive şi dacă este cazul pot avea tipuri distincte. În cazul unui tablou am fi avut trei elemente, numerotate cu 0, 1 şi 2, toate de tipul **int**. Numărul de ordine al elementului nu putea sugera semnificaţia lui, trebuia ca noi să reţinem că în elementul 0 am stocat ziua, în elementul 1 luna şi în elementul 2 anul.

Tipul de dată structură cu numele **Membru_familie** este format din doi membri, un tablou de caractere şi un întreg. Tipul structură **Suma_platita** este format din trei membri dintre care unul este de tipul structură **Data**. Şi în final, tipul structură **Cheltuieli_familie**, are trei membri, primul de tip structură, al doilea tablou cu elemente de tipul structură, iar ultimul poantor la un tip structură.

Declararea unor variabile de tip structură

Tipul structurii informează compilatorul despre modul în care se va construi variabila de tipul respectiv. Odată ce avem declarat un tip structură se pot declara variabile de tipul structură. Mai jos, tipul **Data** este declarat în **liniile 1-6**. Numele **zinastere** şi **zisalar** sunt nume de variabile. Între acolada de închidere şi caracterul ; se pot specifica oricâte nume de variabile, separate prin virgulă, care vor deveni variabile de tipul structură respectiv. Dacă dorim să declarăm şi alte variabile de acelaşi tip, însă după caracterul ;, deşi **Data** este un tip de dată asemenea, de exemplu, lui **int**, **struct** trebuie să preceadă numele tipului structură, adică nu se poate scrie **Data** ci este obligatorie scrierea sub forma **struct Data**.

```
1  struct Data
2  {
3          int zi;
4          int luna;
5          int an;
6  } zinastere, zisalar;
7
8  struct Data zioarecare;
9
10 struct Data saptamana[7];
```

saptamana este un **tablou** de **7 elemente**, fiecare element fiind o structură de tipul **Data**. Fiecare element are deci un an, o lună şi o zi distinctă. Ne referim la elementul **0** prin **saptamana[0]**, iar prin **saptamana[0].zi** la membrul **zi** al elementului **0**. Pentru a accesa **luna** elementului **4** scriem **saptamana[4].luna** etc.

Iniţializarea variabilelor de tip structură

În capitolul despre tablouri am văzut modul în care se foloseau acoladele pentru iniţializarea tablourilor cu una sau mai multe dimensiuni. Iniţializarea structurilor se face după o sintaxă similară. De exemplu, dacă dorim să iniţializăm variabila **zioarecare** în locul **liniei 8** scriem:


```
struct Data zioarecare ={19,12,1976};
```


prima valoare - **19** - i se atribuie primului membru - **zi**, a doua valoare - **12** - celui de al doilea membru - **luna** şi a treia valoare - **1976** - membrului **an**. Dacă se declară o variabilă de tipul structură cu numele **babe**:

ANTAL Tiberiu Alexandru
tel: 0040-264-530918
e-mail: antaltiberiu@pcnet.ro

```
struct Membru familie babe;
```

iniţializarea membrilor ei se face astfel:

```
strcpy(babe.numepersoana,"Smaranda Quiz");
babe.varsta=24;
```

Pentru că este posibil ca o structură să declare membri de tipul structură, voi prezenta un exemplu de iniţializare şi pentru acest caz. Să presupunem că se declară o variabilă **x** de tipul structură **Suma_platita**. În cazul tablourilor cu mai multe dimensiuni, pentru fiecare dimensiune foloseam o pereche de acolade. Pentru structuri cu membri structură, iniţializarea unui membru se face similar, fiind necesară pentru fiecare astfel de membru câte o pereche de acolade.


```
struct Suma_platita x={
        {12,7,1976},
        12345000,
        "Nimicuri"
};
```


10.1 Operatorul de selecţie a unui membru de structură

Formă:	**expresie.membru**

O expresie urmată de **operatorul de selecţie a membrului "."** şi apoi de numele membrului face selecţia membrului din structură sau reuniune (reuniunea sau **union** va fi prezentată în capitolul următor). Primul operand, **expresie**, trebuie să fie o structură sau o reuniune, al doilea trebuie să identifice un **membru**. Rezultatul este valoarea membrului selectat şi este o **l-value** dacă membrul este o **l-value**, adică poate să apară în stânga operatorului de atribuire.

```
/* STRUCT1.C */
#include<stdio.h>
#include<string.h>
struct Data
{
        char zi;
        char luna;
        int an;
};

struct Membru_familie
{
        char nume_persoana[50];
        int varsta;
        char functie[25];
};

struct SumaPlatita
{
        struct Data data_platii;
        long suma;
        char motiv[255];
};

struct Cheltuieli
{
        struct Membru_familie cine;
        struct SumaPlatita cit[2];
        struct Cheltuieli *ptrCheltuieli;
};

int main(void)
{
        struct Cheltuieli a = {

                {"Sile aMicu",47,"cap de familie"},
                {{{1,2,1998},100000L,"chef"}, {{3,4,1999},2000000L,"nunta"}},
                NULL
        };
        int i;

        printf("%s, de varsta %i, %s, a cheltuit asa:\n", a.cine.nume_persoana,\
a.cine.varsta, a.cine.functie);
        for(i=0;i<2;i++)
                printf("- zi = %i, luna = %i, an = %i, lei = %li, pentru %s\n",\
a.cit[i].data_platii.zi, a.cit[i].data_platii.luna, a.cit[i].data_platii.an, a.cit[i].suma, \
a.cit[i].motiv);

        return 0;
}
```


Rezultate:

```
Sile aMicu, de varsta 47, cap de familie, a cheltuit asa:
- zi = 1, luna = 2, an = 1998, lei = 100000, pentru chef
- zi = 3, luna = 4, an = 1999, lei = 2000000, pentru nunta
```

Accesul la un membru tablou

Dacă membrul de accesat este tablou cu o dimensiune, de exemplu **cit**, având elementele de tipul structură **SumaPlatita**, operatorul [] trebuie folosit pentru accesul la elementele tabloului după, cum se observă în **liniile 48** şi **49**.

Accesul la un membru al unui membru structură

Când un membru este la rândul lui o structură, de exemplu membrul **cine** este o structură de tipul **Membru_familie**, trebuie folosit de două ori operatorul "." pentru a accesa un membru din interiorul membrului structură (vezi **liniile 44** şi **45**).

10.2 Comparație între tablou și structură

Atât tabloul cât şi structura sunt tipuri de date structurate. Tipurile structurate se definesc în termenii unor tipuri deja existente. Există mai multe metode de combinare a tipurilor constituente pentru a forma o structură de date complexă, acestea fiind numite **metode de structurare**. Metodele de structurare diferă prin strategia de construire a tipului structurat şi modalităţile de acces la componentele acesteia. Variabila tablou grupează componente de acelaşi tip, numite elemente, sub un singur nume, numele tabloului. Metoda de selectare a unei componente este indexarea, numele tabloului primind ca argument indexul pentru a selecta o componentă particulară. Variabila structură grupează sub acelaşi nume componente de tipuri diferite, numite membri, metoda de acces la acestea fiind selectorul de membru. Putem spune că ambele tipuri structurate permit gruparea de variabile sub un singur nume. În ciuda acestei asemănări, compilatorul tratează cele două tipuri structurate diferit. După cum ştiţi deja, în C numele unui tablou este adresa primului element de tablou. Pentru o variabilă de tipul structură, numele variabilei structură este asemenea unui nume de variabilă simplă (nu este adresa unui membru al structurii).

Când un tablou se transferă ca parametru al unei funcţii nu se poate alege modul de transfer pentru că numele tabloului este automat transformat într-un poantor constant la primul element de tablou. Nu există nici o metodă prin care să forţăm transferul unui parametru tablou prin valoare. În cazul structurii, decizia este la dispoziţia programatorului, aceasta putându-se transfera atât prin valoare cât şi prin poantor.

O funcţie nu poate întoarce un tablou. Este însă posibil să întoarcem un poantor la un element de tablou, iar dacă acesta este chiar primul element putem spune că întregul tablou este întors. Argumentul de mai sus este însă prea slab în comparaţie cu structura, care se poate întoarce atât prin valoare cât şi prin poantor.

Cele spuse mai sus pot fi sintetizate în tabelul următor:

Tip obiect	Numele obiectului este	Transfer în caz de parametru de funcţie	Valoare întoarsă de funcţie
tablou	poantor constant la primul element	poantor	-

`structură`	structura în sine	valoare sau poantor	prin valoare sau prin poantor

10.3 Atribuirea structurilor

O variabilă de tipul structură poate fi atribuită unei alte variabile de acelaşi tip structură prin intermediul lui "=". Valorile tuturor membrilor variabilei structură din dreapta lui = vor fi copiate în membrii variabilei structură din stânga lui = (inclusiv membri tablou sau structură). De exemplu, folosind tipul structură `Data` declarat mai sus şi două variabile `zi1` şi `zi2`, dintre care prima este iniţializată,

```
struct Data zi1 {28,7,1967}, zi2;
```

este corect să scriem:

```
zi2=zi1
```

Indiferent de cât de complicată va fi structura, toţi membrii ei se vor copia. În cazul tablourilor, o astfel de operaţie este imposibilă pentru că numele tabloului este un poantor constant la primul element de tablou. În cazul unei declaraţii de forma:

```
char a[80], b[80];
```

o atribuire de genul "`b=a`" ar încerca să modifice adresa poantorului constant `b` la cea a lui `a`, lucru imposibil pentru că poantorul fiind constant, el va poanta tot timpul execuţiei programului aceeaşi adresă. Compilatorul va sancţiona o astfel de atribuire printr-un mesaj de eroare. De asemenea, este imposibilă comparaţia între două variabile de tipul structură, dar comparaţia se poate face membru cu membru, dacă membrii nu sunt tablouri sau structuri.

10.4 Transferul variabilelor de tipul structură ca parametri de funcţii

Dacă în cazul tabloului nu era de ales, la structuri o variabilă se poate transfera prin valoare sau prin poantor. Problema care se pune este de alegere a celei mai avantajoase metode de transfer. Toate operaţiile de transfer se derulează prin stivă, în sensul că obiectele de transferat se copiază pe aceasta. Dacă obiectul este mare copierea lui va dura mult şi programul va rula încet. În plus, apare posibilitatea depăşirii dimensiunii maxime a stivei. Transferul unui poantor va ocupa cel mult 4 octeţi. Transferul unei valori de tipul structură va fi dependent de tipul membrilor structurii. Alegerea între copierea a 4 octeţi sau a unui număr mai mare este simplă. Putem spune că este mai avantajoasă folosirea transferului prin poantor, aceasta asigurând şi independenţa de spaţiul necesar pentru stocarea variabilei structură. Considerând tipul structurat `Data`, cele două metode de transfer posibile sunt ilustrate în următoarele prototipuri:


```
void f_valoare(struct Data zi);
void f_poantor(struct Data *zi);
```

iar apelurile celor două funcţii, considerând o variabilă cu numele `zi` de tipul structurat

`Data` sunt:

```
f_valoare(zi);
f_poantor(&zi);
```

În primul caz, variabila `zi` fiind formată din `3` tipuri `int` care se stochează pe `4` octeţi pe stivă se vor copia `12` octeţi. În cel de al doilea caz, poantorul va ocupa pe stivă numai `4` octeţi.

10.5 Poantori la structuri

Transferul prin poantori este mai eficient decât cel prin valoare dar codul din funcţii devine mai complicat de scris.


```
1  /* STRUCT2.C */
2  #include<stdio.h>
3  struct Data
4  {
5          char zi;
6          char luna;
7          int an;
8  };
9
10 void citeste_data(struct Data *);
11
12 int main(void)
13 {
14         struct Data o_zi;
15         citeste_data(&o_zi);
16         printf("Data este: %i, %i, %i\n",o_zi.zi, o_zi.luna, o_zi.an);
17 }
18
19 void citeste_data(struct Data *pzi)
20 {
21         printf("zi = ");
22         scanf("%i",&(*pzi).zi);
23         printf("luna = ");
24         scanf("%i",&(*pzi).luna);
25         printf("an = ");
26         scanf("%i",&(*pzi).an);
27 }
```


Rezultate:

```
zi = 28
luna = 12
an = 1978
Data este: 28, 12, 1978
```


În cazul transferului unui poantor `p` la o structură, un membru numit `membru` se va accesa prin:

```
(*p).membru
```

Observaţi această formă de scriere în `liniile 22, 24, 26`. Întrebarea care apare este legată de necesitatea folosirii parantezelor rotunde.

10.5.1 De ce (*p).membru?

De ce este necesar să scriem `(*p).membru` în loc de `*p.membru`? Cei doi operatori, "*" şi "." au precedenţe diferite. Operatorul de selecţie a unui membru de structură, ".", are precedenţa cea mai mare. Astfel, dacă se scrie `*p.membru`, avem implicit `*(p.membru)`. Pentru ca la compilare totul să meargă bine, `p` ar trebui să fie o structură. Dar `p` nu este o structură, ci un poantor la structură. Dacă totuşi `p` ar fi fost o structură, `membru` ar fi fost accesat, însă operatorul "*" ar fi aflat valoarea la care poantează `p.membru`. Adică ar fi fost departe de a accesa ceea ce credeam noi - valoarea lui `membru` poantată de `p`.

Deoarece au folosit mult poantorii la structuri, Kernighan şi Ritchie au inventat un operator nou care să fie mai simplu de folosit. El era format din două caractere separate, "-" şi ">" care combinate au dat "->". Era ceva similar cu "/" combinat cu "*" din care a rezultat "/*", adică secvenţa de deschidere a unui comentariu. Forma de scriere complicată `(*p).membru` devine `p->membru`, fiind mai uşor de scris şi de citit.

10.5.2 Utilizarea lui p->membru

După cum se observă în codul care urmează, de exemplu `(*pzi).zi` devine `pzi->zi`.

```
void citeste_data(struct Data *pzi)
{
        printf("zi = ");
        scanf("%i",&pzi->zi);
        printf("luna = ");
        scanf("%i",&pzi->luna);
        printf("an = ");
        scanf("%i",&pzi->an);
}
```


Ce se petrece însă dacă membrul structurii este la rându-i o structură?

```
1  /* STRUCT3.C */
2  #include<stdio.h>
3  struct Data
4  {
5          char zi;
6          char luna;
7          int an;
8  };
9
10 struct Persoana
11 {
12         char nume[80];
13         struct Data datanasterii;
14 };
15
16 void citeste_pers(struct Persoana *);
17 void scrie_pers(struct Persoana *);
18
19 int main(void)
20 {
21         struct Persoana unul;
22
23         citeste_pers(&unul);
```

```
24          scrie_pers(&unul);
25
26          return 0;
27  }
28
29  void citeste_pers(struct Persoana *om)
30  {
31
32          printf("Numele: ");
33          gets(om->nume);
34          printf("zi = ");
35          scanf("%i",&om->datanasterii.zi);
36          printf("luna = ");
37          scanf("%i",&om->datanasterii.luna);
38          printf("an = ");
39          scanf("%i",&om->datanasterii.an);
40  }
41
42  void scrie_pers(struct Persoana *om)
43  {
44          printf("Numele este: %s\n", om->nume);
45          printf("\tData nasterii este: %i, %i, %i\n",om->datanasterii.zi,\
46  om->datanasterii.luna, om->datanasterii.an);
47  }
```

Rezultate:

```
Numele: Ion Apostol
zi = 11
luna = 12
an = 1947
Numele este: Ion Apostol
        Data nasterii este: 11, 12, 1947
```

Observaţi că **`datanasterii`** este un membru structură, membrul **`zi`** este accesat prin **`om->datanasterii.zi`** şi nu prin **`om->datanasterii->zi`**, din moment ce **`datanasterii`** este o structură şi nu un poantor la structură.

10.5.3 Problema transferului prin poantor

Am spus deja că transferul printr-un poantor este mai eficient decât cel prin referinţă, dar în cazul acestui transfer este posibilă modificarea valorilor poantate de poantor, ceea ce este uneori un dezavantaj. În cazul tablourilor acest fapt este de neevitat, iar elementele tabloului vor putea fi întotdeauna modificate în interiorul funcţiei care le primeşte ca parametru. Un dezavantaj al transferului prin poantor al parametrilor structură este cazul în care nu dorim să realizăm modificări ale structurilor în interiorul funcţiei, dar din grabă, de exemplu, în locul operatorului "`==`" scriem "`=`".

Utilizarea lui **`const`**

Soluţia acestei probleme este utilizarea cuvâtului cheie **`const`** discutat în paragraful 3.3.2. La fel ca în cazul unei variabile obişnuite, este posibil ca un poantor să fie declarat poantor constant. De exemplu, prin declaraţia **`const float *x;`** se declară **`x`** ca un poantor la un **`float`** constant. Valoarea stocată în poantor se poate modifica. De exemplu, putem scrie **`x--;`**, dar valoarea stocată la adresa respectivă nu se poate modifica, adică nu se poate scrie **`*x=21;`** întrucât compilatorul va semnala linia cu eroare. În cazul structurii, folosirea lui **`const`** conduce la un poantor la o structură constantă, motiv pentru care membrii structurii nu vor putea fi modificaţi prin intermediul poantorului.

10.6 Întoarcerea unei valori structură de către o funcţie

Funcţiilor li se pot transfera structuri prin valoare şi pot întoarce structuri, deşi de multe ori această modalitate de întoarcere este ineficientă. Consideraţiile legate de eficienţa întoarcerii structurilor rămân aceleaşi ca în cazul transferului de parametri. Cu cât structura este mai mare, cu atât eficienţa transferului prin valoare este mai scăzută faţă de cea prin poantor. Există cazuri în care beneficiile transferului unei valori structură permit implementarea unor operatori. Un astfel de caz se prezintă în continuare. Presupunem că dorim să scriem o funcţie care să poată aduna două numere complexe. Întoarcerea unei valori structură permite scrierea unei funcţii **adunac()** care să apară în dreapta unui egal. Dacă **c1** şi **c2** sunt cele două numere complexe, iar în **c3** dorim să stocăm suma celor două numere, scriem **c3=c1+c2**. Folosind funcţia **adunac()**, ce realizează adunarea complexă, scriem **c3=adunac(c1,c2)** (vezi **linia 15**).


```
1  /* COMPLEX1.C */
2  #include <stdio.h>
3
4  struct complex
5  {
6          double real,imaginar;
7  };
8
9  void afisarec(struct complex a);
10 struct complex adunac(struct complex a, struct complex b);
11
12 int main(void)
13 {
14         struct complex c1={1.,2.}, c2={-2.,3.}, c3;
15         c3=adunac(c1,c2);
16         afisarec(c3);
17
18         return 0;
19 }
20
21 void afisarec(struct complex a)
22 {
23         printf("%+lf %+lf\n",a.real,a.imaginar);
24 }
25
26 struct complex adunac(struct complex a, struct complex b)
27 {
28         struct complex suma=a;
29
30         suma.real+=b.real;
31         suma.imaginar+=b.imaginar;
32
33         return suma;
34 }
```


Rezultat:

```
-1.000000 +5.000000
```

Varianta de scriere a programului folosind poantorii este:

```
1  /* COMPLEX2.C */
2  #include <stdio.h>
3
4  struct complex
5  {
6          double real,imaginar;
7  };
8
9  void afisarec(struct complex a);
10 void adunac(struct complex *a, struct complex *b, struct complex *sumac);
11
12 int main(void)
13 {
14         struct complex c1={1.,2.}, c2={-2.,3.}, c3;
15         adunac(&c1,&c2,&c3);
16         afisarec(c3);
17
18         return 0;
19 }
20
21 void afisarec(struct complex a)
22 {
23         printf("%+lf %+lf\n",a.real,a.imaginar);
24 }
25
26 void adunac(struct complex *a, struct complex *b, struct complex *sumac)
27 {
28         sumac->real=a->real+b->real;
29         sumac->imaginar=a->imaginar+b->imaginar;
30 }
```


Pentru acest program se foloseşte transferul prin poantori pentru toţi parametrii. Chiar dacă nu există ineficientul transfer prin valoare, observaţi modul în care suntem obligaţi să apelăm funcţia de adunare `adunac(&c1,&c2,&c3);` (vezi `linia 14`) spre deosebire de linia `c3=adunac(c1,c2);` din programul precedent.

O eroare fatală

Probabil unii mai grăbiţi ar fi rescris codul primului program pentru a lucra cu transferul prin poantori sub forma:


```
struct complex* adunac(struct complex a, struct complex b)
{
        struct complex suma=a;

        suma.real+=b.real;
        suma.imaginar+=b.imaginar;

        return &suma;
}
```


Programul conţine o eroare fatală. Variabila `suma` este stocată pe stivă, adică spaţiul pentru aceasta este alocat pe stivă la intrarea în funcţie şi eliberat la ieşirea din aceasta, astfel că după ieşirea din funcţie, adresa poantată de `&suma` nu mai face parte din spaţiul de date al programului.

10.7 Crearea unei liste înlănțuite prin folosirea structurilor

Limbajul C permite crearea şi manipularea unor structuri de date avansate cum ar fi listele înlănţuite, arborii, grafele etc. În cele ce urmează prezint cel mai simplu caz, lista simplu înlănţuită, folosită pentru stocarea unor numere întregi (`int`). **`Lista simplu înlanțuită`** este o structură de date în care fiecare element conţine un poantor la următorul

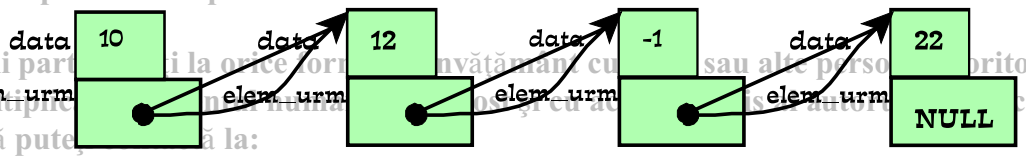


Figura 20

element, formându-se astfel o listă liniară. Reprezentarea ei grafică o găsiţi în *Figura 20*, iar un element al listei se declară prin structura:

```
struct element {
        int data; /* aici este stocata data "utila" */
        struct element * elem_urm;
}
```


Structura cu numele **`element`** are o particularitate curioasă şi anume aceea că se referă la ea însaşi. Am întâlnit deja aşa ceva în cazul funcţiilor când am vorbit despre recursivitate. După cum se vede, aceasta apare atât la nivelul funcţiilor cât şi la cel al datelor. Structura **`element`** este formată dintr-un întreg de tipul **`int`** şi un poantor la o structură de tipul **`element`**. Deşi compilatorul nu ştie exact ce înseamnă un poantor la o structură de tipul **`element`**, el cunoaşte numărul de octeţi necesari pentru stocarea unui poantor. Astfel creează un spaţiu în structură necesar pentru stocarea unui poantor. În programul **`LISTA1.C`** lista este creată prin declararea a patru variabile de tipul **`element`**. Observaţi că mai întâi este declarat şi iniţializat elementul **`e4`** care este ultimul element al listei. Întrucât acest element nu are urmaş, fiind ultimul, valoarea corespunzătoare poantorului la următorul element de listă este iniţializată cu valoarea **`NULL`**. Apoi este declarat **`e3`** şi legat de **`e4`** prin iniţializarea din **`linia 12`**. Aceasta este singura metodă prin care elementele listei pot fi legate, adică nu se poate să-l declarăm pe **`e3`** şi să încerăm să-l legăm de un **`e4`** care încă nu există (pentru că din moment ce nu există nu-i cunoaştem adresa).


```
1  /* LISTA1.C */
2  #include<stdio.h>
3
4  struct element {
5          int data;
6          struct element *elem_urm;
7  };
8
9  int main(void)
10 {
```

```
11      struct element e4 = {22, NULL};
12      struct element e3 = {-1, &e4};
13      struct element e2 = {12, &e3};
14      struct element e1 = {10, &e2};
15
16      struct element *el_curent = &e1;
17
18      while (el_curent != NULL)
19      {
20              printf("Valoarea este: %i\n", el_curent->data);
21              el_curent=el_curent->elem_urm;
22      }
23
24      return 0;
25  }
```


Rezultate:

```
Valoarea este: 10
Valoarea este: 12
Valoarea este: -1
Valoarea este: 22
```

O alternativă pentru crearea listei este codul:

```
1  struct element e4 = {22, NULL};
2  struct element e3 = {-1, NULL};
3  struct element e2 = {12, NULL};
4  struct element e1 = {10, NULL};
5  struct element *el_curent;
6
7  e3.elem_urm = &e4;
8  e2.elem_urm = &e3;
9  e1.elem_urm = &e2;
10 el_curent=&e1;
```


Aici în prima etapă poantorii nu poantează nicăieri, fiind iniţializaţi toţi cu valoarea **NULL**, după care poantorii sunt completaţi prin codul din **liniile 7-10** care duce la înlănţuirea elementelor listei. Ordinea iniţializărilor este importantă. De exemplu, dacă s-ar fi folosit secvenţa:

```
e4.elem_urm = &e3;
e3.elem_urm = &e2;
e2.elem_urm = &e1;
el_curent=&e4;
```

legăturile între elemente se fac în sensul opus (**e4** este primul element şi **e1** este ultimul), după cum se poate observa la afişarea rezultatelor:

```
Valoarea este: 22
Valoarea este: -1
Valoarea este: 12
Valoarea este: 10
```

10.7.1 Parcurgerea listei simplu înlănțuite

Porţiunea de program care realizează afişarea elementelor listei este un exemplu clasic de vizitare a tuturor elementelor din listă. Spaţiul datelor este prezentat în *Figura 21*, adresele folosite în figură au un rol didactic, uşurând înţelegerea operaţiei de parcurgere a listei. Adresa primului element de listă este stocată în poantorul **el_curent** prin atribuirea din **linia 16** (vezi **LISTA1.C**). Modul de înlănţuire a elementelor listei ne obligă să o parcurgem într-un singur sens. Dacă în locul **liniei 16** scriam **struct element *el_curent = &e4;** parcurgerea listei ar fi fost compromisă, pentru că **e4** nu are un element următor. Acest ultim element al listei se mai numeşte şi **coada listei**, iar primul element, acela cu care se începe parcurgerea listei, **capul listei**.

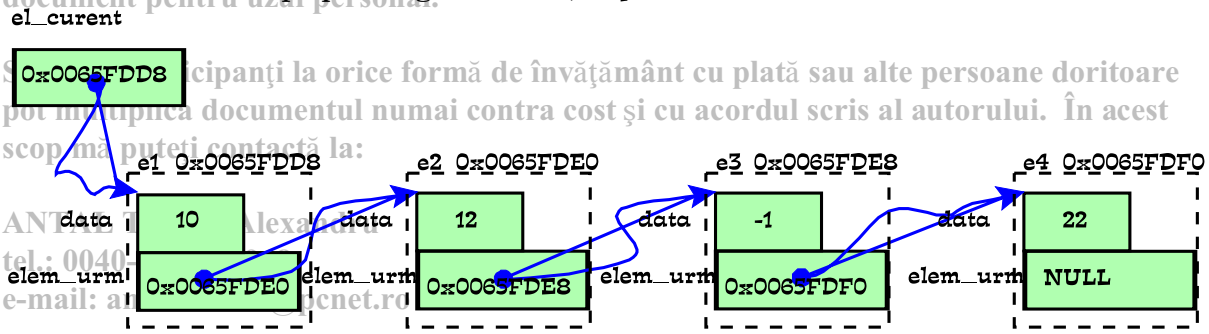


Figura 21

Parcurgerea listei prin vizitarea tuturor elementelor ei se realizează prin ciclul **while** din **linia 18**. Cât timp **el_curent** este diferit de valoarea **NULL** se va executa afişarea valorii stocate în membrul **data** al structurii, apoi prin linia de program **el_curent = el_curent->elem_urm;** adresa următorului element din listă este transferată în **el_curent**. Observaţi că în poantorul **el_curent** vor fi stocate pe rând adresele **0x0065FDD8**, **0x0065FDE0**, **0x0065FDE0**, **0x0065FDF0** şi, în final, valoarea specială **NULL**, caz în care condiţia de reluare a corpului ciclului **while** devine **falsă**.

10.8 Instrucţiunea typedef

În C clasic, tipurile de date definite de utilizator nu primeau un nume, excepţie făcând **struct** (şi **union** care va fi discutat în paragraful următor). Chiar şi în acest caz, declaraţiile acestor variabile trebuiau precedate de cuvintele **struct** sau **union**. ANSI C, implementează un nou nivel de ascundere a informaţiilor prin **typedef**. Aceasta permite crearea de noi nume pentru tipurile de date, prin asocierea unui tip de dată existent la un nume, apoi putându-se declara variabile de acel tip. Iată câteva exemple:

```
typedef int *ptrInt;
typedef char Nume[22];
typedef enum {mascul, femela, necunoscut} sex;
typedef struct {
            Nume nume,prenume;
            sex Gen;
            short Varsta;
            float Medie;
        } student;
typedef student andestudii[100];
```

Formă:	typedef nume_def_tip identificator

typedef permite simplificarea scrierii declaraţiilor de date conducând la compactizarea şi uniformizarea programelor. Nu se poate folosi în interiorul unei funcţii. O declaraţie **typedef** specifică un nume care, în cadrul vizibilităţii declaraţiei, devine sinonim al tipului specificat în secţiunea **nume_def_tip** din declaraţie. Spre deosebire de declaraţiile **struct**, **union** şi **enum**, **typedef** nu creează un tip de dată nou, ci numai un nou nume pentru un tip deja existent. Se poate spune că **typedef** este asemenea lui **#define** cu excepţia că este interpretată de compilator şi nu de preprocesor. Fie declaraţiile:

```
typedef char CHAR; /* tip caracter */
typedef CHAR * PSTR; /* poantor la un sir (char *) */
typedef CHAR far * LPSTR /* poantor far la un sir */
```

Numele introduse prin declaraţiile de mai sus sunt sinonime cu tipurile următoare:

Nume	Tip sinonim
CHAR	**char**
PSTR	**char ***
LPSTR	**char __far ***

Rolul lui **typedef**

Conform celor spuse deja **typedef** nu aduce nimic nou în semantica declaraţiilor, adică variabilele declarate cu **typedef** au aceleaşi proprietăţi cu cele ale căror declaraţii sunt scrise explicit. Primul motiv al folosirii lui **typedef** constă în creşterea lizibilităţii programelor. Al doilea constă în faptul că **typedef** permite ascunderea datelor asigurând independenţa implementării de detaliile acesteia. Dacă un program are porţiuni dependente de maşina pe care se rulează, folosirea lui **typedef** permite ca numai declaraţiile **typedef** să fie modificate la mutarea programului pe o altă maşină.

10.9 Tipul de dată union

union sau tipul reuniune este similar cu **struct**, dar în cazul unei variabile de tipul **union** toţi membrii acesteia sunt stocaţi începând de la aceeşi adresă de memorie, sub numele variabilei de tipul **union**, practic partajându-se memoria totală a reuniunii între toate componentele ei. Se va aloca spaţiul maxim pentru fiecare variabilă de tipul **union**, prin alocarea spaţiului pentru membrul de dimensiunea cea mai mare a reuniunii. Sub numele reuniunii se va stoca o singură componentă. Eventual, este posibil accesul sub forma mai multor tipuri de date la acelaşi grup de locaţii de memorie alocate pentru reuniune. Forma generală este:

```
Formă:      union <nume_union> {
                        <tip1> <nume_membru1>;
                        <tip2> <nume_membru2>;
                                ...
                        <tipn> <nume_membrun>;
            };
```

Variabila de tipul **union** permite deci stocarea la momente diferite a unor valori de tipuri diferite în aceeaşi zonă de RAM. Utilizarea lui **union** poate produce erori grave în execuţie dacă se manipulează o variabilă de tipul **union** fără a şti care din valorile posibile au fost stocate în ea. O modalitate de evitare a acestei capcane stă în stocarea într-o variabilă auxiliară a unei valori ce să permită identificarea tipului valorii stocate în **union** .


```
/* UNION.C */
#include<stdio.h>
#define INTREG 1
#define REAL 2
#define CARACTER 3
main()
{
        struct {
                int iCare;
                union{
                        int iA;
                        float fA;
                        char cA;
                };
        }tabloustruct[3];
        int i;

        tabloustruct[0].iCare=INTREG;
        tabloustruct[0].iA=10;

        tabloustruct[1].iCare=REAL;
        tabloustruct[1].fA=11.123457;

        tabloustruct[2].iCare=CARACTER;
        tabloustruct[2].cA='a';

        for(i=0;i<3;i++) {
       switch (tabloustruct[i].iCare) {
                case INTREG:printf("intreg = %d\n", tabloustruct[i].iA);
                            break;
                case REAL:printf("real = %f\n", tabloustruct[i].fA);
                            break;
                case CARACTER:printf("caracter = %c\n", tabloustruct[i].cA);
                            break;
                defalut: printf("Eroare/n");
        }
        }
}
```


Rezultate:

```
intreg = 10
real = 11.1234
caracter = a
```

10.10 Câmpuri de biţi

Ca o completare adusă membrilor unui tip **struct** sau tip **union**, identificatorului unui membru i se poate specifica printr-o declaraţie numărul de biţi pe care se va stoca. Acest tip de membru se va numi câmp de biţi. Lungimea se specifică în declaratorul numelui de

câmp prin caracterul ":". Un câmp de biţi este interpretat ca tip întreg.

Formă: `tip declarator : expresie constantă`

`expresie constantă` specifică numărul de biţi ai câmpului. Tipul declaratorului poate fi: `unsigned int`, `signed int` sau `int`, iar `expresie constantă` trebuie să fie un întreg pozitiv. Nu este permisă declararea de tablouri, poantori de câmpuri de biţi sau de funcţii care întorc câmpuri de biţi. Nu se poate aplica operatorul `&` asupra acestor câmpuri. Câmpuri anonime (fără nume) pot fi declarate în scopul alinierii în memorie, dar acestea nu pot fi referite, iar conţinutul lor este nedefinit în timpul execuţiei programului. Mărimea câmpului de biţi trebuie să fie suficient de mare pentru a conţine numărul de biţi. În acest sens, următoarea declaraţie este ilegală:

```
short a:17;
```

În exemplul care urmează se defineşte un tablou bidimensional de structuri cu numele de `ecran`:

```
struct
{
        unsigned short caracter: 8;
        unsigned short culoare:4;
        unsigned short subliniere:1;
        unsigned short blink:1;
} ecran [25][80];
```

Tabloul are 2000 de elemente, fiecare element este o structură cu patru membri câmpuri de biţi: `caracter`, `culoare`, `subliniere` şi `blink`. Mărimea structurii este de 2 octeţi.

ANSI C tratează câmpurile de biţi din punct de vedere semantic ca întregi. Aceasta înseamnă că un câmp de biţi va fi folosit într-o expresie în mod identic cu o variabilă întreagă, indiferent de numărul de biţi din câmp. Pentru compilatorul C al Microsoftului câmpurile de biţi se aliniază în interiorul unui tip `int` începând de la bitul cel mai puţin semnificativ. Fie codul:

```
struc {
        unsigned short a:3;
        unsigned short b:6;
        unsigned short c:7;
} x;

void main(void)
{
        x.a=2; /* in binar  10 */
        x.b=7; /* in binar 111 */
        x.c=0; /* in binar   0 */
}
```

biţii vor fi stocaţi astfel:

```
00000000 00111010
ccccccb bbbbbaaa
```

Observaţi că accesul la câmpurile de biţi este acelaşi cu cel la membri `struct` sau `union`.

Când se folosesc câmpurile cu biţi?

Există cazuri când spaţiul de RAM este critic şi dorim să îl ocupăm cât mai bine, împachetând mai multe variabile într-un singur octet al maşinii. Un alt motiv sunt interfeţele, care impun un anumit mod de grupare a biţilor unui octet, de exemplu, pentru a comanda un dispozitiv hardware legat la calculator.

Universitatea Tenhică din Cluj-Napoca
Facultatea Construcţii de Maşini
Catedra de Mecanică şi Programare
Curs de limbaj C

11 Consideraţii privind stilul de programare în limbajul C - generalizarea sortării prin inserare directă cu ajutorul poantorilor la funcţii

Se numeşte **sortare** procesul de rearanjare, pe baza unui criteriu, a unei mulţimi de obiecte, într-o ordine specifică. Rolul sortării este facilitarea găsirii unor obiecte din mulţimea de obiecte sortate. Teoria algoritmilor de sortare spune că algoritmul de sortare ales este dependent de structura datelor de sortat. În general, metodele de sortare sunt clasificate în două categorii mari: *sortarea fişierelor* şi *sortarea tablourilor*. Cele două strategii sunt uneori numite şi *sortare externă,* pentru că fişierele sunt stocate în memorii externe (de exemplu hard disk-uri), respectiv *internă* pentru că tablourile sunt stocate în memoria internă (RAM) a calculatorului.

11.1 Terminologie şi notaţii

Fiind date articolele $a_1, a_2, ..., a_n$, sortarea constă în permutarea lor în ordinea $a_{k1}, a_{k2}, ..., a_{kn}$, astfel încât fiind dată o funcţie f de ordonare, să avem $f(a_{k1}) \le f(a_{k2}) \le ... \le f(a_{kn})$. În cazurile banale, funcţia de ordonare nu se evaluează pe baza unor expresii matematice, ci se stochează explicit într-un membru numit **cheie** a unei structuri care va ţine şi articolele de sortat. Din cele spuse, tipul de dată structură care stochează cele descrise mai sus este:

```
struct tip_obiect {
        int cheie;
        float a;
        /* "alte componete" ale structurii */
};
```

Comentariul "*alte componente*" din structura de mai sus reprezintă date relevante, caracteristice pentru fiecare obiect de sortat. *cheie* reprezintă un câmp întreg care identifică unic un obiect din mulţimea de sortat. Alegerea lui ca întreg este oarecum arbitrară, practic fiind posibilă alegerea oricărui tip pentru care se poate defini o relaţie

de ordine totală.

În continuare vom studia cazul sortării tablourilor **in situ**, când algoritmul de sortare produce rezultatul în acelaşi tablou în care la începerea sortării erau obiectele nesortate. Există algoritmi de sortare internă care produc rezultatul sortării într-un tablou nou, de aceeaşi dimensiune şi tip cu cel de sortat. Se merge deci pe principiul minimizării risipei de RAM din calculator. Deoarece cantitatea de RAM necesară, conform celor afirmate, se rezumă chiar la tabloul original de sortat, adică sortarea se face în tabloul original nesortat, trebuie să alegem criterii de evaluare a algoritmului studiat diferite de cantitatea de RAM folosită pentru sortare. Eficienţa acestor algoritmi se va măsura prin numărul de comparaţii de chei **C** şi a numărului de modificări prin atribuirile **A** ale obiectelor de sortat în vederea realizării sortării. Aceste numere (**C** şi **A**) sunt funcţii de numărul **n** al obiectelor de sortat. Un algoritm de sortare performant are **C** proporţional cu **n lg (n)** respectiv cu n^2, când algoritmul este mai modest. În continuare se va prezenta algoritmul de sortare prin inserare directă.

11.2 Sortarea prin inserare directă

Metoda constă în divizarea conceptuală a tabloului de sortat *a* într-o secvenţă destinaţie $a_1, ..., a_{i-1}$,deja *sortat*ă pe baza valorilor de chei, şi una *surs*ă, $a_i, ..., a_n$, încă *nesortat*ă. La fiecare pas, începând cu *i=2 (al doilea articol)* şi incrementând pe *i*, elementul cu numărul de ordine *i* din şirul sursă este selectat pentru a fi mutat în secvenţa destinaţie, prin inserare la locul dictat de valoarea cheii. Procesul de găsire a locului obiectului a_i se face prin operaţiile de comparare şi/sau deplasare, inserare. Obiectul a_i se compară cu obiectul curent a_j din secvenţa destinaţie după care, fie se inserează în secvenţa destinaţie, fie obiectul a_j se deplasează la stânga cu o poziţie în secvenţa destinaţie. Algoritmul este formulat în liniile următoare:

variabila a este tablou[1..n] de tipul tip_obiect

sortare_prin_inseraredirecta()
{
    variabile întregi *i, j;*
    variabile de tipul *tip_obiect x;*
    repetă pentru *i:=2* ***până la*** *n*
    {
        x:=a[i];
        j=i-1;
        repetă cât timp *((j >=1)* şi *(x.cheie < a[j].cheie))* **<- COMPARAŢIE**
        {
            a[j+1]=a[j]; **<- ATRIBUIRE**
            j=j-1;
        }
        a[j+1]=x; **<- ATRIBUIRE**
    }
}

Numărul de comparaţii de chei C_i în deplasarea a *i*-a este cel mult *i-1* şi cel puţin **1**. Numărul A_i de atribuiri de obiecte este C_i+1, aşadar numărul total de comparaţii şi atribuiri pentru cele două cazuri extreme sunt:

$$C_{min} = \sum_{i=1}^{n-1}(1) = n-1 \qquad A_{min} = \sum_{i=1}^{n-1}(2) = 2(n-1)$$

$$C_{max} = \sum_{i=1}^{n-1}(i-1) = \frac{1}{2}(n^2-3n+2) \qquad A_{max} = \sum_{i=1}^{n-1}(i) = \frac{1}{2}(n^2-n)$$

Menţionez că în formulele de mai sus *i* reprezintă numărul de reluări a ciclului exterior, în timp ce în descrierea algoritmului acelaşi *i* reprezintă poziţia elementului de inserat. Cel mai favorabil caz apare când obiectele sunt iniţial ordonate. Cel mai defavorabil caz apare când acestea sunt ordonate invers.

11.3 Implementarea algoritmului de sortare


```
/* SINDIR1.C */
#include<stdio.h>
int iTab[10] = {3,6,1,2,3,8,4,1,7,5};

void sinsdir(int a [ ], int N);

int main(void)
{
        int i;
        putchar('\n');
        for(i=0;i<10;i++)
                printf("%d ", iTab[i]);
        sinsdir(iTab,10);
        putchar('\n');

        for(i=0;i<10;i++)
                printf("%d ", iTab[i]);

        return 0;
}
void sinsdir(int a [ ], int N)
{
        int i,j;
        int iX;
        for (i=1;i<N;i++)
        {
                iX=a[i];
                j=i-1;
                while ((iX < a[j]) && (j >= 0))
                {
                        a[j+1]=a[j];
                        j=j-1;
                }
                a[j+1]=iX;
        }
}
```

11.4 Să revenim la problema inițială ... poantorii la funcții

Să considerăm următoarea problemă reală. Dorim să scriem **o funcție care să poată sorta virtual orice colecție de date** ce se poate stoca într-un tablou. Ar putea fi un tablou de şiruri, unul de întregi sau unul de valori reale sau chiar de structuri. **SINSDIR1.C** este scris pentru sortarea unor întregi. Ce ar trebui să modificăm pentru ca programul să lucreze şi cu alte tipuri de date? Începem cu funcţia **sinsdir()**, unde funcţia de comparare va fi modificată.


```
/* SINSDIR2.C */
#include<stdio.h>
int iTab[10] = {3,6,1,2,3,8,4,1,7,5};

void sinsdir(int a [ ], int N);
int compara(int m, int n);

int main(void)
{
        int i;
        putchar('\n');
        for(i=0;i<10;i++)
                printf("%d ", iTab[i]);

        sinsdir(iTab,10);
        putchar('\n');
        for(i=0;i<10;i++)
                printf("%d ", iTab[i]);
        return 0;
}
void sinsdir(int a [ ], int N)
{
        int i,j;
        int iX;
        for (i=1;i<N;i++)
        {
                iX=a[i];
                j=i-1;
                while (compara(iX,a[j]) && (j >= 0))
                {
                        a[j+1]=a[j];
                        j=j-1;
                }
                a[j+1]=iX;
        }
}

int compara(int m, int n)
{
        return (m < n);
}
```

Pentru că dorim ca funcţia de sortare să fie independentă de tipurile de date de sortat, o variantă de lucru este folosirea de poantori la tipuri **void** în loc de poantori la tipul **int**. Pentru început, programul va fi puţin modificat pentru a ne apropia de o variantă din care să se poată face trecerea simplă la varianta cu poantori **void**. Versiunea care urmează este modificată la poantori la **int**.

```
/* SINSDIR3.C */
#include<stdio.h>

int iTab[10] = {3,6,1,2,3,8,4,1,7,5};
void sinsdir(int *pa, int N);
int compara(int *m, int *n);

int main(void)
{
        int i;
        putchar('\n');
        for(i=0;i<10;i++)
                printf("%d ", iTab[i]);
        sinsdir(iTab,10);
        putchar('\n');

        for(i=0;i<10;i++)
                printf("%d ", iTab[i]);

        return 0;
}
void sinsdir(int *pa, int N)
{
        int i,j;
        int iX;
        for (i=1;i<N;i++)
        {
                iX=pa[i];
                j=i-1;
                while (compara(&iX,&pa[j]) && (j >= 0))
                {
                        pa[j+1]=pa[j];
                        j=j-1;
                }
                pa[j+1]=iX;
        }
}
int compara(int *m, int *n)
{
        return (*m < *n);
}
```

Observaţi modificările. Acum lui **sinsdir()** i se transferă un poantor la **int**, iar din funcţia **sinsdir()** se transferă poantori la elementele unui tablou care dorim să fie comparat prin intermediul funcţiei de comparare **compara()**. Normal, în **compara()** se face dereferenţierea poantorilor pentru a putea face comparaţia dorită. În următorul pas se vor

converti poantorii din **sinsdir()** la poantori la tipul **void** pentru ca funcţia să devină şi mai independentă de tipul datelor de sortat.

```
/* SINSDIR4.C */
#include<stdio.h>
```



```
int iTab[10] = {3,6,1,2,3,8,4,1,7,5};
```



```
void sinsdir(int *pa, int N);
int compara(void *m, void *n);
```



```
int main(void)
{
        int i;
        putchar('\n');
        for(i=0;i<10;i++)
                printf("%d ", iTab[i]);

        sinsdir(iTab,10);
        putchar('\n');
        for(i=0;i<10;i++)
                printf("%d ", iTab[i]);

        return 0;
}

void sinsdir(int *pa, int N)
{
        int i,j;
        int iX;
        for (i=1;i<N;i++)
        {
                iX=pa[i];
                j=i-1;
                while (compara((void *)&iX, (void*)&pa[j]) && (j >= 0))
                {
                        pa[j+1]=pa[j];
                        j=j-1;
                }
                pa[j+1]=iX;
        }
}

int compara(void *m, void *n)
{
        int *m1, *n1;
        m1= (int *) m;
        n1= (int *) n;
        return (*m1 < *n1);
}
```


Observaţi că în apelul lui **compara()** s-a introdus operatorul de conversie forţată a tipului poantorilor de la **void** la tipul de dată de sortat în momentul actual. Pentru că ceea ce se transferă lui **sinsdir()** este încă tot un poantor la întregi, ar trebui să convertim aceşti poantori în poantori la tipul **void** când îi transferăm ca parametri în linia de apel a lui **compara()**.

Să revenim la problema lui **sinsdir()**. Dorim să facem primul parametru de tipul **void**, dar în acelaşi timp trebuie făcut ceva cu **iX** care şi el este de tipul **int**. De asemenea, acolo unde apare **iX=pa[i];** tipul lui **pa[i]** trebuie să fie cunoscut pentru a şti câţi octeţi vor fi copiaţi în **iX** sau în orice altă variabilă care ar putea fi substituită cu **iX**.

Versiunea **SINSDIR4.C**, prin funcţia **sinsdir()** obţine informaţii despre tipul datelor de sortat, deci şi despre dimensiunea individuală a fiecărui element de tablou datorită faptului că primul parametru este poantor la tipul întreg. Dacă dorim să modificăm pe **sinsdir()** ca să sorteze orice tip de dată, trebuie ca acel poantor să fie la tipul **void**. Dar aşa se pierde informaţia legată de mărimea individuală a elementului de tablou, motiv pentru care în **SINSDIR5.C** acesta se transferă ca parametru separat.


```
/* SINSDIR5.C */
#include<stdio.h>
#include<string.h>
long iTab[10] = {3,6,1,2,3,8,4,1,7,5};

void sinsdir(void *pa, size_t marime, int N);
int compara(void *m, void *n);
int main(void)
{
        int i;
        putchar('\n');
        for(i=0;i<10;i++)
                printf("%d ", iTab[i]);

        sinsdir(iTab, sizeof iTab[0],10);
        putchar('\n');
        for(i=0;i<10;i++)
                printf("%ld ", iTab[i]);

        return 0;
}

void sinsdir(void *pa, size_t marime, int N)
{
        int i,j;
        unsigned char cBufX[4];
        unsigned char *bpa=pa;

        for (i=1;i<N;i++)
        {
                /* iX=pa[i]; */
                memcpy(cBufX, bpa+i*marime, marime);
                j=i-1;
                while (compara( (void *) &cBufX, (void *) (bpa +j*marime)) && (j >= 0))
                {
                        memcpy(bpa+(j+1)*marime, bpa+j*marime, marime);
                        /* pa[j+1]=pa[j]; */
                        j=j-1;
                }
                memcpy(bpa+(j+1)*marime, cBufX, marime);
                /* a[j+1]=iX; */
```

```
        }
}
int compara(void *m, void *n)
{
        long *m1, *n1;
        m1= (long *) m;
        n1= (long *) n;
        return (*m1 < *n1);
}
```

Observaţi că tipul de dată al tabloului a trecut de la **int** la **long**, aceasta pentru a ilustra că sunt necesare modificari şi în **compara()**. Problema lui **iX** este şi ea rezolvată printr-un buffer de 4 caractere fără semn folosit pentru a stoca un **long**. Poantorul la caractere fără semn ***bpa** este folosit pentru a poanta baza (primul element) tabloului de sortat.

A fost necesar să se modifice şi ceea ce i se tranfseră lui **compara()** şi felul în care se face deplasarea elementului de inserat. Utilizarea lui **memcpy()** şi notaţia de poantor în locul celei de tablou duc la reducerea dependenţei de tipul de dată.

Trecem în continuare la **SINSDIR6.C** care sortează şiruri de caractere în locul celor de întregi **long**. Desigur, trebuie să modificăm funcţia de comparaţie din moment ce şirurile se compară altfel decât întregii.

```
/* SINSDIR6.C */
#include<stdio.h>
#include<string.h>
#define MAX_BUF 256

char sTab2[6][23] = {"Shivambu Kalpa",
                     "De magistro",
                     "Yoga Sutra",
                     "Paramarthasara",
                     "Shiva-Purana",
                     "Paduka Panchaka"};

void sinsdir(void *pa, size_t marime, int N);
int compara(void *m, void *n);

int main(void)
{
        int i;
        putchar('\n');
        for(i=0;i<6;i++)
                printf("%s\n", sTab2[i]);
        sinsdir(sTab2,23,6);
        putchar('\n\n');

        for(i=0;i<6;i++)
                printf("%s\n", sTab2[i]);
        return 0;
}

void sinsdir(void *pa, size_t marime, int N)
{
```

```
        int i,j;
        unsigned char cBufX[MAX_BUF];
        unsigned char *bpa=pa;

        for (i=1;i<N;i++)
        {
                memcpy(cBufX, bpa+i*marime, marime);
                j=i-1;
                while ((j>=0) && (compara((void *)&cBufX, (void *)(bpa +j*marime)) != 0))
                {
                        memcpy(bpa+(j+1)*marime, bpa+j*marime, marime);
                        j=j-1;
                }
                memcpy(bpa+(j+1)*marime, cBufX, marime);
        }
}
int compara(void *m, void *n)
{
        int k;
        char *m1 = m;
        char *n1 = n;
        k=strcmp(m1,n1);
        if (k < 0) return 1;
        return 0;
}
```


Faptul că **sinsdir()** nu s-a schimbat în raport cu varianta din **SINSDIR5.C** indică faptul că funcţia poate sorta un domeniu larg de tipuri de date. Ceea ce a rămas de făcut este transferul numelui funcţiei de comparare pe care dorim să o utilizăm pentru ca algoritmul să fie într-adevăr general. La fel cum numele tabloului este adresa primului element din tablou în segmentul de date, numele funcţiei se descompune în adresa acelei funcţii din segmentul de cod. Astfel, vom folosi un poantor la funcţie, în acest caz unul la funcţia de comparare.

Poantorii la funcţii trebuie să fie identici ca număr şi tip de parametri, respectiv de valori returnate, cu funcţiile la care poantează. Pentru cazul nostru se declară poantorul la funcţie sub forma:

```
int (*pf)(const void*, const void *);
```


Atenţie, dacă în loc de linia de mai sus am fi scris **int *pf(const void*, const void *)** am fi avut prototipul pentru o funcţie care întoarce un poantor la întreg. Aceasta pentru că în C operatorul paranteză **()** are precedenţă mai mare decât operatorul poantor *. Prin punerea parantezelor sub forma **(*pf)** se specifică faptul că declarăm un poantor la o funcţie. Acum modificăm declaraţia funcţiei **sinsdir()** prin adăugarea unui al 4-lea parametru, un poantor la funcţia corectă. Prototipul funcţiei devine astfel:


```
void sinsdir(void *pa, size_t marime, int N, int (*pf)(const void*, const void *));
```

Când se apelează **sinsdir()**, se utilizează numele funcţiei de comparaţie dorit. Programul **SINSDIR7.C** ilustrează modul în care această abordare permite utilizarea aceleiaşi funcţii **sinsdir()** pentru sortarea diferitelor tipuri de date.

```
/* SINSDIR7.C */
#include<stdio.h>
#include<string.h>
#define MAX_BUF 256

long lTab[9] = {1,5,7,3,5,9,8,5,6};
char sTab2[6][23] = {"Shivambu Kalpa",
                     "De magistro",
                     "Yoga Sutra",
                     "Paramarthasara",
                     "Shiva-Purana",
                     "Paduka Panchaka"};
void sinsdir(void *pa, size_t marime, int N, int (*pf)(const void*, const void *));
int compara_long(const void *m, const void *n);
int compara_sir(const void *m, const void *n);
int main(void)
{
        int i;
        puts("\nInainte de sortare:\n");
        for(i=0;i<9;i++)
                printf("%ld ", lTab[i]);
        puts("\n");
        for(i=0;i<6;i++)
                printf("%s\n", sTab2[i]);

        sinsdir(lTab,4,9,compara_long);
        sinsdir(sTab2,23,6,compara_sir);
        puts("\n\nDupa de sortare:\n");
        for(i=0;i<9;i++)
                printf("%ld ", lTab[i]);
        puts("\n");
        for(i=0;i<6;i++)
                printf("%s\n", sTab2[i]);

        return 0;
}

void sinsdir(void *pa, size_t marime, int N,int (*pf)(const void*, const void *))
{
        int i,j;

        unsigned char cBufX[MAX_BUF];
        unsigned char *bpa=pa;

        for (i=1;i<N;i++)
        {
                memcpy(cBufX, bpa+i*marime, marime);
                j=i-1;
                while ((j>=0) && (pf((void *)&cBufX, (void *)(bpa +j*marime)) != 0))
                {
                        memcpy(bpa+(j+1)*marime, bpa+j*marime, marime);
                        j=j-1;
                }
                memcpy(bpa+(j+1)*marime, cBufX, marime);
        }
```

```
}
int compara_sir(void *m, void *n)
{
        int k;
        char *m1 = m;
        char *n1 = n;
        k=strcmp(m1,n1);
        if (k < 0) return 1;
        return 0;
}
int compara_long(void *m, void *n)
{
        long *m1,*n1;
        m1 = (long *) m;
        n1 = (long *) n;
        return (*m1 < *n1);
}
```

Rezultatele afişate de program sunt:

```
Inainte de sortare:

1 5 7 3 5 9 8 5 6

Shivambu Kalpa
De magistro
Yoga Sutra
Paramarthasara
Shiva-Purana
Paduka Panchaka

Dupa de sortare:

1 3 5 5 5 6 7 8 9

De magistro
Paduka Panchaka
Paramarthasara
Shiva-Purana
Shivambu Kalpa
Yoga Sutra
```

Pentru cei care nu au întâlnit încă cele două funcţii utilizate în programele de mai sus, urmează câteva explicaţii.

memcpy - copiază caractere între buffer-e

Prototip: `void *memcpy( void *dest, const void *src, size_t count );`
Fişier prototip: `memcpy` - <`memory.h`> sau <`string.h`>
Valori întoarse: `memcpy()` întoarce valoarea lui `dest`.
Parametri: `dest` - buffer destinaţie;
`src` - buffer sursă;
`count` - numărul de caractere ce se copiază.
Remarcă: Funcţia `memcpy()` copiază `count` octeţi din `src` în `dest`.

strcmp - compară două şiruri - vezi 9.9.2

12 Operaţii de intrare/ieşire (I/E) în C

Operaţiile de **intrare/ieşire**, scrise prescurtat uneori sub forma **I/E**, nu sunt incluse în limbajul C, însă majoritatea programelor trebuie să interacţioneze într-un oarecare fel cu mediul în care se rulează, ANSI C foloseşte în acest scop un set de funcţii definite precis în biblioteca standard a limbajului. Majoritatea funcţiilor de I/E ale C folosesc fluxuri (**streams**) de date. Fluxurile sunt conectate la dispozitive pe care C le numeşte generic fişiere (**files**). În acest mod C reuşeşte să ne scutească de cunoaşterea detaliilor de funcţionare a acestor dispozitive (tastatură, monitor, pe disc, imprimantă). Un **flux** reprezintă o secvenţă de date de tipul text sau binar. Primul flux se numeşte **flux de text** şi conţine numai caractere ASCII, iar cel de al doilea se numeşte **flux binar** şi poate conţine orice fel de secvenţă de biţi. Conexiunea dintre fluxuri şi dispozitivele asociate implicit acestora este dată în tabelul care urmează:

Denumire flux	Este conectat la dispozitivul
stdout	ecran.
stdin	tastatură.
stprn	imprimantă.
stdaux	ecran.
stderr	ecran.

Pentru că teoretic funcţiile de I/E din C sunt făcute să lucreze cu fluxuri, unele din funcţii sunt dedicate exclusiv lucrului cu un anumit flux. De exemplu **printf()** lucrează numai cu fluxul **stdout**. Există şi funcţii "mai evoluate" care pot lucra cu orice fluxuri. De exemplu **fprintf()** lucrează cu oricare dintre fluxurile de mai sus, însă numele fluxului cu care se lucrează trebuie explicit precizat ca parametru al funcţiei. În programul următor efectul celor două funcţii este echivalent:

```
/* FLUX1.C */
#include<stdio.h>
int main()
{
        printf("Te salut. (din printf)\n");
        fprintf(stdout, "Te salut. (din fprintf)\n");

        return 0;
}
```

Rezultate:

```
Te salut. (din printf)
Te salut. (din fprintf)
```

Când un program îşi începe execuţia, codul de pornire al programului deschide automat fluxurile de mai sus, iar programul poate lucra fără alte operaţii premergătoare cu acestea. Dacă, de exemplu, am dori să lucrăm cu un fişier pe disc, fluxul i se asociază fişierului prin operaţia numită deschiderea (**`open`**) fişierului. Numai după această operaţie se va putea folosi **`fprintf()`** pentru a stoca date în fişier.

Rolul unui flux

Am spus că un flux trebuie să fie asociat unui fişier pe care să-l manipuleze, însă **standardul** nu ne spune clar ce este un flux şi care este rolul lui. De asemenea, nu se specifică modul în care fluxul trebuie să fie implementat, chestiunea fiind lăsată la latitudinea celor care scriu compilatorul. În vremurile când dispozitivele de intrare/ieşire erau încete, programele în curs de execuţie în RAM rulau mult mai rapid decât aceste dispozitive erau în stare să furnizeze datele cerute de program. De exemplu, când un program citea caractere individuale de pe disc, acesta era nevoit să aştepte până ce discul făcea o rotaţie şi sectorul cu data în cauză putea fi citit apoi pentru citirea următorului caracter din program, aştepta până când discul era în poziţia bună. Producătorii de dispozitive hardware au inventat pentru eliminarea acestei aşteptări memoria **`cache`** pentru ca discul să nu citească numai un singur caracter. În acest caz deşi programul citea un singur caracter, el era extras din cache şi perioada de aşteptare era mult mai mică (asta pentru că în cache se citeau în avans mai multe caractere fără a fi siguri că va fi nevoie de ele; algoritmi complecşi erau folosiţi pentru a determina care caractere să fie stocate şi care să fie descărcate din memoria cache). Acest cache este o memorie tampon între dispozitiv şi program care, prin preîncărcarea cu date, poate duce la creşterea vitezei de lucru cu dispozitivul. Fluxurile realizează aceeaşi operaţie ca şi memoria cache la nivel software. Astfel, dacă dispozitivul cu care se lucrează nu are la nivel hardware o memorie cache, software-ul rezolvă acest neajuns (probleme apar dacă însă acesta are, deoarece cache-ul este dublat şi viteza scade uşor din acest motiv).

De ce există **`stdout`** şi **`stderr`**?

Din tabelul prezentat anterior se observă că atât **`stdout`** cât şi **`stderr`** afişează informaţiile pe ecran. Întrebarea care se pune este de ce a fost nevoie de două fluxuri care lucrează cu acelaşi dispozitiv? Răspunsul se află în filozofia sistemului de operare UNIX care a crescut împreună cu C. UNIX a introdus noţiunile de redirectare şi conductă (**`pipe`**) care au devenit atât de populare încât noile sisteme de operare MS-DOS, Windows 95/98/NT şi OS/2 le-au implementat şi ele. Dacă avem un program executabil numit **prg.exe**, lansat în execuţie sub forma **prg**, el va folosi ca ieşire ecranul monitorului, dar lansat în execuţie prin linia **prg > fisier.txt**, va folosi ca ieşire fişierul cu numele **fisier.txt** care va fi creat în directorul curent şi va conţine ceea ce a fost anterior afişat pe ecran. Deci ieşirea normală, adică ecranul monitorului, a fost redirectată către disc. O altă variantă ar fi folosirea unei conducte. De exemplu, dacă lansarea în execuţie se face prin **prg | print**, datele de ieşire ale lui **prg** sunt transmise automat programului **print** care va afişa conţinutul primit de la **prg** la imprimantă. Cele două idei de mai sus au devenit fundamentale pentru UNIX, însă apărea o problemă în cazul erorilor: dacă **prg** dorea să afişeze un mesaj de eroare, acesta fie era inserat în fişierul **fisier.txt** la grămadă cu datele utile, fie era afişat şi el la imprimantă. Era nevoie ca aceste mesaje să fie afişate pe ecran indiferent de redirectările pe care le făcea utilizatorul. Problema a fost rezolvată prin crearea a două fluxuri **`sdtout`** care se poate redirecta, şi **`stderr`** care este independent de operaţiile de redirectare.

12.1 Funcţii de ieşire pe stdout

printf - realizează ieşirea cu format pe fluxul stdout

Prototip: `int printf(const char *format, [, argumente] ...);`
Fişier prototip: `printf - <stdio.h>`
Valori întoarse: `printf()` întoarce numărul de caractere afişate sau o valoare negativă dacă a apărut o eroare.
Parametri: `format` - şir de formatare a informaţiilor afişate;
`argumente` - argumente opţionale.

Remarcă: funcţia `printf()` afişează formatat un grup de caractere şi valori pe fluxul standard de ieşire, `stdout`. Dacă argumentele există, în şirul format trebuie să existe specificaţii stricte care să determine modul de afişare a argumentelor. `format` este un şir de caractere, deci o secvenţă de caractere alfanumerice cuprinse între ghilimele. Acesta trebuie să fie prezent în orice instrucţiune `printf()`, controlând modul în care se converteşte, formatează şi afişează fiecare argument al funcţiei. `format` conţine două categorii de caractere, `caractere simple` şi `specificatori de conversie`. `Caracterele simple` sunt pur şi simplu copiate pe ecran, iar `specificatorii de conversie` acţionează asupra argumentelor [, `argumente`] realizând traducerea acestora din reprezentarea internă folosită de C la cea de caractere înainte de afişarea lor pe `stdout`. În exemplul care urmează, `"Suma este %d \n"` este `format`, `%d` şi `\n` sunt specificatori de conversie, iar `Suma este` este un grup de caractere simple. `iSuma` este unicul argument al funcţiei `printf()`, fiind corespuzător lui [, `argumente`] din forma generală a funcţiei:

```
printf("Suma este %d \n", iSuma);
```


Explicaţia de mai sus este un pic trasă de coadă pentru că oricine observă că `printf()` are în realitate **2** argumente, `"Suma este %d \n"` şi `iSuma`. De ce am spus atunci că unicul argument al funcţiei este `iSuma`? Funcţia `printf()` necesită cel puţin un argument, în sensul că acesta este obligatoriu, motiv pentru care nu l-am mai numărat şi pe acesta.

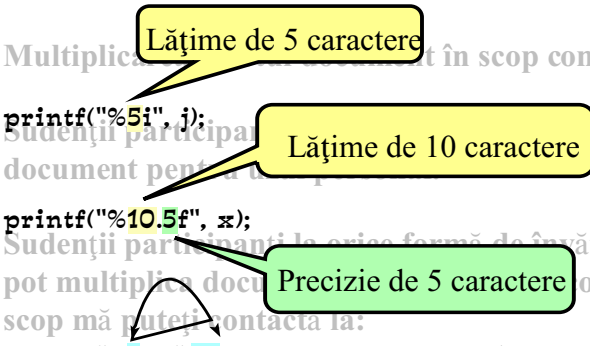
`Există două categorii de specificatori de conversie`, unii încep cu simbolul `%` (procent) şi sunt, de obicei, urmaţi de un singur caracter, purtând denumirea de `specificatori de format` sau `descriptori de format`, şi specifică tipul datei de afişat şi modalitatea de formatare a acesteia la afişarea pe ecran. Pentru fiecare specificator de format trebuie să existe câte un singur `argument` de afişat; dacă tipul acestuia şi cel specificat de descriptor nu sunt identici, C va încerca să realizeze conversia de la tipul `argument` la cel al descriptorului de format. Descriptorul `%d` specifică o afişare a unei valori întregi. Alţi specificatori mai frecvent utilizaţi sunt prezentaţi în tabelul care uremează:

Specificator de conversie	Tipul argumentului	Modul de afişare
`d, i`	`int`	număr zecimal întreg.
`o`	`int`	număr octal fără semn.
`x, X`	`int`	număr hexazecimal fără semn cu `abcdef` pentru `0x` şi cu `ABCDEF` pentru `0X`.
`u`	`int`	număr zecimal fără semn.
`c`	`int`	un singur caracter.

`s`	`char *`	afişează caracterele unui şir până la întâlnirea lui '`/0`' sau până la atingerea numărului maxim de caractere specificate în precizie.
`f`	`double`	număr real implicit în formatul: [-]`m.zzzzzz`; numărul de zecimale `z` este dat de precizia folosită la afişare şi implicit este `6`.
`e, E`	`double`	număr real implicit în formatul: [-]`m.zzzzzz e ±xx` sau [-]`m.zzzzzz E ±xx`; numărul de zecimale `z` este dat de precizia folosită la afişare şi implicit este `6`.
`g, G`	`double`	foloseşte modul de afişare de la `%e` sau `%E` dacă exponentul este mai mic decât `-4` sau mai mare sau egal decât precizia; altfel foloseşte modul de afişare de la `%f`; nu afişează zerourile nesemnificative, numărul de caractere folosite pentru afişare fiind minim.
`p`	`void *, poantor`	numărul afişat este un poantor (adresă de memorie).

Unele exemple prezentate până acum foloseau `%f` pentru afişarea valorilor variabilelor de tipul `float`, iar altele `%lf` pentru afişarea valorilor variabilelor de tipul `double`, deşi tabelul de mai sus nu spune nimic despre tipul `float`. Realitatea este că `printf()` lucrează numai cu `%f` atât pentru tipul `float`, cât şi pentru tipul `double` datorită conversiilor implicite (vezi 6.3.1) făcute în cazul argumentelor de funcţii. Acestea se aplică la `printf()` pentru că argumentele intervin în partea variabilă a funcţiei cazul fiind similar cu cel al apelurilor de funcţii fără prototip.

`Lăţimea` sau `precizia` afişării se poate specifica explicit ca o valoare întreagă sau prin caracterul `*`, caz în care valoarea se calculează prin conversia argumentului, obligatoriu de tipul `int`, care precede obiectul de afişat. În cazul când sunt specificate atât lăţimea cât şi precizia, acestea trebuie separate prin caracterul punct (.).



Observaţi că lăţimea (numărul de caractere) pe care se afişează valoarea argumentului se defineşte prin plasarea după caracterul `%` a unui număr întreg. De exemplu, un câmp întreg cu lăţimea de `5` se scrie `%5d`. Valorile afişate vor fi aliniate la dreapta cu spaţii pe locul poziţiilor din lăţime neocupate de numărul afişat. Pentru afişarea caracterului `%` se scrie `%%`. Ultimul exemplu prezintă cazul lăţimii variabile. Săgeata arată că valoarea numerică de `10` se pune în corespondenţă cu caracterul `*`. Este posibil ca în locul valorii numerice să fie folosită o variabilă de tipul întreg.

Forma generală a specificatorului de format este: `%[indicator][lăţime][.precizie] [F|N|h|l] SpecificatorConversie`

Caracterele `indicator`, sunt `-`, `+`, `#` şi `spaţiu`. Pot apărea în orice combinaţie şi ordine având semnificaţiile:

Indicator	Semnificaţie
`-`	Aliniere la stânga, completare la dreapta cu spaţii. Dacă nu este specificat se face aliniere la dreapta şi completare la stânga cu zerouri sau spaţii.
`+`	Conversiile cu semn rezultă cu `+` sau `-`.
`spaţiu`	Dacă valoarea este pozitivă sau nulă ieşirea începe cu un spaţiu în loc de `+`; valorile negative păstrează semnul `-`.
`#`	Specifică afişarea argumentului convertit conform celor care urmează: • când este folosit împreună cu `o`, `x` sau `X`, prefixează orice valoare nenulă de ieşire cu `0`, `0x`, sau `0X`. Implicit, nu se afişează spaţii; • când este folosit împreună cu descriptorii `e`, `E` sau `f`, va forţa afişarea punctului zecimal în toate cazurile. Implicit, acesta apare numai dacă numărul are şi parte zecimală; • când este folosit împreună cu `g` sau `G` forţează afişarea punctului zecimal în toate cazurile şi previne trunchierea zerourilor care urmează după număr. Implicit, punctul zecimal apare numai dacă după acesta urmează cifre. Dacă acestea sunt zerouri ele vor fi suprimate; • este ignorat când este folosit împreună cu `c`, `d`, `i`, u, sau `s`.

În forma generală a specificatorului de format, toate câmpurile cuprinse între paranteze drepte sunt opţionale. `[F|N|h|l|L]` poartă denumirea de modificatori ai valorii de intrare, specificând lungimea argumentului conform celor care urmează:

`F` - `poantor far`

`N` - `poantor near`

`h` - `short int` sau `short`

`l` - `long` sau `unsigned long`

`L` - `long double`

Aceşti modificatori afectează modul în care `printf()` interpretează tipul de dată corespunzător argumentului. În cazul normal, argumentele corespunzătoare lui `%p` şi `%s` sunt poantori de tipul modelului de memorie folosit la generarea programului executabil. `F` spune că "interpretează argumentul ca pe un poantor far". `h` şi `l` suprascriu mărimea implicită a argumentelor numerice din `printf()`, astfel că `l` se aplică la întregi (`d, i, x`) şi la valori reale (`e, f, g`), iar `h` numai la tipurile întregi. În prezenţa lui `l` argumentul este interpretat ca `long int` pentru `d, i, x`, ca `double` în prezenţa lui `e, f, g` sau ca `long double` în cazul lui `L`.

Un **alt tip de specificator de conversie** începe cu **caracterul backslash** (\). Secvenţa **\n** este cunoscută, din motive istorice, sub numele de **secvenţă escape** şi reprezintă un caracter special inserat într-un şir. În acest caz **\n** inserează un caracter **newline** - trecere la linie nouă şi la început de rând . Câteva din secvenţele escape mai utile sunt:

- **\f** - trecere la început de rând (**formfeed**);
- **\t** - tabulator orizontal (**horizontal tab**);
- **\b** - şterge caracterul din stânga cursorului (**backspace**);
- **\a** - caracterul alertă (**bell**) are ca efect emiterea unui semnal sonor;
- **\r** - retur de car (**carriage return**);
- **\n** - trecere la linie nouă (**newline**);
- **** - caracterul **backslash**, adică \;
- **\?** - caracterul semn de întrebare, adică **?**
- **\ooo** - caracterul ASCII în **notaţie octală**, unde **o** este o cifră între **0** şi **7**;
- **\xhh** - caracterul ASCII cu codul **hexazecimal hh**, unde **h** este o cifră între **0** şi **F**.

Funcţia **printf()** foloseşte primul argument pentru a afla numărul de argumente care urmează şi tipul acestora. Ea va genera probleme dacă numărul de argumente sau tipul acestora este incorect.

puts- realizeaza ieşirea unui şir pe fluxul stdout

Prototip: **int puts(const char *şir);**

Fişier prototip: **puts - <stdio.h>**

Valori întoarse: **puts()** întoarce o valoare nenegativă în caz de succes, iar altfel întoarce constanta **EOF**.

Parametri: **şir** - şirul care va fi trimis pe stdout.

Remarcă: funcţia **puts()** trimite pe stdout şirul, înlocuind caracterul terminator de şir ('**\0**') cu caracterul de trecere la linie nouă ('**\n**').

12.2 Funcţii de intrare de pe stdin

Pentru funcţiile care vor fi prezentate în continuare trebuie să ştiţi că acestea sunt buffer-ate. Aceasta înseamnă că datele introduse pe **stdin** vor fi prelucrate numai după apăsarea tastei <**Enter**>. Un **buffer** este o zonă de memorie folosită pentru stocarea de mesaje. Tipic, buffer-ul are un poantor de intrare care va poanta noile date scrise în buffer, un poantor de ieşire care va poanta la următorul element care va fi citit şi un contor care numără spaţiul încă rămas neocupat din buffer. Principala lui utilizare este decuplarea a două procese, astfel încât citirea şi scrierea să poată opera la viteze diferite şi pentru blocuri de date de mărimi diferite. Există mai mulţi algoritmi pentru utilizarea buffer-elor: **First in First Out** (**FIFO** sau **coadă**), **Last In First Out** (**LIFO** sau **stivă**), **double buffering** care permite ca un buffer să fie citit în timpul în care celălat este înscris, **buffer circular** în care dacă citirea sau scrierea trece de capăt acestea se continuă de la început buffer-ului.

getchar - realizeaza citirea unui caracter de pe stdin

Prototip: `int getchar(void);`

Fişier prototip: `getchar - <stdio.h>`

Valori întoarse: întoarce caracterul citit.


```
1  /* GETCHAR.C */
2  #include<stdio.h>
3  int main(void)
4  {
5          int car;
6
7          while((car = getchar()) != EOF)
8                  printf("<- '%c'\n", car);
9
10         printf("EOF\n");
11
12         return 0;
13 }
```


Rezultate:

```
xyz
<- 'x'
<- 'y'
<- 'z'
<- '
'
a
<- 'a'
<- '
'
EOF
```


Întrucât `stdin` foloseşte un buffer la citirea şirului de caractere `xyz` cu funcţia `getchar()` aceste caractere vor fi prelucrate numai după apăsarea tastei <**Enter**>. La apăsarea acestei taste se observă că ciclul este executat de patru ori, apoi funcţia `getchar()` rămâne fără caractere şi continuă să aştepte date de pe `stdin` (tastatură). La apăsarea lui `a` alte două caractere sunt prelucrate, caracterul '`a`' şi <**Enter**>, iar ciclul este din nou executat de două ori. În sistemul de operare MS-DOS caracterul <**Ctrl**>+<**Z**> care se generează de la tastatură prin apăsarea simultană a tastelor <**Ctrl**> şi <**Z**> este folosit pentru marcarea sfârşitului de fişier. Când `getchar()` întoarce acest caracter ciclul se termină.

De ce `int` şi nu `char`?

Poate că v-a sărit deja în ochi că variabila `car` este declarată de tipul `int` şi nu de tipul `char`. În cartea lui K&R, la fel ca şi în `C standard`, nu se specifică dacă tipul `char` este cu sau fără semn. Deşi chestiunea pare a fi nesemnificativă, valoarea lui `EOF` este definită ca `-1`. Dacă un producător de compilator alege să-l implementeze pe `char` fără semn, atunci când `getchar()` va întoarce valoarea `EOF`, adică pe `-1`, pentru a indica sfârşitul fişierului, în variabila `car` ar fi stocată valoarea `255` (pentru că ea nu poate stoca numere negative).

Compararea valorii `255` cu `-1` face ca ciclul să fie executat la nesfârşit, condiţia de părăsire nefiind îndeplinită niciodată. Folosirea lui `int` garantează că semnul valorii numerice va fi reprezentat corect, motiv pentru care situaţia descrisă nu poate să apară.

scanf - realizează intrarea cu format de pe fluxul stdin

Prototip: `int scanf(const char *format, [, argumente] ...);`

Fişier prototip: `scanf - <stdio.h>`

Valori întoarse: `scanf()` întoarce numărul de câmpuri convertite şi atribuite cu succes; nu sunt incluse câmpurile citite şi neatribuite. Valoarea `0` înseamnă că nu s-a efectuat nici o atribuire. Valoarea `EOF` se întoarce în caz de eroare sau dacă s-a detectat caracterul terminator de fişier.

Parametri: `format` - şir de caractere folosit pentru formatarea informaţiilor de citit;
`argumente` - argumente opţionale.

Remarcă: funcţia `scanf()` citeşte date de pe `stdin` şi le scrie în locaţiile de memorie specificate de argumente. Fiecare argument trebuie să fie un poantor la o variabilă de tipul corespunzător specificatorului de format din `format`. `scanf()` se opreşte din citire când caracterele de conversie din şirul `format` au fost epuizate sau când, datorită unor scăpări, se încearcă introducerea unor caractere care duc la o eroare. Tipurile de caractere care pot fi întâlnite în şirul `format` sunt: `spaţii` sau `tabulatori` care sunt ignorate, `caractere obişnuite` (neprecedate de caracterul `%`), care trebuie să fie identice cu ceea ce se citeşte de pe fluxul de intrare când se ajunge la acestea şi `specificatori de conversie` care se formează la fel ca şi la `printf()` din caracterul `%`, urmat de caracterul `*` opţional de anulare a atribuirii valorii citite, un număr opţional ce specifică lăţimea maximă a câmpului, un `h`, `l` sau `L` opţional care indică lungimea sursei şi un caracter de conversie a cărui semnificaţie se prezintă în tabelul următor.

Specificator de conversie	Tipul argumentului	Data de intrare
`d`	`int`	număr zecimal întreg.
`i`	`int`	număr întreg zecimal, octal (începe cu `0`) sau hexazecimal (începe cu `0x` sau `0X`).
`o`	`int`	număr întreg octal fără semn, cu sau fără `0` la început.
`x, X`	`int`	număr întreg în notaţie hexazecimală fără `0x` sau `0X` la început, folosind `abcdef` pentru `0x` şi `ABCEDF` pentru `0X`.
`u`	`int`	număr întreg zecimal fără semn.
`c`	`int`	un caracter.
`s`	`char *`	şir de caractere.
`f, e, g`	`double`	număr real în virgulă flotantă cu semn, punct zecimal şi exponent opţional.

Funcţia **scanf()** este opusul funcţiei **printf()**. Ea realizează operaţia de intrare de pe **stdin**, în timp ce **printf()** o realizează pe cea de ieşire pe **stdout. scanf()** citeşte caracterele de pe **stdin** şi le interpretează conform caracterelor de conversie din primul argument, şirul **format**. Argumentele opţionale **argumente** trebuie să fie adrese ale unor locaţii de memorie.


```
int a, b;
        . . .
scanf("%d %d", &a, &b);
```


În exemplul de mai sus, se citesc 2 întregi, separaţi printr-un spaţiu; primul va fi stocat în variabila **a**, iar cel de al doilea în variabila **b**. Pentru specificarea adresei la care sunt stocate variabilele se foloseşte operatorul **&** (adresă). Spaţiul între cei doi **%d** înseamnă mai mult decât un simplu spaţiu, şi anume orice combinaţie de un număr de spaţii, tab-uri şi linii noi între cele 2 valori. Dacă, de exemplu, dorim să separăm numerele printr-o singură virgulă în loc de spaţii scriem:


```
scanf("%d,%d", &a, &b);
```

Dacă dorim să citim un şir de caractere scriem:

```
#include <stdio.h>
main()
{
        char nume[30];
        printf("Cum te numesti: ");
        scanf("%s",nume);
        printf("Salut, %s\n",nume);
}
```


Pentru că **nume** este un şir de caractere, care în C se stochează sub forma unui tablou, valoarea lui **nume** este adresa de început a tabloului, din care motiv nu se foloseşte operatorul **&** în faţa lui **nume**. Probleme apar în varianta de mai sus dacă şirul de caractere introdus de la tastatură conţine spaţii. În acest caz sunt citite caracterele numai până la primul spaţiu, el fiind considerat ca terminator al şirului. O soluţie poate fi programul:


```
#include<stdio.h>
main()
{
        char nume[10], prenume1[10], prenume2[10];
        printf("Cum te numesti: ");
        scanf("%s %s %s",nume, prenume1, prenume2);
        printf("Salut, %s %s\n",nume, prenume2);
}
```

O altă variantă ar fi utilizarea funcţiei `gets()` care consideră ca terminator al şirului introdus apăsarea tastei **<Enter>**, moment în care inserează automat la capătul şirului caracterul `null` (`'\0'`).

gets - realizează intrarea unei linii de pe fluxul stdio

Prototip: `char *gets(char *buffer);`

Fişier prototip: `gets - <stdio.h>`

Valori întoarse: `gets()` întoarce argumentul în caz de succes. Un poantor `null` se întoarce în caz de eroare.

Parametri: `buffer` - zonă de RAM în care se va stoca şirul.

Remarcă: funcţia `gets()` citeşte o linie de pe `stdin` şi o stochează în variabila `buffer`. Linia constă în toate caracterele ei, inclusiv primul caracter de trecere la linie nouă (`'\n'`). `gets()` va înlocui acest ultim caracter cu cel terminator de şir (`'\0'`) înainte de a întoarce linia citită.

12.3 Funcţii pentru operaţii cu fişiere

După cum am mai spus, majoritatea funcţiilor de I/E din biblioteca C sunt făcute să lucreze cu fluxuri. De exemplu ştiţi deja că `printf()` este versiunea simplificată a lui `fprintf()`. În multe din cazuri, singura diferenţă dintre funcţiile particulare fluxurilor `stdin` şi `stdout` discutate până acum (`printf()`, `scanf()`, `gets()`, `puts()` etc.) şi cele care pot lucra cu fişiere în general, constă în faptul că pentru cele din urmă se poate specifica numele fluxului cu care operează. Denumirea acestor funcţii care lucrează cu fişiere se formează de obicei din numele funcţiei de I/E discutat până acum, prefixat de caracterul `'f'`. Astfel, `scanf()` devine `fscanf()`, `printf()` devine `fprintf()`, `gets()` devine `fgets()` etc.

12.3.1 Deschiderea fişierelor

Înainte de a putea manipula fişierul, acestuia trebuie să i se asocieze un flux. Această asociere dintre flux şi fişier se realizează prin funcţia `fopen()`. Funcţia va asocia un flux cu un nume de fişier existent pe disc, fiind obligatorie şi specificarea modului de acces la datele din fişier.

fopen - deschide un fişier

Prototip: `FILE *fopen(const char *numefisier, const char *mod );`

Fişier prototip: `fopen - <stdio.h>`

Valori întoarse: `fopen()` întoarce un poantor la fişierul deschis. Un poantor `NULL` este întors dacă există erori.

Parametri: `numefisier` - numele fişierului;
`mod` - tipul accesului permis la fişier.

Declararea unui flux

Un flux este declarat prin tipul `FILE *`, adică un poantor la tipul de dată `FILE`. Dacă doriţi să ştiţi cum este definit acest tip de dată puteţi căuta definiţia lui în fişierul cu prototipuri `stdio.h`. Fără a intra în prea multe

amănunte **FILE**, este o structură care stochează informaţii despre fişier: poziţia buffer-ului, poziţia caracterului curent din buffer, dacă fişierul este citit sau scris, dacă au apărut sau nu erori. Utilizatorul nu trebuie să cunoască aceste detalii de implementare, el fiind responsabil numai pentru declararea unei variabile de acest fel sub forma **FILE* pfis;**, după cum se observă în **linia 6** a programului care urmează. Argumentul folosit pentru specificarea modului de acces la fişier este prezentat în tabelul următor.

Mod de deschidere a fişierului	Semnificaţie
"r"	Deschide un fişier existent pentru citire.
"w"	Deschide sau creează, dacă este cazul, un fişier pentru scriere.
"a"	Deschide sau creează un fişier pentru adăugare.
"r+"	Deschide un fişier existent pentru citire şi scriere.
"w+"	Creează sau deschide un fişier pentru citire şi scriere.
"a+"	Deschide sau creează, dacă este cazul, un fişier pentru citire şi adăugare.
"rb"	Deschide un fişier binar pentru citire.
"wb"	Deschide un fişier binar pentru scriere.
"ab"	Deschide un fişier binar pentru adăugare.
"rt"	Deschide un fişier text pentru citire.
"wt"	Deschide un fişier text pentru scriere.
"at"	Deschide un fişier text pentru adăugare.
"r+b"	Deschide un fişier binar pentru citire şi scriere.
"w+b"	Creează un fişier binar pentru scriere.
"a+b"	Deschide un fişier binar pentru adăugare.
"r+t"	Deschide un fişier text pentru citire şi scriere.
"w+t"	Creează un fişier text pentru citire şi scriere.
"a+t"	Deschide un fişier text pentru citire şi scriere.

12.3.2 Scrierea datelor într-un fişier

Dacă veţi executa programul care urmează veţi găsi pe disc un fişier numit salut.txt, de **11** biţi lungime, care conţine mesajul din **linia 10**.

```
1  /* FOPEN1.C */
2  #include<stdio.h>
```

```
3  int main(void)
4  {
5          FILE *pfis;
6
7          if (( pfis = fopen("salut.txt", "w")) != NULL)
8          {
9                  fprintf(pfis, "Salutare!\n");
10                 fclose(pfis);
11         }
12         else
13         {
14                 printf("Eroare de scriere in fisierul salut.txt\n");
15                 perror("pentru ca:");
16                 return 1;
17         }
18
19         return 0;
20 }
```


`fopen()` întoarce un poantor la o structură de tipul `FILE` care se numeşte mai sus `pfis`. Deci numele fluxului nostru este `pfis` şi este conectat la fişierul **salut.txt** de pe disc. Întrucât membrii structurii `FILE` sunt trataţi direct de funcţiile pentru accesul la fişiere, utilizatorul nu trebuie să-i cunoască pentru ca să lucreze cu aceştia. Ceea ce utilizatorul trebuie să cunoască este numele poantorului la fişier.

Problema accesului la fişiere

Dacă fişierul nu a putut fi deschis, un poantor cu valoarea `NULL` este întors de funcţie. O eroare comună este scrierea numelor de fişiere fără dublarea caracterelor `backslash` ('\'). De exemplu dacă argumentul `numefisier` conţine şi o cale scrisă sub forma `"C:\prog\limbaje\c\salut.txt"`, caracterele \ simple vor fi interpretate ca parte a unei sevenţe escape, scrierea corectă a şirului de mai sus în C fiind: `"C:\\prog\\limbaje\\c\\salut.txt"`. Dacă în DOS şi Windows problema de mai sus poate să apară, în UNIX, pentru că directorii din cale sunt separaţi prin caracterul `slash` ('/'), ea nu mai este actuală (în C '`\t`', '`\n`' etc. au semnificaţia unor secvenţe escape, dar nu şi '`/t`', '`/n`' etc.).

Tratarea erorilor

Întrucât lucrul cu fluxul se derulează prin poantori, în cazul unei erori tot ceea ce putem indica este faptul că o eroare a apărut prin folosirea valorii speciale `NULL` pentru poantor. Dacă `fopen()` întoarce valoarea `NULL` cu siguranţă a apărut o eroare, însă nu ştim prea multe despre natura erorii. S-ar putea ca numele fişierului să fie greşit sau reţeaua pe care lucrăm să fi căzut, din acest motiv **standardul** tratează erorile prin intermediul unei variabile numite `errno` care stochează numărul erorii. Fiecare implementare de C atribuie o valoare unică fiecărei erori posibile, de exemplu `1` poate fi fişier inexistent, `2` memorie insuficientă etc. Este posibilă accesarea valorii numerice a lui `errno` prin scrierea liniei `extern int errno;` la începutul programului. Valoarea numerică va fi actualizată în cazul unei erori, însă numărul în sine ne spune prea puţine. Ar fi mai util dacă am putea afişa un mesaj corespunzător erorii apărute şi aceasta se poate realiza prin funcţia `perror()` care a fost folosită în `linia 16`. Aceasta afişează pe ecran mesajul dintre ghilimele şi un mesaj extras dintr-o tabelă de mesaje de eroare utile pentru a clarifica sursa exactă a erorii.

perror - afişează un mesaj de eroare

Prototip: `void perror(const char *mesaj);`
Fişier prototip: `perror` - `<stdio.h>` sau `<stdlib.h>`
Valori întoarse: nu are.
Parametri: `mesaj` - şirul de caractere care formează mesajul de afişat.
Remarcă: `perror()` afişează un mesaj de eroare pe `stderr`.

`mesaj` este afişat prima oară, urmat de o virgulă, apoi mesajul de eroare corespunzător ultimului apel din bibliotecă care a produs eroarea de sistem şi, în final, caracterul `newline`. Dacă `mesaj` este un poantor `NULL` sau un poantor la un şir `NULL`, `perror()` afişează numai mesajul de eroare de sistem. Pentru rezultate reale, `perror()` se apelează imediat după funcţia care ar putea întoarce eroarea, altfel apelurile de noi funcţii vor suprascrie valoarea numerică a lui `errno`. Dacă operaţia a reuşit fişierul este deschis şi se scrie în el mesajul dorit. Întrucât scrierea pe ecran se face cu funcţia `printf()`, scrierea într-un fişier se va face cu funcţia `fprintf()`. Aceasta are prototipul `int fprintf(FILE *flux, const char *format [, argumente]...);` şi va folosi, asemenea lui `printf()`, un şir `format` pentru formatarea datelor şi o listă de argumente opţionale. În plus, `fprintf()` are ca argument şi poantorul la fişierul în care va realiza scrierea (pentru cazul de mai sus, `pfis`). În `linia 11` se realizează operaţia de închidere a fişierului prin funcţia `fclose()`, necesară pentru ca sistemul de operare să poată actualiza informaţiile legate de fişier în strucura de director. Ea primeşte ca parametru pe `pfis` şi are ca efect deconectarea fluxului de fişierul pe care l-am deschis pe disc.

12.3.3 Citirea şi afişarea datelor dintr-un fişier

Programul care urmează realizează citirea şi afişarea datelor stocate într-un fişier existent pe disc. Funcţia `getc()` este asemenea funcţiei `getchar()`, cu diferenţa că poate lucra cu un poantor la un fişier, prototipul ei fiind `int getc(FILE *flux);`. Funcţia `putc()` trimite un caracter pe un flux care este specificat prin unicul ei parametru. Funcţiile `getchar()` şi `putchar()` sunt definite în termenii lui `getc()`, `putc()`, `stdin` şi `stdout` prin macrodefiniţiile următoare:

```
#define getchar()    getc(stdin)
#define putchar(c)  putc((c), stdout)
```

În programul `FGET.C` se foloseşte un `tablou numefis` de `80` de caractere pentru stocarea numelui de fişier împreună cu calea către acesta, dacă este cazul. Observaţi că în funcţia `scanf()` din `linia 11` prin care se citeşte numele fişierului lăţimea maximă poate fi de `79` de caractere pentru ca nu cumva să se scrie peste ultimul caracter al şirului, el fiind rezervat caracterului terminator de şir. Observaţi de asemenea că numele tabloului nu este precedat de operatorul adresă (`&`), întrucât numele unui tablou în C este un poantor la primul element din tablou.

```
1  /* FGETC.C */
2  #include<stdio.h>
3
4  int main(void)
5  {
6          char numefis[80];
```

```
7       FILE *pfluxin;
8       int car;
9
10      printf("Numele fisierului de afisat: ");
11      scanf("%79s", numefis);
12
13      if ((pfluxin = fopen(numefis, "r")) == NULL)
14      {
15              fprintf(stderr,"Fisierul %s nu a putut fi deschis pt. citire ", numefis);
16              perror("intrucat");
17              return 1;
18      }
19
20      while ((car = fgetc(pfluxin)) != EOF)
21              putchar(car);
22
23      fclose(pfluxin);
24
25      return 0;
26  }
```

Rezultate:

```
Numele fisierului de afisat: salut.txt
Salutare!
```

Numele unui fişier existent deja pe disc

Şi aici `perror()` se foloseşte pentru a genera un mesaj care explică natura erorii. Observaţi că în `linia 17` se foloseşte `return 1;` pentru ca programul să-i întoarcă sistemului de operare o valoare prin care să indice faptul că a apărut o eroare, motiv pentru care nu şi-a putut termina treaba cu succes.

O altă modalitate de a scrie un program care realizează aceeaşi operaţie cu cel de mai sus se prezintă în continuare:

```
1   /* FGETS.C */
2   #include<stdio.h>
3
4   int main(void)
5   {
6       FILE *pfluxin;
7       char buffer[80], numefis[80];
8
9       printf("Numele fisierului de afisat: ");
10      scanf("%79s", numefis);
11
12      if ((pfluxin = fopen(numefis,"r")) == NULL)
13      {
14              fprintf(stderr,"Fisierul %s nu a putut fi deschis pt. citire ", numefis);
15              perror("intrucat");
```

```
16          return 1;
17      }
18
19      while (fgets(buffer, sizeof(buffer), pfluxin) != NULL)
20          printf("%s", buffer);
21
22      fclose(pfluxin);
23
24      return 0;
25  }
```


Funcţia **fgets()** este perechea lui **gets()** citind un şir de pe un flux. Asemenea lui **gets()**, ea are nevoie de o zonă de RAM în care să stocheze şirul citit. În programul de mai sus o variabilă cu numele **buffer** se foloseşte în acest scop. **fgets()** are nevoie de trei argumente: un poantor la zona de RAM în care se va stoca şirul, lungimea zonei de RAM în care va fi stocat şirul şi un poantor la fişier. Dacă funcţia **gets()** întoarce un poantor cu valoarea **NULL** înseamnă că a apărut o eroare pentru că liniile din fişier s-au terminat. Altfel programul afişează, folosind **printf()**, conţinutul liniei citite.

12.4 Fişiere ASCII şi fişiere binare în C

Limbajul C tratează informaţia din fişiere în două moduri distincte, în funcţie de modul în care fişierul este deschis. În **modul ASCII**, dacă C găseşte în fişier un caracter <**Ctrl**>+<**Z**> (ASCII **26**), la citirea lui, acesta va fi interpretat ca terminator de fişier şi va presupune că s-a ajuns la sfârşitul fişierului. C nu scrie automat la sfârşitul fişierului ASCII acest caracter, dar el poate fi scris explicit "cu mâna". De asemenea, caracterul de trecere la linie nouă '**\n**' (**newline**) este stocat sub forma a două caractere ASCII **13 10** (**carriage return** şi **line feed**), dar în RAM este stocat numai în formatul intern C, sub forma terminatorului de linie, adică ASCII **10** (**line feed**). Atunci când şirul "**Salutare!\n**" a fost scris pe disc fişierul **salut.txt** avea 11 octeţi lungime întrucât a fost scris sub forma **S a l u t a r e ! 13 10** şi pentru că **modul implicit de scriere** pe disc este cel ASCII. În **modul binar** C nu interpretează conţinutul fişierului când scrie sau citeşte datele în el. Acesta va fi modul de lucru cu datele când, de exemplu, scriem un program pentru copierea fişierelor. Dacă şirul de mai sus va fi scris folosind modul binar, conţinutul pe disc al fişierului va fi **S a l u t a r e ! 10**, însă la folosirea lui **fopen()** trebuie specificat şi caracterul "**b**". Dacă de exemplu dorim să scriem un fişier binar în locul caracterului "**w**" folosim pe "**wb**".

Probleme care vin din MS-DOS

Când a fost inventat sistemul de operare MS-DOS, cineva a hotărât ca terminarea unui fişier să fie marcată de caracterul special <**Ctrl**>+<**Z**> a cărui valoare ASCII este **26**. Citirea unui octet cu valoarea **26** în modul ASCII devenea o problemă întrucât acesta putea marca terminarea fişierului. De exemplu **fgetc()** întoarce valoarea **EOF** şi citirea nu se poate continua. Acesta este motivul pentru care citirea fişierelor binare în modul ASCII va conduce la citirea unui număr mic de linii din cele totale pentru că la întâlnirea caracterului ASCII **26** citirea se opreşte. Pentru că MS-DOS a avut o oarecare influenţă asupra sistemelor de operare Windows 95, 98, NT şi OS/2 toate acestea au prezentat

aceeaşi problemă. Azi, caracterul <**Ctrl**>+<**Z**>, uneori scris şi ^**Z**, nu mai este stocat la capătul fişierelor pentru a marca terminarea acestora, întrucât au fost foarte multe probleme cu aplicaţii care nu reuşeau să scrie acest marcaj de terminare pe disc. Aceste fişiere neterminate puteau creşte peste anumite limite admise ocupând, în cel mai rău caz, tot spaţiul discului. Dacă în cazul fişierelor binare acest caracter nu mai marchează terminarea fişierelor, întrebarea care apare este cum se poate detecta terminarea unui fişier binar. Sistemele de operare moderne păstrează pe disc numărul de octeţi stocaţi într-un fişier, aceasta fiind informaţia folosită pentru detectarea terminării fişierului.

Probleme care vin din UNIX

Caracterul '**\n**' (**newline**) era folosit în UNIX pentru comanda terminalelor (UNIX avea cod foarte "dur" scris pentru comanda acestor terminale). Un terminal de tipul consolă sau imprimată avea un cap care atunci când ajungea la capătul unei linii trebuia să revină la începutul liniei. Această operaţie se numea **retur de car**, motiv pentru care caracterul ce comanda această acţiune s-a numit **carriage return**. Apoi, hârtia trebuia să avanseze cu o linie în sus, această operaţie consta în **avansul la o linie nouă**, caracterul care comanda această activitate numindu-se **newline**. În sistemul de operare UNIX aceste două operaţii se făceau la scrierea unui singur caracter şi anume '**\n**', tocmai pentru că aici codul corespunzător comenzii acestor terminale era mare. MS-DOS este o "idee" de sistem de operare. Datorită simplităţii lui s-a decis că **newline** nu trebuia să facă două operaţii, astfel că aici sunt necesare caracterele '**\r**' (**carriage return** - ASCII **13**) şi '**\n**' (**line feed** - ASCII **10**) pentru a obţine acelaşi efect asupra capului de scriere ca şi în UNIX cu '**\n**'.

12.4.1 Poziţionarea în fişierele binare

Există două mecanisme pentru poziţionare în fişiere. Metoda tradiţională este funcţia **fseek**() ce are descrierea:

fseek - mută poantorul de fişier într-o poziţie specificată

Prototip: **int fseek(FILE *flux, long offset, int origine);**
Fişier prototip: **fseek - <stdio.h>**
Valori întoarse: în caz de succes, **fseek**() întoarce **0**. Altfel, întoarce o valoare nenulă. Pentru dispozitive incapabile să efectueze căutarea poziţiei specificate, valoarea întoarsă este nedefinită.
Parametri: **flux** - poantor la structura **FILE**;
offset - număr de octeţi faţă de **origine**;
origine - poziţia iniţială.

Remarcă: **fseek**() mută poantorul de fişier asociat fluxului la o nouă poziţie specificată prin **offset** octeţi faţă de originea **origine**. Următoarea operaţie pe flux va avea loc din noua poziţie. Argumentul **origine** trebuie să fie una din următoarele constante definite în **STDIO.H**:

SEEK_CUR - poziţia curentă a poantorului de fişier;
SEEK_END - sfârşitul fişierului;
SEEK_SET - începutul fişierului.

Când un fişier este deschis pentru adăugare, poziţia curentă este determinată de ultima

operaţie de I/E. Dacă nu s-au efectuat operaţii de I/E asupra fişierului deschis pentru adăugare, poziţia poantorului de fişier este la începutul fişierului.

Pentru fluxuri deschise în modul text, **fseek()** are o utilitate limitată pentru că decodificarea caracterelor **carriage return** şi **linefeed** poate cauza ca **fseek()** să producă rezultate neaşteptate. Singurele poziţionări care sunt garantate pentru fluxuri deschise în mod text sunt: căutarea cu **offset 0** relativ la orice valoarea de origine **origine;** căutarea de la începutul fişierului cu **offset** întors de funcţia **ftell()**.

Problema fundamentală a funcţie **fseek()** este parametrul **offset** care fiind de tipul **long** poate lua valoarea maximă de 2^{31}-1, adică permite poziţionarea peste o porţiune de 2.1Gb a unui fişier. Pentru rezolvarea acestei probleme **standardul** defineşte alte două funcţii cu următoarele prototipuri:


```
inf fgetpos(FILE *flux, fpos_t *ptr);
int fsetpos(FILE *flux, const fpos_t *ptr);
```

Aici **fpos_t** este un tip dependent de implementare în stare să stocheze o poziţie dintr-un fişier de mărime arbitrară. Funcţia **fgetpos()** înregistrează poziţia curentă din **flux** în ***ptr** pentru ca să fie utilizată în **fsetpos()**. Funcţia **fsetpos()** realizează poziţionarea în **flux** la poziţia înregistrată prin **fgetpos()** în ***ptr**. Pentru exemplificarea lucrului cu aceste funcţii vor fi folosite încă două funcţii **fread()** şi **fwrite()** care au prototipurile:


```
size_t fread(void *ptr, size_t marime, size_t nrob, FILE, *flux);
size_t fwrite(const void *ptr, size_t marime, size_t nrob, FILE, *flux);
```

fread() citeşte de pe **flux** în tabloul poantat de **ptr** cel mult **nrob** de obiecte de dimensiunea **marime**. Întoarce numărul obiectelor citite care poate fi mai mic decât numărul cerut de noi. Determinarea stării după citire se face cu una din funcţiile **feof()** sau **ferror()**. **fwrite()** scrie din tabloul poantat de **ptr**, **nrob** de obiecte cu dimensiunea **marime** pe **flux**. Valoarea întoarsă este numărul obiectelor scrise care poate fi mai mic decât **nrrob** în caz de eroare.


```
1  /* FSETGET.C */
2  #include<stdio.h>
3
4  int main()
5  {
6          int nrcit,nrscr;
7          double d;
8          char tc[50];
9          fpos_t pozitie;
10         FILE *i, *e;
11
12         i=fopen("fisier.bin", "rb");
13         e=fopen("noufisier.bin", "wb");
14
15         nrcit=fread(&d, sizeof(d), 1, i);
```

```
16          printf("S-a citit %i obiect.\n",nrcit);
17
18          fgetpos(i, &pozitie);
19          nrcit=fread(tc, sizeof(char), 20, i);
20          printf("S-au citit %i obiecte.\n",nrcit);
21
22          fsetpos(i, &pozitie);
23          nrcit=fread(tc, sizeof(char), 60, i);
24          printf("S-au citit %i obiecte.\n",nrcit);
25          nrscr=fwrite(tc, sizeof (char), nrcit, e);
26          printf("S-au scris %i obiecte.\n",nrscr);
27
28          nrscr=fwrite(&d, sizeof (d), 1, e);
29          printf("S-a scris %i obiect.\n",nrscr);
30
31          fseek(i,0L,SEEK_SET);
32          nrcit=fread(&d, sizeof(char), 8, i);
33          printf("S-au citit %i obiecte.\n",nrcit);
34          nrscr=fwrite(&d, sizeof (d), 1, e);
35          printf("S-a scris %i obiect.\n",nrscr);
36
37          fclose(i);
38          fclose(e);
39
40          return (0);
41  }
```

Pentru a rula acest program aveţi nevoie de un fişier text numit **fisier.bin** care să conţină textul:

```
1.0123456789Acesta este un mic fisier de 57 de caractere.
```

Rzultatele afişate pe ecran sunt:

```
S-a citit 1 obiect.
S-au citit 20 obiecte.
S-au citit 49 obiecte.
S-au scris 49 obiecte.
S-a scris 1 obiect.
S-au citit 8 obiecte.
S-a scris 1 obiect.
```

iar pe disc va fi creat un fişier cu numele **noufisier.bin** având următorul conţinut:

```
6789Acesta este un mic fisier de 57 de caractere.1.0123451.012345
```

Programul este numai un exemplu de utilizare a acestor funcţii şi nu este unul coerent. În

liniile 12-13 sunt deschise cele două fluxuri în modul binar. Primul **fread()** citeşte **sizeof(d) = 8** octeţi de pe fluxul de intrare **i** în variabila **d**. Poziţia curentă din **flux** va fi salvată folosind funcţia **fgetpos()** în **pozitie**. Apoi se încearcă citirea în tabloul **tc** a **20 x 1** octeţi, apoi se revine la această poziţie şi se încearcă citirea a **60** de octeţi. Din rezultate se observă că numai **49** de obiecte (octeţi) au putut fi citite. În final se va folosi funcţia **fseek()** pentru a exemplifica revenirea la începutul fluxului **i**.

12.5 Lucrul cu înregistrări

Noţiunea de înregistrare (**record**) este familiară celor care au lucrat în Pascal sau au folosit sisteme pentru gestionarea bazelor de date. Să presupunem că avem de scris un program care să stocheze numele studenţilor, anul din care fac parte şi media lor generală. O înregistrare cu acest tip de date va fi de forma:

Nume: Vasile Cosmin
An: 1234/1
Medie generală: 9.78

Observaţi că există atât date ASCII cât şi date numerice în virgulă flotantă. În limbajul C o astfel de grupare de date poate fi pusă la un loc folosind o structură de forma:

```
struct student
{
        char nume[40];
        char grupa[10];
        float medie;
};
```

Vom scrie în continuare un program **SCRIE.C** care să stocheze mai mulţi studenţi pe disc, într-un fişier numit **stud.dat**.

```
1  /* SCRIE.C */
2  #include<stdio.h>
3  #include<conio.h>
4  #define NRMAX 100
5
6  struct student
7  {
8          char nume[40];
9          grupa[10];
10         float medie;
11 };
12
13 int main()
14 {
15         FILE *pies;
16         int c, nrstud;
17         struct student tstudenti[NRMAX];
```

```
        float x;

        c=0;
        printf("Introdu nr. de studenti care vor fi stocati: ");
        scanf("%i", &nrstud);
        fflush(stdin);

        do {
                printf("Nume student: ");
                gets(tstudenti[c].nume);
                fflush(stdin);

                printf("Grupa: ");
                scanf("%9s", &tstudenti[c].grupa);

                printf("Media generala: ");
                scanf("%f",&tstudenti[c].medie);
                fflush(stdin);
                ++c;
        }       while (c < nrstud);

        if ((pies = fopen("stud.dat", "w")) == NULL)
        {
                fprintf(stderr,"Fisierul stud.dat nu a putut fi deschis pt. scriere ");
                perror("intrucat");
                return 1;
        }
        fwrite(tstudenti, sizeof(struct student), nrstud, pies);
        fclose(pies);
        return 0;
}
```

Pentru a realiza un program care să poată citi datele scrise în fişierul **stud.dat** tot ceea ce trebuie să facem este să folosim **fread()** în loc de **fwrite()** şi să deschidem fişierul pentru citire în loc de scriere.

```
/* CITESTE.C */
#include<stdio.h>
#include<conio.h>
#define NRMAX 100

struct student
{
        char nume[40], grupa[10];
        float medie;
};
```

```
int main()
{
        FILE *pin;
        int c, nrstud;
        struct student tstudenti[NRMAX];

        printf("Introdu nr. de studenti care vor fi cititi: ");
        scanf("%i", &nrstud);
        fflush(stdin);
        if ((pin = fopen("stud.dat", "r")) == NULL)
        {
                fprintf(stderr,"Fisierul stud.dat nu a putut fi deschis pt. scriere");
                perror("intrucat");
                return 1;
        }
        fread(tstudenti, sizeof(struct student), nrstud, pin);
        fclose(pin);

        c=0;
        do {
                printf("Nume student: ");
                puts(tstudenti[c].nume);
                printf("Grupa: %9s\n", tstudenti[c].grupa);
                printf("Media generala: %f\n",tstudenti[c].medie);
                printf("-----------------------------------------------\n");
                ++c;
        } while (c < nrstud);

        return 0;
}
```


O altă variantă a acestui program însă care foloseşte spaţiu pentru stocarea în RAM numai pentru o singură înregistrare şi nu pentru un tablou de înregistrări, este:

```
/* CITESTE1.C */
#include<stdio.h>
#include<conio.h>

struct student
{
        char nume[40], grupa[10];
        float medie;
};

int main()
{
        FILE *pin;
```

```
14      int c, nrstud;
15      struct student unstudent;
16
17      printf("Introdu nr. de studenti care vor fi cititi: ");
18      scanf("%i", &nrstud);
19
20      if ((pin = fopen("stud.dat", "r")) == NULL)
21      {
22              fprintf(stderr,"Fisierul stud.dat nu a putut fi deschis pt.\ scriere ");
23              perror("intrucat");
24              return 1;
25      }
26
27      c=0;
28      do {
29              fseek(pin, (long) c*sizeof(struct student), 0);
30              fread(&unstudent, sizeof(struct student), 1, pin);
31              printf("Nume student: ");
32              puts(unstudent.nume);
33              printf("Grupa: %9s\n", unstudent.grupa);
34              printf("Media generala: %f\n",unstudent.medie);
35              printf("-----------------------------------------\n");
36              ++c;
37      }       while (c < nrstud);
38
39      fclose(pin);
40
41      return 0;
42  }
```

Programul foloseşte **fseek()** pentru poziţionare în cadrul fişierului. În **linia 29** poziţia noii înregistrări se găseşte prin deplasarea peste **c*sizeof(struct student)** octeţi în fişier, în raport cu începutul fişierului.

12.6 Citirea parametrilor din linia de comandă

Cei care aţi lucrat în sistemul de operare DOS cunoaşteţi comanda **copy *sursă destinaţie*** care permite copierea unui fişier de la o sursă, la o destinaţie. Programul care urmează implementează o variantă rudimentară a acestei comenzi. Utilizatorul trebuie să specifice două nume de fişiere, unul sursă - pe cine copiem - şi unul destinaţie - unde şi sub ce nume copiem. Modul de lucru cu fişierele este cel binar, iar după ce operaţia de copiere s-a derulat, funcţia **fclose()** este folosită pentru a închide fişierele de sursă şi destinaţie. În limbajul de programare C folosirea lui **fclose()** nu este obligatorie întrucât la terminarea programului, care are loc la întâlnirea lui **return()** din funcţia **main()**, fişierele deschise vor fi automat închise. Totuşi, închiderea unui fişier de care nu mai este nevoie trebuie să devină obişnuinţă pentru programator deoarece numărul maxim de fişiere care pot fi

deschise simultan este limitat de obicei de către sistemul de operare. Dacă un fişier deschis va fi închis când nu mai este nevoie de el, atunci un alt fişier va putea fi deschis fără a se depăşi limita maximă admisă de sistemul de operare. Pericolul de mai sus apare când uităm să închidem fişiere într-o funcţie care este apelată în mod repetat, în final mesajul de eroare primit la folosirea lui `fopen()` fiind "too many open files".


```
/* COPIE1.C */
#include<stdio.h>
#define BUFFER 32*1024
char buffer[BUFFER];

int main(void)
{
        char sursa[80], destinatie[80];
        FILE *flux_s, *flux_d;
        size_t nrcit;

        printf("Numele fisierului sursa: ");
        gets(sursa);
        if ((flux_s = fopen(sursa,"rb")) == NULL)
        {
                fprintf(stderr,"Fisierul %s nu a putut fi deschis pentru citire\n", sursa);
                perror("intrucat");
                return 1;
        }
        printf("Numele fisierului destinatie: ");
        gets(destinatie);
        if ((flux_d = fopen(destinatie,"w+b")) == NULL)
        {
                fprintf(stderr,"Fisierul %s nu a putut fi deschis pentru scriere\n",\
destinatie);
                perror("intrucat");
                return 1;
        }
        while (!feof(flux_s))
        {
                nrcit=fread(buffer, sizeof(buffer[0]), BUFFER, flux_s);
                if (ferror(flux_s))
                {
                        perror("\tEroare la citire" );
                        break;
                }
                fwrite(buffer, sizeof(buffer[0]), nrcit, flux_d);
                if (ferror(flux_d))
                {
                        perror("\tEroare la scriere" );
                        break;
```

```
        }
    }
        fclose(flux_s);
        fclose(flux_d);

        return 0;
}
```


Când programul de mai sus este lansat în execuţie, conversaţia cu utilizatorul se va derula sub forma:


```
Numele fisierului sursa: a.exe
Numele fisierului destinatie: b.exe
```

Cei care aţi folosit comanda DOS copy ştiţi că operaţia de mai sus se realizează prin scrierea **copy** ***a.exe b.exe*** la prompterul sistemului de operare. Fişierul ***a.exe*** din directorul curent va fi copiat sub numele de ***b.exe*** în acelaşi director. Limbajul C permite accesul la linia comandă într-o modalitate portabilă, astfel că programul de mai sus poate fi rescris pentru a citi parametrii introduşi după numele programului executabil, la fel ca în cazul comenzii **copy**. Până acum antetul folosit pentru funcţia `main()` a fost `int main(void)`. Pentru a se putea citi parametrii liniei comandă trebuie folosit următorul prototip:


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

Întrucât variabilele `argc` şi `argv` sunt numai parametri numele lor poate fi schimbat de noi când scriem corpul funcţiei `main()`, de exemplu sub forma:

```
int main(int nr_arg, char * linie_arg[ ])
{
    ...;
}
```

cu condiţia ca ordinea şi tipul lor să nu fie modificate. `argc` este un întreg care conţine numărul de cuvinte introduse de utilizator, iar `argv` este un tablou de poantori la şiruri de caractere care stochează cuvintele efectiv introduse de utilizator (mai precis o copie exactă a acestora).

```
/* COPIE2.C */
#include<stdio.h>
#include<string.h>
#define BUFFER 32*1024
char buffer[BUFFER];

int main(int argc, char *argv[ ])
{
        char sursa[80], destinatie[80];
```

```
        FILE *flux_s, *flux_d;
        size_t nrcit;
        int i;

        if (argc != 3)
        {
                fprintf(stderr,"Programul %s necesita 2 parametri\nfisierul sursa si pe\
cel destinatie\n", argv[0]);
                for(i=0; argv[i] != NULL; i++)
                        fprintf(stderr, "argv[%i] = %s\n", i, argv[i]);
                return 1;
        }

        strcpy(sursa, argv[1]);
        strcpy(destinatie, argv[2]);

        if ((flux_s = fopen(sursa,"rb")) == NULL)
        {
                fprintf(stderr,"Fisierul %s nu a putut fi deschis pentru citire\n", sursa);
                perror("intrucat");
                return 1;
        }

        if ((flux_d = fopen(destinatie,"w+b")) == NULL)
        {
                fprintf(stderr,"Fisierul %s nu a putut fi deschis pentru sriere\n",\
destinatie);
                perror("intrucat");
                return 1;
        }

        while (!feof(flux_s))
        {
                        nrcit=fread(buffer, sizeof(buffer[0]), BUFFER, flux_s);
                        if (ferror(flux_s))
                        {
                                perror("\tEroare la citire" );
                                break;
                        }
                        fwrite(buffer, sizeof(buffer[0]), nrcit, flux_d);
                        if (ferror(flux_d))
                        {
                                perror("\tEroare la scriere" );
                                break;
                        }
        }

        fclose(flux_s);
        fclose(flux_d);
```

```
    printf("Copierea a fost realizata cu succes.\n");
    return 0;
}
```

Dacă programul este lansat în execuţie prin linia de comandă **copie2** ***a.exe b.exe***, spaţiul de date pentru `argc` şi `argv` se reprezintă în *Figura 22*.

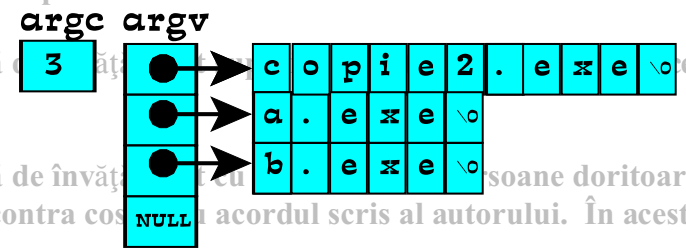


Figura 22

ANTAL Tiberiu Alexandru
tel.: 0040-264-530918
e-mail: antaltiberiu@pcnet.ro

Conf. dr. ing. ANTAL Tiberiu Alexandru
Universitatea Tenhică din Cluj-Napoca
Facultatea Construcţii de Maşini
Catedra de Mecanică şi Programare
Curs de limbaj C

Copyright 2001. Toate drepturile sunt rezervate autorului.

Multiplicarea acestui document în scop comercial este interzisă.

Sudenţii participanţi la orice formă de învăţământ superior bugetar pot multiplica acest document pentru uzul personal.

Sudenţii participanţi la orice formă de învăţământ cu plată sau alte persoane doritoare pot multiplica documentul numai contra cost şi cu acordul scris al autorului. În acest scop mă puteţi contactă la:

ANTAL Tiberiu Alexandru
tel.: 0040-264-530918
e-mail: antaltiberiu@pcnet.ro

Universitatea Tenhică din Cluj-Napoca
Facultatea Construcţii de Maşini
Catedra de Mecanică şi Programare
Curs de limbaj C

13 Alocarea dinamică a memoriei

Un program aflat în execuţie divizează porţiunea de memorie în care a fost încărcat conform desenului din *Figura 23*. Dimensiunea segmentelor de `cod` şi de `date` sunt fixate pe toată durata execuţiei programului. Dimensiunea segmentului de `stivă` creşte când se apelează funcţii, se transferă parametri şi se creează variabile locale, respectiv scade când se revine din funcţii, iar spaţiul pentru variabilele locale şi parametri este dezalocat. Segmentul de `heap`, sau heap-ul, este plasat în "opoziţie" cu cel de stivă în sensul că atunci când spaţiul alocat stivei creşte, cel disponibil pentru heap scade şi invers. Linia dintre heap şi stivă este punctată pentru a simboliza că cele două zone nu sunt delimitate static.

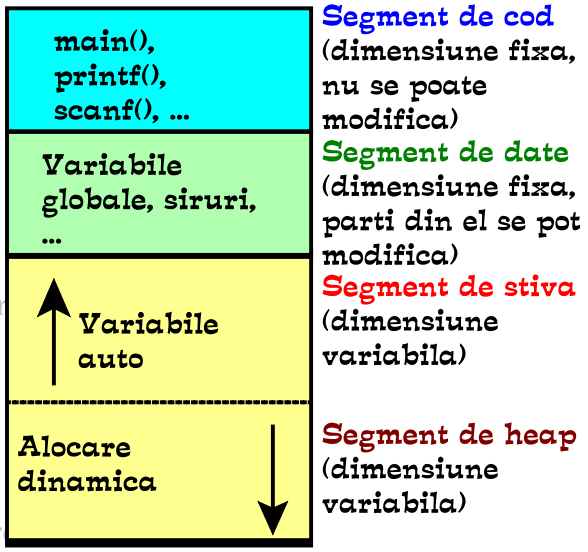


Figura 23

Heap-ul pune la dispoziţia programatorului un spaţiu de RAM pe care acesta îl poate aloca dinamic. Problema este mărimea acestui spaţiu. În cazul sistemelor de operare simple, cum este MS-DOS, programul executabil conţine stocat în el un număr care defineşte cantitatea totală de memorie necesară pentru ca programul să fie executat. Numărul acesta este spaţiul de RAM maxim care se alocă programului de către sistemul de operare. După încărcarea segmentelor de cod şi de date, ceea ce rămâne se împarte între stivă şi heap. Dacă segmentul de stivă creşte prea mult şi dă peste cel de heap programul "crapă".

Alocarea memoriei în cazul sistemelor de operare moderne

În cazul sistemelor de operare moderne programul este încărcat undeva în RAM şi lăsat să se execute. Dacă spaţiul în care s-a încărcat devine prea mic, de exemplu pentru că stiva şi heap-ul cresc prea mult, sistemul de operare găseşte o altă zonă de RAM mai mare în care va copia segmentele de cod şi de date. În noua zonă, stiva şi heap-ul vor fi plasate cât mai departe, apoi din nou programul este lăsat să se execute. Dacă stiva şi heap-ul cresc prea mult se încearcă din nou găsirea unei zone şi mai mari de

RAM, iar numărul de repetări ale procedurii este practic influenţat numai de limitele memoriei fizice a calculatorului. Dacă sistemul de operare ştie să lucreze cu memorie virtuală, limitele memoriei fizice sunt depăşite în căutarea spaţiului necesar executării programului. În cazul sistemelor de operare "pe 32 de biţi" cum sunt UNIX, Windows NT şi 95/98, limita este dată de 2^{32} octeţi, ceea ce corespunde la aproximativ 2 GB de RAM. În realitate, pentru că majoritatea sistemelor de operare trebuie să rămână rezidente în RAM, cantitatea de memorie disponibilă este ceva mai mică. În cazul sistemelor de operare "pe 64 de biţi", de exemplu Winows NT pe un procesor DEC Alpha, limita creşte la 2^{64} biţi care ar fi un număr de ordinul TB (Tera octeţi). Ultima limită a spaţiului folosit pentru stocarea programelor în execuţie este hard disc-ul. Memoria virtuală folosită de un sistem de operare trebuie salvată undeva dar întrucât majoritatea calculatoarelor nu dispun de foarte multă memorie RAM, următorul dispozitiv de stocare a datelor care nu mai încap în RAM este hard disc-ul.

13.1 Funcţii pentru alocarea dinamică a memoriei

malloc - alocă un bloc de memorie pe heap

Prototip: `void *malloc(size_t dimensiune);`

Fişier prototip: `malloc()` - `<stdlib.h>` şi `<malloc.h>`

Valori întoarse: `malloc()` întoarce un poantor de tipul `void` la spaţiul alocat sau `NULL` dacă spaţiul liber existent pe heap este insuficient pentru a realiza alocarea. Dacă se întoarce un poantor de tipul diferit de `void`, trebuie folosit operatorul de conversie forţată a tipului pentru valoarea întoarsă. Dacă dimensiune este zero, `malloc()` alocă spaţiu pentru un obiect de lungime `0` octeţi şi întoarce un poantor valid la acesta.

Parametri: `dimensiune` - numărul de octeţi alocaţi pe heap.

Remarcă: Spaţiul alocat de `malloc()` pe heap are cel puţin mărimea lui `dimensiune` în octeţi. Blocul poate fi mai mare din cauza stocării şi a unor informaţii legate de gestionarea heap-ului şi a operaţiei de aliniere. Codul care lansează în execuţie programul utilizatorului foloseşte `malloc()` pentru alocarea spaţiului pentru variabilele `environ`, `envp` şi `argv`.

calloc - alocă un tablou pe heap cu elementele iniţializate cu 0

Prototip: `void *calloc(size_t număr, size_t dimensiune);`

Fişier prototip: `calloc()` - `<stdlib.h>` şi `<malloc.h>`

Valori întoarse: `calloc()` întoarce un poantor la un spaţiu suficient de mare pentru stocarea unui număr de obiecte de dimensiunea specificată prin `dimensiune` pe heap. Fiecare obiect este iniţializat cu valoarea zero.

Parametri: `număr` - numărul de obiecte;

`dimensiune` - mărimea, în octeţi, a unui obiect alocat pe heap.

realloc - realocă blocurile de memorie

Prototip: `void *realloc(void *ptrbloc, size_t dimensiune);`

Fişier prototip: `calloc()` - `<stdlib.h>` şi `<malloc.h>`

Valori întoarse: `realloc()` întoarce un poantor `void` la blocul de memorie realocat. Valoarea întoarsă este `NULL` dacă `dimensiune` este zero şi `ptrbloc` nu este `NULL` sau dacă nu este suficientă memorie disponibilă pentru a expanda spaţiul blocului la cel specificat în `dimensiune`. În primul caz spaţiul originar alocat blocului este eliberat, iar în cel de al doilea blocul originar rămâne nemodificat.

Parametri: `ptrbloc` - poantor la spaţiul originar alocat blocului de memorie;

`dimensiune` - noua dimensiune a blocului de memorie în octeţi.

free - dezalocă (eliberează) spaţiul alocat unui bloc de memorie

Prototip: `void free(void *ptrbloc);`

Fisier prototip: `free()` - `<stdlib.h>` şi `<malloc.h>`

Valori întoarse: nu are

Parametri: `ptrbloc` - poantor la un bloc de memorie alocat deja cu `malloc()`, `calloc()` sau `realloc()` care va fi eliberat (dezalocat).

Remarcă: după dezalocarea unui bloc de memorie spaţiul disponibil pe heap creşte.

13.2 Alocarea dinamică a memoriei pentru vectori

În limbajul C tablourile sunt implementate într-o formă destul de primitivă. Nu există un mecanism al limbajului pentru modificarea numărului de elemente sau de dimensiuni ale unui tablou după declararea acestuia (spre deosebire de limbajul BASIC care are instrucţiunea `ReDim` numai pentru acest scop). Este însă posibil ca un spaţiu continuu de adrese fizice să fie alocat unui tablou pe heap. Condiţia de continuitate a tabloului este garantată însăşi prin modul de lucru al funcţiilor de manipulare a heap-ului. În acest scop programul trebuie să declare un poantor la adresa de început a blocului de memorie alocat pe heap. De exemplu, în locul unui tablou declarat prin `float a[100];` putem declara un poantor la tipul `float` (`float *pa;`), apoi folosind funcţia `malloc()` alocăm spaţiul dinamic pe heap tabloului cu expresia `pa = malloc(100* sizeof(float));`. Spaţiul de memorie în acest caz va fi alocat pe heap şi nu pe segmentul de date ca şi în primul caz. Un element al tabloului `pa` va putea fi referit prin expresia `pa[i]`. Dacă se alocă mai multe tablouri pe heap este nevoie pentru fiecare tablou de câte un poantor de tipul elementelor de tablou. Este posibilă şi folosirea lui `malloc()` pentru alocarea pe heap a spaţiului folosit pentru stocarea unei singure variabile. Aceasta este cea mai simplă modalitate de folosire a funcţiei `malloc()`, iar programul care urmează ilustrează modul de folosire a funcţie `malloc()` în acest scop:

```
/* MALLOC1.C */
#include <malloc.h>
#include <stdio.h>
```

```
int main(void)
{
        double *pf;
        pf= (double *) malloc(sizeof(double));
        *pf=123.456789;
        printf("Continutul lui pf: %p\n",pf);
        printf("Valoarea la care poanteaza: %lf\n",*pf);
        free(pf);

        return 0;
}
```

Rezultate:

```
Continutul lui pf: 00780EB0
Valoarea la care poanteaza: 123.456789
```

În programul de mai sus poantorului **pf** la tipul **double** i se atribuie valoarea întoarsă de funcţia **malloc()** - declarată în **malloc.h**. Apoi lui ***pf** i se atribuie valoarea **123.456789**, stocând această valoare la adresa la care poantează **pf**. Linia **(double *) malloc(sizeof (double))** face următoarele:

- expresia **malloc(sizeof (double))** întoarce numărul de octeţi utilizaţi pentru stocarea unui tip **double**, adică valoarea **8**;

- funcţia **malloc(dimensiune)** alocă un număr de **dimensiune** octeţi consecutivi pe heap-ul neutilizat şi întoarce adresa de început a blocului de memorie;

- expresia **(double *)** forţează tratarea adresei de început ca un poantor de tipul **double** (această operaţie am spus deja că poartă denumirea de forţare a tipului - **casting**). Unele compilatoare fac această operaţie automat, dar majoritatea compilatoarelor C vor genera mesajul de eroare **Non-portable pointer assignement** dacă conversia este omisă;

- în final, adresa este stocată în **pf**. Aceasta înseamnă că s-a creat dinamic o variabilă reală de tipul **double** pe care o putem referi prin ***pf**.

Dacă doriţi să alocaţi dinamic spaţiul pentru elementele unui vector, programul următor ilustrează modul de lucru cu funcţia **calloc()**.

```
1  /* CALLOC1.C */
2  #include <malloc.h>
3  #include <stdio.h>
4  int main(void)
5  {
6          int *Vector,i,n;
7
8          printf("Nr. elementelor de vector = ");
9          scanf("%i",&n);
10
11         if ((Vector=(int *) calloc(n,sizeof(int))) == NULL)
12         {
13                 fprintf(stderr,"Nu se poate aloca spatiul de %u octeti"\
```

```
14              "\npentru %u intregi pe stiva\n", n*sizeof(int), n);
15          return 1;
16      }
17
18      for (i=0; i<n;i++)
19      {
20          printf("Vector + %2i = ", i);
21          scanf("%i", Vector+i);
22      }
23      printf("Lista adreselor:");
24      for (i=0; i<n;i++)
25          printf("%8p ",(Vector + i));
26      printf("\nLista valorilor: ");
27      for (i=0; i<n;i++)
28          printf("%8d ",*(Vector + i));
29      printf("\n");
30
31      free(Vector);
32
33      return 0;
34  }
```

Rezultate:

```
Nr. elementelor de vector = 3
Vector + 0 = 123
Vector + 1 = 34553
Vector + 2 = 2323
Lista adreselor: 00780EB0 00780EB4 00780EB8
Lista valorilor:      123     34553      2323
```

Programul permite manipularea valorilor de tipul **int** în condiţiile de necunoaştere a numărului acestora în momentul compilării programului. În **linia 11** se încearcă alocarea spaţiului pentru un număr de **n** elemente de tipul **int**. Când **calloc()** întoarce valoarea **NULL** înseamnă că nu există suficient spaţiu pe heap pentru alocarea blocului cerut. Elementele vectorului la care poantează **Vector** sunt referite prin sintaxa ***(Vector + i)**. În locul acesteia s-ar fi putut scrie şi **Vector[i]**. În **linia 34** se eliberează spaţiul alocat pe heap pentru bloc.

Ce se petrece dacă la un moment dat ne dăm seama că avem nevoie de mai multe elemente decât cele citite în variabila **n**? Funcţia **realloc()** permite modificarea cantităţii de memorie alocate unui bloc. Să presupunem că în locul celor **n** elemente dorim să realocăm spaţiu pentru **2*n** elemente de tipul **int**. Va fi nevoie de un nou poantor. În exemplul care urmează poantorul se numeşte **ReVector**, iar spaţiul pentru stocarea tabloului va fi realocat folosind expresia: **ReVector = realloc(Vector, 2*n*sizeof(int));**.

Şi în acest caz este posibil ca realocarea să nu reuşească, adică să nu se găsească o zonă continuă de memorie dinamică cu dimensiunea cerută. În acest caz `realloc()` va întoarce valoarea `NULL` care va fi atribuită lui `ReVector`. Dacă adresa obţinută din `realloc()` ar fi fost atribuită direct lui `Vector`, în cazul nereuşitei alocării, adresa iniţială ar fi fost pierdută pentru că valoarea `NULL` ar fi fost pusă în locul celei iniţiale (datorită operatorului de atribuire). Dacă alocarea reuşeşte, următoarele două scenarii sunt posibile:

1. `realloc()` a fost în stare să crească regiunea de memorie iniţial alocată blocului, motiv pentru care adresa lui `ReVector` este aceeaşi cu cea a lui `Vector` *sau*
2. `realloc()` nu a putut creşte mărimea blocului alocat iniţial şi a trebuit să găsească un bloc nou. Adresa lui `ReVector` şi cea a lui `Vector` vor fi diferite în acest caz. Blocul iniţial va fi copiat la noua adresă şi în continuarea lui se va aloca spaţiul pentru obiectele noi.

Ce se petrece în cazul tablourilor cu mai multe dimensiuni şi cum anume se va aloca dinamic spaţiul pentru acestea? Să presupunem că lucrăm cu un tablou `float tempmedzil[10][365]` care va stoca temperatura medie pentru fiecare zi din an a `10` regiuni distincte. Dacă am dori să alocăm dinamic spaţiul pentru acest tablou, permiţând ca numărul de regiuni să fie variabil şi numai partea de `365` de `float`-uri să fie fixă, codul ar putea fi scris astfel:

```
float **pptmpmedzil;
**pptmpmedzil = calloc(nrregiuni, 365* sizeof(float));
```


unde `nrregiuni` va conţine numărul de regiuni în care se va măsura temperatura medie. Acest stil de declarare nu oferă însă suficiente informaţii pentru ca poantorul `pptmpmedzil` să permită deplasarea corectă prin tablou. De exemplu, când scriem `ptmpmedzil[3][100]`, `3` trebuie să producă saltul peste `4` blocuri de `365` de `r`-uri (`3*365*4 = 5696` octeţi) şi apoi `100` peste încă `4*100 = 400` de octeţi. În total trebuie să se sară `6096` octeţi, însă compilatorul nu poate determina acest lucru folosind declaraţia poantorului.


```
/* PTTAB.C */
#include<stdio.h>
```



```
int main(void)
{
        float tmpmedzil[3][10],**pptmpmedzil;
        tmpmedzil[0][0]=3.;
        pptmpmedzil=(float **)tmpmedzil;
```



```
        printf("  tmpmedzil      %p\n", tmpmedzil);
        printf(" pptmpmedzil      %p\n", pptmpmedzil);
        printf("  tmpmedzil[0][0] %f\n", tmpmedzil[0][0]);
        printf("  &tmpmedzil[0][0] %p\n", &tmpmedzil[0][0]);
        printf("&pptmpmedzil[0][0] %p\n", &pptmpmedzil[0][0]);

        return 0;
}
```

Rezultate:

```
   tmpmedzil       0065FD80
 pptmpmedzil       0065FD80
   tmpmedzil[0][0] 3.000000
  &tmpmedzil[0][0] 0065FD80
&pptmpmedzil[0][0] 40400000
```


În programul de mai sus se poate observa uşor că **tmpmedzil[0][0]** şi **pptmpmedzil[0][0]** sunt două obiecte diferite pentru că au adrese diferite, deşi **tmpmedzil** şi **pptmpmedzil** poantează la aceeaşi adresă. Cum se poate rezolva această problemă? Soluţia stă în declararea poantorului sub forma:


```
float (*ptmpmedzil)[365];
```

Compilatorul "va şti" că în cazul expresiei **ptmpmedzil[3][100]**, **3** trebuie înmulţit cu dimensiunea a **365** de **float**-uri (**pptmpmedzil** fiind poantor la un tablou de **365** de **float**-uri **3** duce la saltul peste **3** astfel de tablouri). De asemenea, "ştie" că **100** trebuie scalat prin dimensiunea lui **float**, astfel că adresele folosite pentru poziţionarea în tablou vor fi calculate corect.

ANTAL Tiberiu Alexandru
tel.: 0040-264-530918
e-mail: antaltiberiu@pcnet.ro

13.3 Alocarea dinamică a memoriei pentru matrice

După cum aţi dedus din cele citite până acum, când se alocă dinamic spaţiul pentru un tablou unidimensional (vector), dimensiunea acestuia poate fi găsită în timpul execuţiei programului. La utilizarea unor tablouri cu mai multe dimensiuni, nu trebuie să cunoaştem prima dimensiune în momentul compilării. Necesitatea cunoaşterii celorlalte dimensiuni depinde numai de felul în care vom scrie noi codul. În continuare se vor discuta problemele alocării dinamice a spaţiului pentru tablourile de întregi cu 2 dimensiuni (matrice de întregi). Pentru început, considerăm cazul în care se cunoaşte a 2-a dimensiune în momentul compilării programului.

Metoda 1: Una dintre metodele de a trata problema este utilizarea lui **typedef**. Pentru alocarea spaţiului unui tablou de două dimensiuni reaminţiţi-vă notaţiile echivalente:


```
iTab2[rand][coloana] = 1;     *(*(iTab2 + rand) + coloana) = 1;
```

este de asemenea adevărat că următoarele notaţii generează acelaşi cod:

ANTAL Tiberiu Alexandru
tel.: 0040-264-530918
e-mail: antaltiberiu@pcnet.ro

```
iTab2[rand]                   *(iTab2 + rand)
```

deci **iTab2** poate fi privit ca un tablou de tablouri, iar **iTab2[n]** ca un poantor la cel de al **n**-elea tablou din tabloul de tablouri. Fie programul:

```
#include <stdio.h>
#include<stdlib.h>
#define COLOANE 5

typedef int tip_TabRanduri[COLOANE];

tip_TabRanduri* prand;

int main(void)
{
        int iNrRanduri = 10;
        int iRand, iColoana;
        prand=malloc(iNrRanduri* COLOANE * sizeof(int));
        for (iRand; iRand < iNrRanduri; iRand++)
        {
                for (iColoana = 0; iColoana < COLOANE; iColoana++)
                {
                        prand[iRand][iColoana]=22;
                }
        }
        return 0;
}
```


Am presupus că utilizaţi un compilator ANSI care face automat conversia de la poantorul **void** întors de **malloc()**. Dacă utilizaţi un compilator antic, trebuie să faceţi conversia cu mâna, scriind **prand= (tip_TabRanduri *) malloc (...)** . Utilizarea acestor declaraţii face ca **prand** să aibă toate caracteristicile unui nume de tablou (cu excepţia că **prand** poate fi modificat) şi notaţia de tablou poate fi folosită în continuarea programului. Aceasta înseamnă că dacă doriţi să scrieţi o funcţie care să modifice conţinutul tabloului, **COLOANE** trebuie să fie unul din parametrii transferaţi funcţiei, după cum am discutat la tablouri şi funcţii.

Metoda 2: În metoda anterioară **prand** este un poantor la un tip de "tablou unidimensional de număr de **COLOANE** de întregi". Există o scriere sintactică mai simplă pentru a exprima acest lucru, fără a utiliza **typedef**:


```
int (*p1rand)[COLOANE];
```


Aici variabila **p1rand** are aceleaşi caracteristici cu **prand** şi reprezintă un poantor la un tablou de întregi, iar dimensiunea tabloului se dă prin **COLOANE**. Plasarea parantezelor face ca notaţia de poantor să fie predominantă întrucât fără acestea notaţia de tablou are precedenţă mai mare. Dacă scriam:

```
int *p1rand[COLOANE];
```

p1rand ar fi fost un tablou de poantori cu numărul de poantori la întregi egal cu **COLOANE**.

Metoda 3: Considerăm cazul în care nu se cunosc numărul de elemente la momentul compilării. O metodă ar fi crearea unui tablou de poantori la tipul întreg şi apoi alocarea spaţiului pentru fiecare rând în parte şi stocarea adresei rândului alocat în poantorul corespunzător din tablou.


```
/* ALDINTA1.C */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   int iNrRanduri = 5; /* Numarul de linii cat si cel de coloane se evalueaza */
   int iNrColoane = 10; /* sau se citeste in timpul executiei programului */
   int iRand;
   int **pRand;
   pRand = malloc(iNrRanduri * sizeof(int *));
   if (pRand == NULL)
   {
      puts("\nNu pot aloca spatiul pentru iRand poantori.\n");
      exit(0);
   }

   printf("\n\n\nIndice  Poantor(hexa)  Poantor(zec)  Diferenta(zec)");

   for (iRand = 0; iRand < iNrRanduri; iRand++)
   {
      pRand[iRand] = malloc(iNrColoane * sizeof(int));
      if (pRand[iRand] == NULL)
      {
         printf("\nNu pot aloca spatiul pentru Rand[%d]\n",iRand);
         exit(0);
      }
      printf("\n%d      %p      %d", iRand, pRand[iRand], pRand[iRand]);
      if (iRand > 0)
      printf("      %d",(int)(pRand[iRand] - pRand[iRand-1]));
   }

   return 0;
}
```

Rezultate:

```
Indice  Poantor(hexa)   Poantor(zec)    Diferenta(zec)
0       00770710        7800592
1       00770760        7800672         20
2       007707B0        7800752         20
3       00770800        7800832         20
4       00770850        7800912         20
```

În codul anterior **pRand** este un poantor la un poantor la tipul **int**. Pentru cazul nostru, poantează la primul element al unui tablou de poantori la tipul **int**. Dacă lucraţi cu această metodă accesul la un element de tablou se face tot în stilul clasic, adică **pRand[rand][coloana]= 22**, dar aceasta nu înseamnă că datele din tabloul cu 2 dimensiuni sunt stocate continuu în memorie.

Numărul de apeluri al funcţiei **malloc()** este:

Pentru tabloul de poantori	1
Pentru alocarea spaţiului fiecărui rând	5
Total	6

Dacă doriţi ca un bloc continuu de memorie să fie alocat pentru tablou, puteţi folosi metoda următoare.

Metoda 4: Aici se alocă un bloc continuu de memorie pentru stocarea tabloului. Apoi se creează un tablou de poantori care vor poanta la fiecare rând. Deşi se utilizează un tablou de poantori, tabloul actual este continuu în memorie.

```
/* ALDINTA2.C */
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
   int **pRand;
   int *piA;
   int *ptest;
   int k;
   int iNrRanduri = 5;/* Ambele dimensiuni pot fi evaluate sau pot fi*/
   int iNrColoane = 8;/* citite in timpul executiei programului */
   int iRand, iColoana;
   /* Aici se aloca spatiul pentru tablou */

   piA = malloc(iNrRanduri * iNrColoane * sizeof(int));
   if (piA == NULL)
   {
      puts("\nNu pot aloca spatiul pentru tablou");
      exit(0);
   }
   /* Aici se aloca spatiu pentru poantorii la randuri */

   pRand = malloc(iNrRanduri * sizeof(int *));
   if (pRand == NULL)
   {
```

```
        puts("\nNu pot aloca spatiul pentru poantori");
        exit(0);
    }

    /* Aici se va poanta la poantori */
```



```
    for (k = 0; k < iNrRanduri; k++)
    {
        pRand[k] = piA + (k * iNrColoane);
    }
```



```
    /* Aici se ilustreaza modul in care poantorii la randuri se incrementeaza */
    printf("\n\nIlustrarea modului in care poantorii la randuri se incrementeaza");
    printf("\n\nIndex    Poantor(hex)  Diferenta(zec)");
    for (iRand = 0; iRand < iNrRanduri; iRand++)
    {
        printf("\n%d      %p", iRand, pRand[iRand]);
        if (iRand > 0)
        printf("      %d",(pRand[iRand] - pRand[iRand-1]));
    }
    printf("\n\nAcum se tipareste tabloul\n");
    for (iRand = 0; iRand < iNrRanduri; iRand++)
    {
        for (iColoana = 0; iColoana < iNrColoane; iColoana++)
        {
            pRand[iRand][iColoana] = iRand + iColoana;
            printf("%d ", pRand[iRand][iColoana]);
        }
        putchar('\n');
    }
```



```
    puts("\n");
    /* Aici se ilustreaza faptul ca se utilizeaza un tablou
       bidimesional intr-un bloc continuu de memorie. */
    printf("Acum demonstram continuitatea in memorie\n");

    ptest = piA;
    for (iRand = 0; iRand < iNrRanduri; iRand++)
    {
        for (iColoana = 0; iColoana < iNrColoane; iColoana++)
        {
            printf("%d ", *(ptest++));
        }
        putchar('\n');
    }

    return 0;
}
```

Rezultate:

```
Ilustrarea modului in care poantorii la randuri se incrementeaza

Index   Poantor(hex)    Diferenta(zec)
0       007706D0
1       007706F0        8
2       00770710        8
3       00770730        8
4       00770750        8

Acum se tipareste tabloul
0 1 2 3 4 5 6 7
1 2 3 4 5 6 7 8
2 3 4 5 6 7 8 9
3 4 5 6 7 8 9 10
4 5 6 7 8 9 10 11
Acum demonstram continuitatea in memorie
0 1 2 3 4 5 6 7
1 2 3 4 5 6 7 8
2 3 4 5 6 7 8 9
3 4 5 6 7 8 9 10
4 5 6 7 8 9 10 11
```


Numărul de apeluri ale funcţie **malloc()** este:

Pentru tabloul în sine	1
Pentru tabloul de poantori	1
Total	2

Sigur v-aţi întrebat de ce este importantă continuitatea în memorie a tabloului. Un motiv ar fi acela că prin **memset()** se poate face iniţializarea lui în mod elegant. Un alt motiv ar fi că sistemul de operare implementează **malloc()** sub forma unei liste înlănţuite care conţine date despre mărimea fiecărui bloc. Deci fiecare apel a lui **malloc()** va duce la alocarea de spaţiu în lista înlăţuită descrisă mai sus. Cum în acest caz numărul de apeluri a lui **malloc()** este mai mic, spaţiul auxiliar consumat pentru stocarea poziţiei şi dimensiunii blocurilor de memorie alocate de această implementare este mai mic decât în cazul anterior.

13.4 Structuri dinamice de date

Alocarea dinamică a memoriei se poate face şi pentru structuri de date diferite de tablou. De exemplu, **struct** este des folosită pentru structuri de date care au elementele legate între ele cum sunt listele, arborii, grafele etc. unde numărul de elemente (noduri) necesare pentru stocarea datelor nu poate fi cunoscut exact la momentul compilării programului.

În *Figura 24* se reprezintă schema unei structuri de dată numită **listă simplu înlanțuită.**

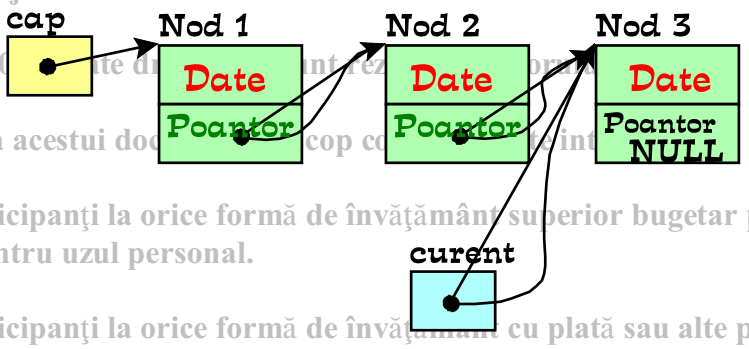


Figura 24 - Schema unei liste simplu înlănţuite

Datele sunt stocate într-o structură, care pe lângă informaţia utilă are şi un poantor la următorul nod. Oricând i se poate adăuga un nou nod listei dacă ultimul poantor este actualizat pentru a poanta la noul element adăugat. Dacă se pleacă de la începutul listei, aceasta se va crea din aproape în aproape, adică nod cu nod, prin folosirea poantorului la nodul următor. Ultimul poantor este de obicei iniţializat cu valoare **NULL** pentru a marca faptul că aici lista se termină. Un alt exemplu de listă este cea dublu înlănţuită. Aceasta are doi poantori, unul la nodul următor şi unul la nodul anterior, în acest fel lista putând fi parcursă în două sensuri, spre deosebire de cea simplu înlanţuită care poate fi parcursă numai într-un singur sens. Probabil cel mai bun exemplu de listă înlanţuită este Tabela de Alocare a Fişierele (**FAT** sau **File Allocation Table**) folosită de sistemul de operare DOS. Aceasta este o listă de cluster-e (cluster - spaţiu pe disc cu mărimea minimă alocată pentru stocare de date) alocată fişierelor stocate pe disc. Din moment ce dimensiunea unui fişier poate creşte, listele înlănţuite sunt folosite în scopul stocării lor. Fiecare intrare în FAT reprezintă un cluster pe disc şi stochează poziţia următoarei intrări. Pentru a găsi care clustere sunt folosite pentru stocarea unui fişier se parcurge FAT-ul până când se ajunge la un marcaj de terminare a listei de clustere.

Exemplul care urmează prezintă un grup de funcţii care alocă spaţiu pentru un nod al unei liste simplu înlănţuite, afişează datele stocate în aceasta prin parcurgerea ei de la cap la coadă şi în final eliberează spaţiul alocat pe heap nodurilor listei. În paragraful 10.7 noţiunea de listă a fost deja prezentată, noutatea programului care va fi prezentat constă în alocarea dinamică a memoriei pentru elementele listei (pentru a mă apropia de terminologia specifică listelor, aici elementele au fost numite noduri).


```
/* LISTASI.C */
#include<stdio.h>
#include<string.h>
#include<malloc.h>
#include<stdlib.h>
struct Nod
{
        int cod;
        struct Nod *p_urm_Nod;
};
```

```
struct Nod* nod_Nou(int c)
{
        struct Nod *p;
        if ((p = malloc(sizeof(struct Nod))) == NULL)
        {
                fprintf(stderr, "S-a epuizat memoria dinamica disponibila!\n");
                exit(9);
        }

        p->cod=c;
        p->p_urm_Nod=NULL;

        return p;
}
struct Nod* nod_Inserare(struct Nod *curent, int c)
{
        struct Nod* NN;

        NN=nod_Nou(c);
        curent->p_urm_Nod=NN;

        return NN;
}

void lista_Afisare(const struct Nod *curent)
{
        printf("%i ", curent->cod);
}
void lista_Parcurgere(const struct Nod *p)
{
        struct Nod *curent;

        curent=(struct Nod *) p;
        while (curent != NULL)
        {
                lista_Afisare(curent);
                curent=curent->p_urm_Nod;
        }
}

void lista_Eliberare(struct Nod *p)
{
        struct Nod *curent;

        printf("\nEliberarea nodurilor:");

        curent = p;
        while (curent != NULL)
```

```
        {
                curent=curent->p_urm_Nod;
                lista_Afisare(p);
                free(p);
                p=curent;
        }
}
int citeste_Nod()
{
        int n;

        printf("n = ");
        scanf("%i", &n);
        return n;
}
int main(void)
{
        int n;
        struct Nod *cap, *curent;

        if ((n = citeste_Nod()) <= 0)
        {
                printf("Am terminat!\n");
                exit(0);
        }
        curent=nod_Nou(n);
        cap=curent;
        while (n > 0)
        {
                if ((n = citeste_Nod()) > 0)
                        curent=nod_Inserare(curent,n);
        }

        lista_Parcurgere(cap);
        lista_Eliberare(cap);

        return 0;
}
```


Rezultate:

```
n = 1
n = 2
n = 3
n = 4
```

```
n = 5
n = 6
n = 7
n = 8
n = 0
1 2 3 4 5 6 7 8
Eliberarea nodurilor:1 2 3 4 5 6 7 8
```


Conf. dr. ing. ANTAL Tiberiu Alexandru
Universitatea Tenhică din Cluj-Napoca
Facultatea Construcţii de Maşini
Catedra de Mecanică şi Programare
Curs de limbaj C

Copyright 2001. Toate drepturile sunt rezervate autorului.

Multiplicarea acestui document în scop comercial este interzisă.

Sudenţii participanţi la orice formă de învăţământ superior bugetar pot multiplica acest document pentru uzul personal.

Sudenţii participanţi la orice formă de învăţământ cu plată sau alte persoane doritoare pot multiplica documentul numai contra cost şi cu acordul scris al autorului. În acest scop mă puteţi contactă la:

ANTAL Tiberiu Alexandru
tel.: 0040-264-530918
e-mail: antaltiberiu@pcnet.ro

14 Biblioteca C Standard

Capitolul care urmează este o prezentare sumară a funcţiilor din biblioteca C ANSI. Biblioteca standard **nu este** o parte a limbajului C, dar toate mediile de programare C asigură o metodă de lucru cu fişierele antet descrise în continuare şi o bibliotecă standardizată corespunzătoare acestor declaraţii. Declaraţiile de funcţii, tipuri de date şi macrodefiniţii sunt stocate în fişiere denumite **`antete standard`**. Acestea sunt prezentate în continuare, împreună cu o sumară descriere.

Diagnostic: <**`assert.h`**>

Testare tip caractere: <**`ctype.h`**>:

Numere de eroare: <**`errno.h`**>

Limite definite prin implementare pentru virgulă flotantă: <**`float.h`**>

Limite definite prin implementare: <**`limits.h`**>

<**`locale.h`**>

Funcţii matematice: <**`math.h`**>

Salturi nelocale: <**`setjmp.h`**>

Semnale: <**`signal.h`**>

Liste cu număr variabil de argumente: <**`stdarg.h`**>

<**`stddef.h`**>

Intrare şi ieşire: <**`stdio.h`**>

Funcţii utilitare: <**`stdlib.h`**>

Funcţii pentru prelucrarea şirurilor: <**`string.h`**>

Funcţii pentru oră şi dată: <**`time.h`**>

Includerea acestor antete în programul sursă se face prin directiva **`#include`** discutată deja în capitolul 7.

14.1 <assert.h>

`void assert(int expression);` macro folosit pentru realizarea diagnosticării unui program. Dacă expresia **`expression`** este **`falsă`** (ia valoarea numerică 0) un mesaj va fi afişat pe **`stderr`** şi apoi se apelează funcţia **`abort()`** pentru terminarea execuţiei programului. Fişierul

sursă şi numărul de linie a macrodefiniţiei se afişează prin `__FILE__`, `__LINE__` a preprocesorului. Dacă `NDEBUG` este definită acolo unde s-a inclus `<assert.h>`, macro `assert()` este ignorat.

14.2 <ctype.h>

Antetul conţine declaraţiile funcţiilor pentru testarea caracterelor. Conform standardului, submulţimea literelor implicite cuprinde 26 de caractere mici şi 26 de caractere mari ale alfabetului latin. Acestea pot varia în funcţie de anumite setări locale specifice, sub controlul funcţiei `setlocale()`.

ANTAL Tiberiu Alexandru
tel.: 0040-264-530918
e-mail: antaltiberiu@pcnet.ro

`int isalnum(int c);` diferit de 0 dacă `c` este un caracter sau un număr (`'A'-'Z'`, `'a'-'z'`, `'0'-'9'`), adică dacă sunt adevărate `isalpha(c)` sau `isdigit(c)` . Altfel are valoare `0`;

`int isalpha(int c);` diferă de `0` dacă `c` este o literă din alfabet (`'A'-'Z'`, `'a'-'z'`);

`int iscntrl(int c);` diferă de `0` dacă `c` este un caracter de control (`0x7F`, `0x00-0x1F`);

`int isdigit(int c);` diferă de `0` dacă `c` este o cifră (`'0'-'9'`);

`int isgraph(int c);` diferă de `0` pentru cazul în care caracterul este grafic, adică se poate afişa şi este diferit de spaţiu (`0x21-0x7E`);

`int islower(int c);` diferă de `0` dacă `c` este o literă mică (`'a'-'z'`);

`int isprint(int c);` diferă de `0` dacă `c` este un caracter ce se poate afişa, inclusiv spaţiu (`0x20-0x7E`);

`int ispunct(int c);` diferă de `0` dacă `c` este un semn de punctuaţie diferit de spaţiu, cifră sau literă;

`int isspace(int c);` diferă de 0 dacă `c` este `spaţiu`, `formfeed`, `newline`, `carriage return`, `horizontal tab` sau `vertical tab` (`0x09-0x0D`, `0x20`);

`int isupper(int c);` diferă de `0` dacă `c` este o literă mare (`'A'-'Z'`);

`int isxdigit(int c);` diferă de `0` dacă `c` este o cifră hexazecimală;

`int tolower(int c);` întoarce echivalentul literei mici corespunzătoare lui `c`;

`int toupper(int c);` întoarce echivalentul literei mari corespunzătoare lui `c`.

Remarcă

`isupper(c)` implică `isalpha(c)`
`islower(c)` implică `isalpha(c)`
`issspace(c)`, `ispunct(c)`, `iscntrl(c)` sau `isdigit(c)` implică `!isalpha(c)`

ANTAL Tiberiu Alexandru
tel.: 0040-264-530918
e-mail: antaltiberiu@pcnet.ro

14.3 <float.h>

`FLT_RADIX` - rădăcina reprezentării exponenţiale;

`FLT_ROUNDS` - modul de rotunjire pentru adunări în virgulă flotantă;

`FLT_DIG` - numărul de cifre `q` astfel încât un număr în virgulă flotantă cu `q` zecimale să poată fi rotunjit la o reprezentare în virgulă flotantă şi invers fără pierderea

preciziei;

`FLT_EPSILON` - cel mai mic număr **`x`** astfel încât **`1.0 + x != 1.0`**

`FLT_MANT_DIG` - numărul de cifre zecimale din rădăcină specificate de **`FLT_RADIX`** semnificative în virgulă flotantă;

`FLT_MAX` - cel mai mare număr în virgulă flotantă care se poate reprezenta;

`FLT_MAX_EXP` - cel mai mare întreg astfel încât **`FLT_RADIX`** ridicat la acea putere să dea un număr care se poate reprezenta în virgulă flotantă;

`FLT_MIN` - valoarea minimă normalizată pentru un număr în virgulă flotantă;

`FLT_MIN_EXP` - valoarea întreagă negativă minimă astfel încât **`FLT_RADIX`** ridicat la acel număr să dea o valoarea care se poate reprezenta în virgulă flotantă;

Constantele prezentate până acum au format obiectul valorilor în virgulă flotantă cu simplă precizie (corespunzătoare tipului **`float`**). Cele care urmează au aceleaşi semnificaţii (şi se pot identifica prin terminaţia scrisă în urma primului caracter **`underscore`**, " _", din nume; de exemplu, **`FLT_DIG`** şi **`DBL_DIG`** au aceeaşi semnificaţie, dar valorile lor sunt diferite. **`FLT_DIG = 6`**, iar **`DBL_DIG = 15`**) cu cele prezentate deja cu diferenţa că sunt definite pentru valorile reprezentate în virgulă flotantă cu dublă precizie (corespunzătoare tipului **`double`**): **`DBL_DIG`**, **`DBL_EPSILON`**, **`DBL_MANT_DIG`**, **`DBL_MAX`**, **`DBL_MAX_EXP`**, **`DBL_MIN`**, **`DBL_MIN_EXP`**.

14.4 <limits.h>

`CHAR_BIT` - numărul de biţi dintr-un tip **`char`**;

`CHAR_MAX` - valoarea maximă pentru o variabilă de tipul **`char`**;

`CHAR_MIN` - valoarea minimă pentru o variabilă de tipul **`char`**;

`INT_MAX` - valoarea maximă pentru o variabilă de tipul **`int`**;

`INT_MIN` - valoarea minimă pentru o variabilă de tipul **`int`**;

`LONG_MAX` - valoarea maximă pentru o variabilă de tipul **`long`**;

`LONG_MIN` - valoarea minimă pentru o variabilă de tipul **`long`**;

`SCHAR_MAX` - valoarea maximă pentru o variabilă de tipul **`signed char`**;

`SCHAR_MIN` - valoarea minimă pentru o variabilă de tipul **`signed char`**;

`SHRT_MAX` - valoarea maximă pentru o variabilă de tipul **`short`**;

`SHRT_MIN` - valoarea minimă pentru o variabilă de tipul **`short`**;

`UCHAR_MAX` - valoarea maximă pentru o variabilă de tipul **`unsigned char`**;

`UCHAR_MIN` - valoarea minimă pentru o variabilă de tipul **`unsigned char`**;

`UINT_MAX` - valoarea maximă pentru o variabilă de tipul **`unsigned long`**;

`ULONG_MAX` - valoarea minimă pentru o variabilă de tipul **`unsigned long`**;

`USHRT_MAX` - valoarea minimă pentru o variabilă de tipul **`unsigned short`**;

14.5 <math.h>

Din motive istorice biblioteca matematică este definită numai pentru tipul `double`. Toate numele formate prin adăugarea unui `f` sau `l` numelui din `<math.h>` sunt rezervate pentru a permite specificarea de biblioteci având definiţii şi pentru tipurile `float` şi `long double`.

14.5.1 Funcţii trigonometrice

`double sin(double x);` - sinus

`double cos(double x);` - cosinus

`double tan(double x);` - tangenta are singularităţi la multiplii impari ai lui $\pi/2$ şi se apropie de valorile $+\infty$ şi $-\infty$ la extremităţile domeniului de definiţie. Implementările comune preferă să reducă argumentul folosind cea mai bună reprezentare a maşinii pentru π; pentru argumente foarte apropiate de singularităţi reducerea poate conduce la o valoare de partea greşită a singularităţii;

Conf. dr. ing. ANTAL Tiberiu Alexandru
tel.: 0040-264-530918
e-mail: antaltiberiu@pcnet.ro

`double asin(double x);` - arcsinus, rezultatul este în domeniul $[-\pi/2, \pi/2]$ şi $x \in [-1,1]$;

`double acos(double x);` - arccosinus, rezultatul este în domeniul $[0, \pi]$ şi $x \in [-1,1]$;

`double atan(double x);` - arctangentă, rezultatul este în domeniul $[-\pi, \pi]$.

`double atan2(double y, double x);` - funcţia arctangentă `atan2()` este preluată din limbajul FORTRAN şi corespunde lui `arctan(y/x)`. Este bine definită şi pentru cazul în care `x=0`, dar nu şi pentru cazul în care ambele argumente sunt nule; rezultatul este în domeniul $[-\pi, \pi]$;

Conf. dr. ing. ANTAL Tiberiu Alexandru
Universitatea Tenhică din Cluj-Napoca
Facultatea Construcții de Mașini
Catedra de Mecanică și Programare
Curs de limbaj C

`double sinh(double x);` - sinus hiperbolic;

`double cosh(double x);` - cosinus hiperbolic;

`double tanh(double x);` - tangentă hiperbolică.

14.5.2 Funcţii exponenţiale şi logaritmice

`double exp(double x);` - e la `x`, e^x ;

`double log(double x);` - logaritm natural, $\ln(x)$, $x>0$;

`double log10(double x);` - logaritm zecimal, $\log_{10}(x)$, $x>0$;

`double pow(double x, double y);` - `x` la puterea `y`, x^y ; o eroare de domeniu apare dacă $x = 0$ şi $y \le 0$ sau dacă $x<0$ şi `y` nu este întreg;

`double sqrt(double x);` - rădăcină pătrată, $\sqrt{x}$, $x \ge 0$.

14.5.3 Cel mai apropiat întreg, valoare absolută, resturi

Conf. dr. ing. ANTAL Tiberiu Alexandru
tel.: 0040-264-530918

`double ceil(double x);` - cel mai mic număr întreg care nu este mai mic decât `x`;

`double floor(double x);` - cel mai mare număr întreg care nu este mai mare decât `x`;

double fabs(double x); - valoarea absolută a lui **x**, |**x**|;

double ldexp(double x, int n); - întoarce **x*2** ^{n};

double frexp(double x, int* exp); - împarte pe **x** într-o fracţie normalizată în intervalul [**1/2, 1**] care este întors şi o putere a lui **2**, care este stocată în ***exp**. Dacă **x = 0** ambele rezultate sunt **0**;

double modf(double x, double *ip); - împarte un număr **x** în partea întreagă şi cea fracţionară, fiecare cu acelaşi semn al lui **x**. Partea întreagă este stocată în ***ip**, iar partea fracţionară este întoarsă;

double fmod(double x, double y); - restul împarţirii lui **x** la **y**, **mod(x,y)**;

Funcţiile **frexp()**, **ldexp()** şi **modf()** sunt primitive şi folosite de celelalte funcţii din bibliotecă. Folosirea lor este problematică pe unele arhitecturi **ldexp()** pierzând precizia şi **frexp()** putând fi ineficientă.

14.5.4 Tratarea condiţiilor de eroare

Macrodefiniţiile **EDOM** şi **ERANGE** sunt constante întregi nenule folosite pentru semnalarea erorilor de **domeniu** şi de **reprezentare**. O eroare de **domeniu** apare atunci când argumentul este în afara domeniului pentru care este definită funcţia. În cazul acesta variabila **errno** ia valoarea **EDOM**. O eroare de **reprezentare** apare dacă rezultatul unei funcţii nu se poate reprezenta folosind tipul **double**. Dacă rezultatul a dus la o depăşire, funcţia în cauză întoarce valoarea **HUGE_VAL** şi **errno** primeşte valoarea **ERANGE**. Dacă rezultatul este o valoare prea mică, funcţia întoarce valoarea **0**; unele implementări pot seta în acest caz **errno** la **ERANGE**.

14.6 <setjmp.h>

Declaraţiile din **<setjmp.h>** furnizează o metodă de evitare a secvenţei normale de apel şi revenire dintr-o funcţie, tipic pentru a permite o revenire imediată dintr-un apel de funcţii imbricate.

int setjmp(jmp_buf env); - salvează informaţia de stare în **env** pentru a putea fi ulterior folosită de **longjmp()**. Valoarea **0** este întoarsă dacă apelul a fost direct şi o valoare diferită de **0** dacă saltul s-a făcut cu un apel ulterior al unui **longjmp()**.

void longjmp(jmp_buf env, int val); - reface starea programului salvată prin cel mai recent apel al lui **setjmp()** pe baza informaţiei salvate în **env**. Execuţia se reia ca şi când **setjmp()** tocmai s-a executat şi a întors valoare nenulă pentru **val**. Funcţia care conţine **setjmp()** nu a fost terminată. Permite implementarea unui "**goto**" nelocal.

```
/* JMP.C */
#include<stdio.h>
#include<setjmp.h>

int valoare;
jmp_buf saltaici;
```

```
7  void salt(void);
8
9  int main(void)
10 {
11         valoare=setjmp(saltaici);
12         if (valoare != 0)
13         {
14                 printf("Salt cu valoarea %d\n", valoare);
15                 return valoare;
16         }
17         printf("Acum se apeleaza functia salt()\n");
18         salt();
19         return 0;
20 }
21 void salt(void)
22 {
23         longjmp(saltaici,0);
24 }
25
```

Rezultate:

```
Acum se apeleaza functia salt()
Salt cu valoarea 1
```

Remarcă: Aceste rutine sunt utile când se tratează erori sau excepţii apărute în funcţii de nivel foarte jos ale unui program.

14.7 <signal.h>

Antetul **<signal.h>** asigură facilitaţi pentru tratarea condiţiilor excepţionale care apar în cursul execuţiei programului. De exemplu, apariţia unui semnal de întrerupere de la o sursă externă sau apariţia unei erori în cursul execuţiei programului.

void (*signal(int sig, void (*handler)(int)))(int); - instalează o rutină de tratare a erorilor software pentru semnalul **sig**, aceasta determinând modul în care vor fi tratate semnalele ulterioare. Dacă **handler** ia valoarea **SIG_DFL** se foloseşte rutina de tratare definită prin implementare, iar dacă **handler** ia valoarea **SIG_IGN** semnalul este ignorat; altfel, funcţia poantată de **handler** este apelată cu argumentul **sig.** **signal()** întoarce valoarea anterioară a lui **handler** pentru semnalul specific sau **SIG_ERR** în caz de eroare. Când apare un semnal **sig** ulterior, semnalul este refăcut la starea implicită. Dacă se revine din rutina de tratare a erorii execuţia se reia din locul în care a apărut semnalul. Starea iniţială a semnalelor este definită prin implementare. Valori permise pentru semnale sunt:

SIGABRT - terminare anormală, de exemplu cu **abort()**;

SIGFPE - eroare aritmetică, de exemplu împarţire cu zero sau depăşire;

SIGILL - imagine ilegală de funcţie, de exemplu o instrucţiune ilegală;

SIGINT - atenţionare interactivă, de exemplu o întrerupere;

`SIGSEGV` - acces ilegal la o zonă de RAM, de exemplu accesul este în afara limitelor memoriei;

`SIGTERM` - cerere de terminare transmisă programului.

`int raise(int sig);` - transmite semnalul **`sig`** programului, întoarce o valoare nenulă în caz de insucces.

14.8 <stdarg.h>

Antetul **`<stdarg.h>`** asigură facilităţi pentru parcurgerea listei argumentelor unei funcţii, în cazul în care nu se cunoaşte numărul şi tipul acestora.

`void va_start(va_list ap, lastarg);` - macro pentru iniţializare care trebuie apelat înainte de accesarea oricărui argument necunoscut. **`ap`** trebuie declarată ca şi o variabilă locală de tipul **`va_list`**, **`lastarg`** este numele ultimului parametru cunoscut al funcţiei;

`type va_arg(va_list ap, type);` - întoarce o valoare de tipul **`type`** şi de valoarea următorului argument fără nume. Modifică **`ap`** aşa încât următoarea folosire a lui **`va_arg()`** va întoarce următorul argument;

`void va_end(va_list ap);` - trebuie apelată o dată după prelucrarea tuturor argumentelor, şi apoi încă o dată înaintea terminării funcţiei.

14.9 <stdio.h>

Un flux (**`stream`**) este o sursă sau o destinaţie de date care poate fi asociată unui periferic (disc, tastatură, monitor etc.). Multe dintre implementările bibliotecilor C (mai ales sub sistemul de operare UNIX) asigură, pe lângă funcţiile standard de I/E (**`fopen()`**, **`fclose()`**, **`fread()`**, **`fwrite()`**, **`fseek()`**), o mulţime de servicii de I/E nebuffer-ate (**`open()`**, **`close()`**, **`read()`**, **`write()`**, **`lseek()`**). Comitetul de standardizare a decis să nu standardizeze acest grup de funcţii. Biblioteca asigură lucrul cu fluxuri de **`text`** şi **`binare`**, deşi sub unele sisteme de operare, între acestea nu sunt diferenţe. Un **`flux de text`** este o secvenţă de linii, o linie este formată din zero, unul sau mai multe caractere şi se termină prin **`'\n'`**. În unele contexte este necesară conversia unui flux de caractere de la o reprezentare la alta, de exemplu caracterul **`'\n'`** se converteşte în caracterele **`carriage return`** şi **`line feed`**. Un **`flux binar`** este o secvenţă neprelucrată de octeţi care se stochează într-un format intern cu proprietatea că dacă acestea sunt scrise şi apoi citite pe acelaşi sistem, compararea celor două secvenţe de date va conduce la egalitate. Un flux este conectat la un dispozitiv în urma operaţiei numite **`deschiderea fluxului`**, iar conexiunea este întreruptă prin operaţia numită **`închiderea fluxului`**. Deschiderea unui fişier întoarce un poantor la un obiect de tipul **`FILE`** care stochează toate informaţiile necesare pentru controlarea fluxului. Noţiunile de poantor la fişier şi flux sunt echivalente. La începerea execuţiei unui program fluxurile **`stdin`**, **`stdout`** şi **`stderr`** sunt deschise automat.

Concepte privitoarea la fluxuri care pun probleme:

Definiţia liniei: În UNIX, gruparea unui fişier în linii se face sub efectul caracterului **`newline`** (**`'\n'`**). Tehnicile folosite în acest scop sunt dependente de sistemul de operare, de exemplu liniile pot fi separate prin caracterele **`carriage return`** şi **`line feed`**, prin porţiuni neînregistrate ale mediului de stocare sau prin prefixarea liniei prin lungimea acesteia.

În standard această diversitate de tehnici este referită specificând că **`newline`** este folosit ca separator de linie la nivelul programului, însă în funcţie de convenţiile mediului pe care se face rularea, o implementare particulară poate transforma acest caracter, la citire sau la scriere, conform unor convenţii particulare de mediu.

Transparenţa: Unele programe necesită accesul la surse de date externe care nu suportă modificări. De exemplu, transformarea caracterelor **`carriage return`** şi **`line feed`** în caracterul **`newline`** este interzisă în cazul prelucrării unui cod obiect. Standardul defineşte din acest motiv cele două tipuri de fluxuri, cel text şi cel binar, pentru a permite ca un program să specifice dacă la deschiderea unui fişier conţinutul acestuia va fi păstrat exact sau dacă organizarea datelor sub forma liniilor este mai importantă întrucât mediul nu poate reflecta cu acurateţe simultan ambele trăsături.

Accesul aleator: Modelul de I/E UNIX asigură accesul aleator la datele unui fişier, indexat prin numărul de ordine al caracterului. Pe sisteme unde caracterul **`newline`** este prelucrat de program, devenind un număr necunoscut de caractere fizice acest mecanism simplu nu îşi păstrează consistenţa dacă fluxul este de tipul text. Standardul abstractizează proprietăţile semnificative ale accesului aleator la fluxurile de text de exemplu posibilitatea determinării poziţiei curente într-un fişier unde, prin repoziţionare, se va putea reveni mai târziu. În acest scop **`ftell()`** întoarce **indicatorul de poziţie în fişier**, care are o semnificaţie corectă numai dacă este interpretat de **`fseek()`**. Având ca valoare acest indicator, **`fseek()`** va realiza repoziţionarea în fişier în acelaşi loc. Astfel, o implementare poate să codifice informaţia de poziţionare corespunzătoare unui fişier text cât mai bine, singura constrângere fiind aceea că ea trebuie să poată fi reprezentată ca o valoare **`long`**. Folosirea funcţiilor **`fgetpos()`** şi **`fsetpos()`** elimină chiar şi această constrângere.

Buffer-are: UNIX permite controlul extinderii şi al tipului de buffere din mai multe motive. De exemplu, un program poate avea propriul lui buffer de I/E pentru a creşte performanţele sau poate realiza prelucrarea nebufferată a caracterelor unui terminal de I/E care citeşte caraterul imediat cum este introdus. Unele sisteme asigură accesul exclusiv la datele terminalului de intrare la nivel de linie, altele permit lucrul cu buffer-e alocate în program numai prin copierea datelor din sau în buffer-ele alocate de sistemul de operare. În standard buffer-ele sunt manipulate prin funcţiile **`setbuf()`** şi **`setvbuf()`** care asigură o gamă mare de facilităţi.

14.9.1 Constante

`FILE` - tip care stochează informaţiile necesare pentru controlul unui flux;

`stdin` - fluxul standard de intrare; deschis automat la începutul execuţiei unui program;

`stdout` - fluxul standard de ieşire; deschis automat la începutul execuţiei unui program;

`stderr` - fluxul standard de eroare; deschis automat la începutul execuţiei unui program;

`FILENAME_MAX` - lungimea maximă permisă pentru un nume de fişier;

`FOPEN_MAX` - numărul maxim de fişiere care pot fi deschise simultan;

`TMP_MAX` - numărul maxim de fişiere temporare care pot fi deschise în timpul execuţiei unui program.

14.9.2 Funcţii pentru operaţii cu fişiere

Tipul **size_t** este unul întreg, fără semn întors de operatorul **sizeof**.

FILE* fopen(const char* filename, const char* mode); - deschide un fişier şi întoarce un flux sau **NULL** dacă operaţia nu reuşeşte. Modurile (sau combinaţiile) corecte pentru **mode** sunt:

- **"r"** - deschide un fişier text pentru citire;
- **"w"** - creează un fişier text pentru scriere; dacă acesta există deja, conţinutul lui este şters;
- **"a"** - mod de adăugare; scrierea se face începând cu sfârşitul fişierului;
- **"r+"** - deschide un fişier text în modul de actualizare (citire sau scriere);
- **"w+"** - creează un fişier text pentru actualizare; dacă acesta există deja, conţinutul lui se şterge;
- **"a+"** - mod de adăugare; fişierul este deschis sau creat pentru actualizare, iar scrierea se face la sfârşitul lui.

Modul de actualizare permite citirea şi scrierea aceluiaşi fişier. **fflush()** sau o funcţie pentru poziţionare (**fsetpos()**, **fseek()**, **rewind()**) trebuie apelată între o citire şi o scriere sau invers. Dacă modul include **b** după litera iniţială, de exemplu "**rb**" sau "**w+b**", fişierul este prelucrat în modul binar.

FILE* freopen(const char* filename, const char* mode, FILE* stream); - deschide fişierul cu numele **filename** în modul **mode** şi îi asociază un flux. Întoarce fluxul sau **NULL** în cazul unei erori. **freopen()** se foloseşte de obicei pentru a schimba fişierele asociate lui **stdin**, **stdout** sau **stderr**.

int fflush(FILE* stream); - pentru un flux de ieşire, face ca toate buffer-ele care conţin încă date nescrise să fie scrise pe flux; efectul este nedefinit în cazul unui flux de intrare. În caz de eroare întoarce **EOF**, altfel **0**. **flush(NULL)** goleşte toate fluxurile de ieşire.

int fclose(FILE* stream); - închide un flux după ce în prealabil l-a golit de toate datele care încă nu au fost scrise şi a dezalocat spaţiul folosit pentru buffer-ele acestuia. Întoarce **EOF** în caz de eroare sau altfel **0**.

int remove(const char* filename); - şterge fişierul cu numele **filename**. Întoarce o valoare nenulă în caz de eroare;

int rename(const char* oldname, const char* newname); - schimbă numele fişierului de la **oldname** la **newname**. Întoarce o valoare nenulă în caz de eroare;

FILE* tmpfile(); - creează un fişier temporar (**mode** este "**wb+**") care va fi automat şters la închidere sau dacă programul se termină normal. **tmpfile()** întoarce fluxul, respectiv **NULL** în caz de eroare;

char* tmpname(char s[L_tmpnam]); - creează un şir **s** care conţine un nume de fişier unic (numele respectiv nu mai există) şi întoarce un poantor la şirul cu legare internă şi persistenţă **static**. Şirul este stocat în **s** şi trebuie să aibă spaţiu pentru stocarea a cel puţin **L_tmpnam** caractere. **tmpname()** generează câte un nume diferit pentru fiecare apel, cel mult **TMP_MAX** nume diferite fiind garantate în timpul execuţie programului.

`int setvbuf(FILE* stream, char* buf, int mode, size_t size);` - asigură controlul buffer-elor pentru flux. Trebuie apelată înainte de orice operaţie (citire, scriere etc.) cu fluxul. Dacă **`mode`** ia valoarea **`_IOBUF`** buffering-ul este complet, pentru **`_IOLBF`** buffering-ul există numai pentru fişierele text, iar pentru **`_IONBF`** nu se face buffering. Dacă **`buf`** nu este **`NULL`** va fi folosit ca şi buffer. **`size`** determină mărimea buffer-ului. **`setbuf()`** întoarce valoare nenulă în caz de eroare;

`void setbuf(FILE* stream, char* buf);` - dacă **`buf`** este **`NULL`** buffering-ul este dezactivat pentru flux. Altfel, **`setbuf()`** este echivalent cu **`void setbuf(stream, buf, _IOFBF, BUFSIZ);`**

14.9.3 Funcții pentru ieșire cu format

`int fprintf(FILE* stream, const char* format, ...);` - convertește și scrie ieșirea pe fluxul **`stream`** sub controlul lui **`format`**. Valoarea întoarsă este numărul caracterelor scrise sau o valoare negativă în caz de eroare. Şirul **`format`** conţine două tipuri de obiecte: **`caractere normale`**, care sunt copiate pe fluxul de ieşire şi **`specificatori de conversie`**, fiecare dintre aceştia cauzând conversia şi scrierea argumentului corespunzător din **`fprintf()`**. Specificatorii de conversie încep cu **`%`** şi se termină într-un caracter de conversie. Între **`%`** şi caracterul de conversie pot să apară următoarele caractere în orice ordine:

`Indicatori (Flags)`:

`-`, aliniere la stânga;

`+`, afişare a semnului întotdeauna;

`spaţiu`, spaţiu dacă primul caracter al numărului nu este un semn;

`0`, zerouri de aliniere a numărului, pentru uşurarea prelucrărilor numerice;

`#`, modificatori de formă: pentru caracterul de conversie **`0`**, prima cifră va fi zero, pentru [**`xX`**] prefixul va fi **`0x`** sau **`0X`**, pentru [**`eEfgG`**] se va folosi în rezultat întotdeauna punctul zecimal, pentru [**`gG`**] zerourile nesemnificative nu sunt afişate.

`Lățime`: un număr care specifică numărul minim de caractere ale unui câmp. Argumentul care este convertit va fi afişat într-un câmp cu lăţimea (numărul de caractere) cel puţin egală sau mai mare decât valoarea acestui număr. Dacă în urma conversiei argumentul are mai puţine caractere decât lungimea câmpului valoarea va fi aliniată la stânga (sau la dreapta dacă se foloseşte indicatorul **`-`**) pentru a ocupa întreg spaţiul de caractere specificate prin **`Lățime`**. Caracterul folosit pentru umplere este **`spațiu`**, dar poate fi şi caracterul **`0`** dacă indicatorul **`0`** este prezent;

`Punct`: separă lăţimea de lungimea unui câmp;

`Precizie`: un număr care specifică numărul maxim de caractere care sunt afişate dintr-un şir sau numărul de zecimale care sunt afişate după punctul zecimal pentru [**`eEf`**] sau numărul de cifre semnificative pentru [**`gG`**], sau pentru un întreg numărul minim de cifre care vor fi afişate (zerouri pot fi afişate în faţa întregului dacă se foloseşte indicatorul corespunzător).

`Modificatori de lungime`:

`h`, argumentul va fi afişat ca **`short`** sau ca **`unsigned short`**;

`l` sau **`L`**, argumentul va fi afişat ca **`long`** sau ca **`unsigned long`** ;

Lăţime sau **precizie** pot fi specificate prin *, caz în care valorile se calculează prin conversia argumentului corespunzător caracterului * a cărui valoare trebuie să fie obligatoriu de tipul **int**.

Caractere de conversie fprintf():

Caracter	Tip argument	se converteşte la
d, i	**int**	notaţie zecimală cu semn.
o	**int**	notaţie octală fără semn.
x, X	**int**	notaţie hexazecimală fără semn.
u	**int**	notaţie zecimală fără semn.
c	**int**	un singur caracter, după conversie la **unsigned char**.
s	**char ***	caracterele şirului, afişarea continuă până la atingerea lui '\0' sau până când numărul maxim de caractere de afişat a fost atins.
f	**double**	notaţia de forma **[-]mmm.ddd** unde **d**-urile sunt specificate prin **precizie**. Precizia implicită este **6**, precizia **0** suprimă punctul zecimal.
e, E	**double**	notaţia de forma **[-]m.dddddde(+|-)xx** unde numărul **d**-urilor este specificat prin **precizie**.
g, G	**double**	**%e** sau **%E** sunt folosiţi dacă exponentul este mai mic decât **-4 sau mai mare sau egal cu precizia, altfel se foloseşte %f**. Zerourile nesemnificative şi punctul zecimal nu se afişează.
p	**void ***	afişarea de poantor.
n	**int ***	numărul de caractere scrise până în acel moment de **printf()** este scris în argument. Nu se realizează conversia argumentului.
%		afişarea caracterului **%**.


```
int printf(const char* format, ...);
```
- **printf(...)** este echivalentă cu **fprintf(stdout, f, ...)**;

```
int sprintf(char* s, const char* format, ...);
```
- lucrează ca **fprintf()**, cu excepţia că ieşirea este scrisă în şirul **s** terminat în '\0'. **s** trebuie să fie suficient de lung pentru a putea stoca rezultatul. Valoarea întoarsă este numărul de caractere fără '\0'.

```
int vfprintf(FILE* stream, const char* format, va_list arg);
int vprintf(const char* format, va_list arg);
int vsprintf(char* s, const char* format, va_list arg);
```

Funcţiile **vprintf()**, **vfprinf()** şi **vsprintf()** sunt echivalente cu funcţiile corespunzătoare

printf(), cu excepţia că lista variabilă de argumente este înlocuită prin **arg**, care a fost iniţializată folosind macrodefiniţia **va_start**, eventual prin apeluri ale lui **va_arg**.

14.9.4 Funcţii pentru intrare cu format

int fscanf(FILE* stream, const char* format, ...); - citeşte de pe fluxul **stream** sub controlul lui **format** şi atribuie valorile convertite argumentelor care urmează, fiecare dintre acestea trebuind să fie un poantor. Revenirea se face când s-a epuizat **format**. **fscanf()** întoarce valoarea **EOF** dacă s-a întâlnit sfârşitul fişierului sau dacă a apărut o eroare. Altfel întoarce numărul de articole convertite şi atribuite. Şirul format poate conţine:

- **spaţii**, **tab**-uri: se ignoră, acestea fac parte dintre caracterele albe (**white space** = **spaţiu**, **tab**, **new-line**, **carriage return**, **vertical tab**, **formfeed**);
- **caracterе normale**: care trebuie să fie identice cu următorul caracter ne-alb;
- **% : specificatori de conversie** formaţi din **%**, un caracter opţional pentru suprimarea atribuirii, un număr opţional specificând lăţimea maximă a câmpului, unul dintre caracterele opţionale [**hlL**] indicând mărimea destinaţiei, şi un caracter de conversie. Un câmp de intrare se defineşte ca un şir de caractere ne-albe şi ţine până la următorul caracter alb sau, în cazul în care este specificat, este determinat de lăţimea câmpului. Un caracter de conversie determină conversia următorului câmp de intrare. În condiţii normale, rezultatul este plasat în variabila poantată de argumentul corespunzător. Dacă argumentul specifică suprimarea atribuirii prin *, de exemplu **%*s** câmpul de intrare este sărit fără a se face atribuirea. Caracterul de conversie specifică modul de interpretare a câmpului de intrare. Caracterele de conversie [**dinoux**] pot fi precedate de **h** dacă argumentul este un poantor **short**, şi nu unul **int**, sau de **l** dacă argumentul este un poantor **long**. Caracterele de conversie [**efg**] pot fi precedate de **l** dacă argumentul este un poantor la **double**, în locul unuia **float**, sau de **L** dacă poantorul este la **long double**.

Conf. dr. ing. ANTAL Tiberiu Alexandru
Universitatea Tehnică din Cluj-Napoca
Facultatea Construcții de Mașini
Catedra de Mecanică și Programare
Curs de limbaj C

Caractere de conversie fscanf():

Caracter	Tip argument	Data de intrare
d	**int** *	întreg zecimal.
i	**int** *	întregul poate fi în **octal** (începe cu **0**) sau în **hexazecimal** (începe cu **0x** sau **0X**).
o	**int** *	întreg **octal**, cu sau fără **0** la început.
x	**int** *	întreg **hexazecimal** cu sau fără **0x** la început.
u	**unsigned int***	întreg zecimal fără semn.

c	**char** *	următoarele caractere de intrare sunt plasate în tabloul indicat până se atinge lăţimea de caractere specificată (implicit aceasta este **1**). Nu se adaugă '**\0**'. Saltul normal peste caracterele albe este inhibat, pentru citirea următorului caracater ne-alb se foloseşte **%1s**.
s	**char** *	şir de caractere ne-albe (fără ghilimele). Argumentul va poanta la un tablou de caractere suficient de mare pentru stocarea şirului şi a terminatorului '**\0**' care va fi adăugat automat.
e, f, g	**float** *	număr în virgulă flotantă. Opţional, acesta poate conţine semn, un şir de numere care pot sau nu să conţină punctul zecimal sau un exponent opţional conţinând **e** sau **E**, urmat de un număr întreg cu semn.
p	**void** *	valoare de poantor, aşa cum este afişată de **printf("%p")**.
n	**int** *	scrie în argument numărul de caractere citite până în acel moment de apelul funcţiei. Nu se citeşte nimic de pe intrare. Numărul articolelor convertite nu este incrementat.
[...]	**char** *	caracterele din interiorul parantezelor dreptunghiulare definesc o mulţime de căutare pentru caracterele posibile în câmpul de intrare şi diferite de şirul vid. '**\0**' este adăugat automat la mulţime.
[^...]	**char** *	caracterele din interiorul parantezelor dreptunghiulare definesc o mulţime de căutare pentru caractere care *nu pot face parte* din câmpul de intrare şi sunt diferite de şirul vid. '**\0**' este adăugat automat la mulţime.
%	**%**	afişarea caracterului **%** .

int scanf(const char* format, ...); - **scanf(f, ...)** este echivalentă cu **fscanf(stdin, f, ...)**

int sscanf(char* s, const char* format, ...); echivalentă cu **fscanf()**, numai că intrarea este stocată în şirul **s**.

14.9.5 Funcţii de intrare/ieşire la nivel de caracter

int fgetc(FILE* stream); - întoarce următorul caracter de pe fluxul **stream**, convertit la un întreg de tipul **unsigned char** sau **EOF** în caz de eroare, respectiv la terminarea fişierului.

char* fgets(char* s, int n, FILE* stream); - citeşte cel mult **n-1** caractere de pe

stream în şirul **s**, oprindu-se la întâlnirea caracterului **newline**. Caracterul **newline** este inclus în **s** care se termină în '**\0**'; întoarce **s** sau **NULL** în caz de eroare, respectiv la terminarea fişierului.

int fputc(int c, FILE* stream); - scrie pe **c**, convertit la **unsigned char** pe **stream**. Întoarce caracterul scris sau **EOF** în caz de eroare.

char* fputs(const char* s, FILE* stream); - scrie şirul **s** care nu trebuie să conţină caracterul '**\n**' pe fluxul **stream**. În caz de succes întoarce o valoare nenulă, respectiv **EOF** în caz contrar.

int getc(FILE* stream); - echivalentă cu **fgetc()**, cu excepţia faptului că poate fi un macro; poate evalua **stream** de mai multe ori.

int getchar(); - echivalentă cu **getc(stdin)**.

char* gets(char* s); - citeşte următoarea linie de pe **stdin** în **s**. Înlocuieşte caracterul terminator **newline** cu '**\0**'; întoarce **s** sau **NULL** la terminarea fişierului respectiv în caz de eroare.

ANTAL Tiberiu Alexandru
tel.: 0040-264-530918
e-mail: antaltiberiu@pcnet.ro

int putc(int c, FILE* stream); - echivalentă cu **fputc()** cu excepţia faptului că poate fi un macro, poate evalua pe **stream** de mai multe ori.

int putchar(int c); - este echivalentă cu **putc(c, stdout)**.

int puts(const char* s); - scrie pe **s** şi un caracter **newline** pe **stdout**; întoarce o valoare nenegativă în caz de succes, respectiv **EOF** în caz contrar.

int unget(int c, FILE* stream); - îl împinge pe **c**, care trebuie să fie diferit de **EOF**, convertit la **unsignd char** pe **stream**, aşa încât acesta va fi întors la următoarea citire. Este garantată împingerea pe **stream** numai a unui singur caracter; întoarce **c** sau **EOF** în caz de eroare.

14.9.6 Funcţii de intrare/ieşire directe

size_t fread(void* ptr, size_t size, size_t nobj, FILE* stream); - citeşte cel mult **nobj** obiecte cu dimensiunea **size** de pe fluxul **stream** în **ptr**. Întoarce numărul de obiecte citite. Funcţiile **feof()** şi **ferror()** trebuie folosite pentru determinarea stării.

size_t fwrite(const void* ptr, size_t size, size_t nobj, FILE* stream); - scrie pe fluxul **stream** **nobj** obiecte cu dimensiunea **size** dintr-un tablou **ptr**. Întoarce numărul de obiecte scrise care nu va fi mai mic ca **nobj** decât în caz de eroare.

14.9.7 Funcţii de poziţionare

int fseek(FILE* stream, long offset, int origin); - setează poziţia în fişier pentru fluxul **stream**. Pentru un fişier binar poziţia este setată printr-un deplasament de caractere faţă de origine, care poate fi **SEEK_SET** (început), **SEEK_CUR** (poziţia curentă) sau **SEEK_END** (sfârşitul fişierului); pentru un flux text **offset** trebuie să fie **0** sau o valoare întoarsă de **ftell()**, caz în care originea trebuie să fie **SEEK_SET**. Întoarce o valoare nenulă în caz de eroare.

ANTAL Tiberiu Alexandru
tel.: 0040-264-530918
e-mail: antaltiberiu@pcnet.ro

long ftell(FILE* stream); - întoarce poziţia curentă în fişier pentru fluxul **stream** sau **-1L** în caz de eroare.

void rewind(FILE* stream); - este echivalentă cu **fseek(stream, 0L, SEEK_SET); clearerr(stream)**.

`int fgetpos(FILE* stream, fpos_t* ptr);` - îi atribuie lui ***ptr** poziţia curentă în fluxul **stream**. Tipul **fpos_t** se foloseşte pentru stocarea acestor valori; întoarce o valoare nenulă în caz de eroare.

`int fsetpos(FILE* stream, const fpos_t* ptr);` - setează poziţia curentă în fluxul **stream** la ***ptr**; întoarce o valoare nenulă în caz de eroare.

14.9.8 Funcţii pentru tratarea erorilor

`void clearerr(FILE* stream);` - şterge indicatorii de terminare de fişier şi de eroare pentru fluxul **stream**.

`int feof(FILE* stream);` - întoarce o valoare nenulă dacă indicatorul de terminare de fişier este setat pentru fluxul **stream**.

`int ferror(FILE* stream);` - întoarce o valoare nenulă dacă indicatorul de eroare a fluxului **stream** este setat.

`void perror(const char* s);` - afişează pe **s** şi un mesaj de eroare dependent de implementare corespunzător lui **errno**, este echivalentă cu **`fprintf(stderr, "%s: %s\n", s, "mesaj de eroare")`**.

ANTAL Tiberiu Alexandru
tel: 0040-264-330 115
e-mail: antaltiberiu@pcnet.ro

14.10 <stdlib.h>

Antetul **<stdlib.h>** conţine un set de funcţii pentru diferite conversii numerice, alocare de memorie şi alte acţiuni având un caracter similar.

`double atof(const char* s);` - converteşte pe **s** la **double**, este echivalentă cu **`strtod(s, (char**)NULL)`**.

`int atoi(const char* s);` - converteşte pe **s** la **int**, este echivalentă cu **`(int)strtol(s, (char**)NULL, 10)`**.

`long atol(const char* s);` - converteşte pe **s** la **long**, este echivalentă cu **`strtol(s, (char**)NULL, 10)`**.

`double strtod(const char* s, char** endp);` - converteşte prefixul lui **s** la **double**, ignorând orice spaţiu alb aflat în faţă. Stochează un poantor la orice sufix neconvertit în ***endp** excepţie făcând cazul în care **endp** e **NULL**. Dacă rezultatul conduce la depăşire, **HUGE_VAL** este întors cu semnul corespunzător, dacă rezultatul este o valoare prea mică, întoarce valoarea **0**. În ambele cazuri **errno** primeşte valoarea **ERANGE**.

`long strtol(const char* s, char** endp, int base);` - converteşte prefixul lui **s** la **long** ignorând spaţiile albe de la început. Stochează un poantor la orice sufix neconvertit în ***endp** dacă **endp** este diferit de **NULL**. Dacă **base** este între **2** şi **36**, conversia se face presupunând că intrarea este în baza respectivă. Dacă **base** este **0**, baza este **8** sau **16**. Un zero (**0**) în faţă implică octal, iar **0x** sau **0X** implică hexazecimal, altfel zecimal. Cifrele în ambele cazuri trebuie să fie în domeniul [**0, base-1**]. Dacă rezultatul produce depăşire, **LONG_MAX** sau **LONG_MIN** este întors, în funcţie de semnul rezultatului, iar **errno** primeşte valoarea **ERANGE**.

`unsigned long strtoul(const char* s, char** endp, int base);` - lucrează identic cu **strtol()**, excepţie facând rezultatul întors care este de tipul **unsigned long**, iar valoarea de eroare **ULONG_MAX**.

`int rand();` - întoarce un număr aleator în domeniul [`0`, `RAND_MAX`].

`void srand(unsigned int seed);` - foloseşte `seed` pentru a genera o nouă secvenţă de numere pseudo-aleatoare. Valoarea iniţială este `1`.

`void* calloc(size_t nobj, size_t size);` - întoarce un poantor la o zonă nou alocată de RAM corespunzătoare unui tablou iniţializat cu valori nule de `nobj` obiecte, fiecare de dimensiunea `size` sau `NULL` dacă alocarea nu se poate face.

`void* malloc(size_t size);` - întoarce un poantor la o zonă de RAM nou alocată şi neiniţializată pentru un obiect de dimensiunea `size` sau `NULL` dacă alocarea nu se poate face.

`void* realloc(void* p, size_t size);` - modifică spaţiul alocat obiectului poantat de `p` la `size`. Conţinutul blocului este neschimbat până la mimimul dintre dimensiunea veche şi noua dimensiune. Dacă noua dimensiune este mai mare, noul spaţiu rămâne neiniţializat. Întoarce un poantor la noul spaţiu alocat sau `NULL` dacă alocarea nu se poate realiza, caz în care `*p` rămâne nemodificat.

`void free(void* p);` - dezalocă spaţiul la care poantează `p`. `p` trebuie să fie `NULL`, caz în care nu se petrece nimic, sau un poantor întors de `calloc()`, `malloc()` sau `realloc()`.

`void abort();` - cauzează terminarea anormală a programului la fel cu `raise(SIGABRT)`.

`void exit(int status);` - cauzează terminarea normală a programului. Funcţiile instalate folosind `atexit()` sunt apelate în ordinea inversă celor de înregistrare, fişierele deschise sunt golite, fluxurile deschise sunt închise, iar controlul este redat mediului. `status` este întors mediului într-o formă dependentă de implementare. Valoarea zero înseamnă terminare cu succes, de asemenea pot fi folosite valorile `EXIT_SUCCESS` şi `EXIT_FAILURE`.

`int atexit(void (*fcn)(void));` - înregistrează funcţia `fcn` pentru ca să fie apelată la terminarea normală a programului; întoarce valoare nenulă dacă înregistrarea nu se poate realiza.

`int system(const char* s);` - îl transferă pe `s` mediului de execuţie. Când `s` este `NULL`, `system()` întoarce o valoare nenulă dacă există un procesor de comenzi. Când `s` nu este `NULL`, valoarea întoarsă este dependentă de implementare.

`char* getenv(const char* name);` - întoarce, dependent de implementare, şirul din mediu asociat cu `name`, sau `NULL` dacă acest şir nu există.

`void* bsearch(const void* key, const void* base, size_t n, size_t size, int (*cmp)(const void* keyval, const void* datum));` - caută în `base[0]` ... `base[n-1]` un articol identic cu `*key`. Funcţia `cmp` trebuie să întoarcă o valoare negativă dacă primul argument este mai mic decât cel de al doilea, zero dacă acestea sunt egale, sau pozitiv dacă primul argument este mai mare decât cel de al doilea. Cele `n` articole trebuie să fie în ordine crescătoare. Funcţia întoarce un poantor la articolul pentru care condiţia a fost satisfacută sau `NULL` dacă aceasta nu s-a găsit.

`void qsort(void* base, size_t n, size_t size, int (*cmp)(const void*, const void*));` - sortează în ordine crescătoare tabloul `base[0]...base[n-1]` de obiecte cu dimensiunea `size`. Funcţia de comparaţie `cmp` trebuie să întoarcă valoare negativă dacă primul argument este mai mic decât cel de al doilea, zero dacă argumentele sunt egale, iar altfel o valoare pozitivă.

`int abs(int n);` - întoarce valoarea absolută a lui `n` de tipul `int`.

`long labs(long n);` - întoarce valoarea absolută a lui `n` de tipul `long`.

`div_t div(int num, int denom);` - întoarce în câmpurile `quot` şi `rem` ale structurii

de tipul **dive_t** câtul şi restul lui **num/denom**.

ldiv_t ldiv(long num, long denom); - întoarce în câmpurile **quot** şi **rem** ale structurii de tipul **ldiv_t** câtul şi restul lui **num/denom**.

14.11 <string.h>

Există două grupe de funcţii pentru prelucrarea şirurilor. Prima începe cu **str**; a doua începe cu **mem**. Cu excepţia lui **memmove()**, comportamentul este nedefinit dacă copierea are loc între obiecte care se suprapun.

char* strcpy(char* s, const char* ct); - îl copiază pe **ct** în **s**, inclusiv terminatorul '**\0**'; îl întoarce pe **s**.

char* strncpy(char* s, const char* ct, int n); - copiază cel mult **n** caractere din **ct** în **s**; realizează completarea cu '**\0**' dacă lungimea lui **ct** este mai mică decât a lui **n**; îl întoarce pe **s**.

char* strcat(char* s, const char* ct); - îl concatenează pe **ct** la **s**; îl întoarce pe **s**.

char* strncat(char* s, const char* ct, int n); - concatenează lui **s** cel mult **n** caractere din **ct**; îl întoarce pe **s** care se va termina în '**\0**'.

int strcmp(const char* cs, const char* ct); - îl compară pe **cs** cu **ct**; întoarce valoare negativă dacă **cs** < **ct**, zero dacă **cs** == **ct**, sau pozitivă dacă **cs** > **ct**.

int strncmp(const char* cs, const char* ct, int n); - compară cel mult **n** caractere din **cs** cu **ct**. Întoarce valoare negativă dacă **cs** < **ct**, zero dacă **cs** == **ct**, sau pozitivă dacă **cs** > **ct**.

char* strchr(const char* cs, int c); - întoarce un poantor la prima apariţie a lui **c** în **cs** sau **NULL** dacă acesta nu este găsit.

char* strrchr(const char* cs, int c); - întoarce un poantor la ultima apariţie a lui **c** în **cs** sau **NULL** dacă acesta nu este găsit.

size_t strspn(const char* cs, const char* ct); - întoarce lungimea prefixului lui **cs** format din caracterele lui **ct**.

size_t strcspn(const char* cs, const char* ct); - întoarce lungimea prefixului lui **cs** format din caracterele care *nu* fac parte din **ct**.

char* strpbrk(const char* cs, const char* ct); - întoarce un poantor la prima apariţie în **cs** a oricărui caracter din **ct**, sau **NULL** dacă nu s-a găsit nici unul.

char* strstr(const char* cs, const char* ct); - întoarce un poantor la prima apariţie a lui **ct** în **cs**, sau **NULL** dacă aceasta nu s-a găsit.

size_t strlen(const char* cs); - întoarce lugimea lui **cs**.

char* strerror(int n); - întoarce un poantor la un şir definit prin implementare corespunzător erorii **n**.

char* strtok(char* s, const char* ct); - o secvenţă de apeluri la **strtok()** întoarce atomi (**tokens**) din **s** delimitaţi prin caracterele din **ct**. Primul apel din secvenţă are **s** cu valoare diferită de **NULL**. Acesta găseşte primul atom din **s** care nu este format din caracterele lui **ct**; va termina această căutare prin suprascrierea următorului caracter din **s** cu '**\0**' şi întoarce un poantor la atom. Fiecare apel ulterior, indicat prin valoarea **NULL** a lui **s**, întoarce următorul atom, continuând căutarea începând cu ultimul atom. **strtok()**

va întoarce **NULL** dacă nu s-a găsit un nou atom. Şirul **ct** poate fi diferit la fiecare apel.

Programul care urmează prezintă modul de prelucrare a datelor cu ajutorul funcţiei **strtok()**. Observaţi că data poate fi scrisă în oricare dintre formatele: **28 Iul., 2000; 20/07/2000; 28-07-2000.**

```
/* STRTOK.C */
#include<stdio.h>
#include<string.h>

int main(void)
{
        char s[ ]="28 Iul., 2000";
        char *ptr;
        ptr= strtok(s, " .,-/\\");
        printf("ptr = %s\n", ptr);
        ptr= strtok(NULL, " .,-/\\");
        printf("ptr = %s\n", ptr);
        return 0;
}
```

Rezultate:

```
ptr = 28
ptr = Iul
```

void* memcpy(void* s, const void* ct, int n); - copiază **n** caractere din **ct** în **s**; îl întoarce pe **s**. *Nu lucrează corect* dacă cele două obiecte se suprapun.

void* memmove(void* s, const void* ct, int n); - copiază **n** caractere din **ct** în **s**; îl întoarce pe **s**. Lucrează corect dacă cele două obiecte se suprapun.

int memcmp(const void* cs, const void* ct, int n); - compară primele **n** caractere din **cs** cu **ct**; întoarce valoare negativă dacă **cs** < **ct**, zero dacă **cs** == **ct**, pozitivă dacă **cs** > **ct**.

void* memchr(const char* cs, int c, int n); - întoarce un poantor la prima apariţie a lui **c** între primele **n** caractere ale lui **cs**, sau **NULL** dacă acesta nu este găsit.

void* memset(char* s, int c, int n); - înlocuieşte fiecare dintre primele **n** caractere ale lui **s** prin **c**; îl întoarce pe **s**.

14.12 <time.h>

Antetul **<time.h>** conţine declaraţiile tipurilor şi funcţiilor folosite pentru manipularea datelor şi a timpului. Unele funcţii prelucrează timpul local, care poate să difere de cel din calendar, de exemplu din cauza zonei de timp.

clock_t - tip aritmetic folosit pentru reprezentarea timpului;

CLOCKS_PER_SEC - numărul de unităţi **clock_t** pe secundă;

`time_t` - tip aritmetic reprezentând timpul;

`struct tm` - structură care reprezintă timpul din calendar:

`int tm_sec;` - secunde după minut (0, 61);

`int tm_min;` - minute după oră (0,59);

`int tm_hour;` - ore după miezul nopţii (0, 23);

`int tm_mday;` - ziua din lună (1, 31)

`int tm_mon;` - luni trecute după ianuarie (0, 11)l

`int tm_year;` - ani trecuţi după 1900;

`int tm_wday;` - zile trecute după duminică (0, 6);

`int tm_yday;` - zile trecute după 1 ianuarie (0, 356)

`int tm_isdst;` - indicator **D**aylight **S**aving **T**ime (**DST**): este pozitiv dacă este activ, zero dacă este inactiv şi negativ dacă informaţia nu este disponibilă.

`clock_t clock();` - întoarce timpul consumat de procesor de la începutul execuţiei programului sau, **`-1`** dacă nu este disponibil. **`clock()/CLOCKS_PER_SEC`** este timpul în secunde.

`time_t time(time_t* tp);` - întoarce timpul calendaristic curent sau **`-1`** dacă nu este disponibil. Dacă **`tp`** este ne-**`NULL`**, valoarea întoarsă este atribuită lui ***`tp`**.

`double difftime(time_t time2, time_t time1);` - întoarce diferenţa în secunde dintre **`time2`** şi **`time1`**.

`time_t mktime(struct tm* tp);` - întoarce timpul local corespunzător lui ***`tp`** sau **`-1`** dacă reprezentarea nu se poate face.

`char* asctime(const struct tm* tp);` - întoarce timpul din structura ***`tp`** sub forma şirului **`Sun Jan 3 14:14:13 1988\n\0`**.

`char* ctime(const time_t* tp);` - converteşte timpul calendaristic în Coordonate Universale de Timp (**`Coordinated Universal Time`**) şi întoarce şirul echivalent. Este echivalentă cu **`asctime(localtime(tp))`**.

`struct tm* gmtime(const time_t* tp);` - întoarce timpul calendaristic în Coordonate Universale de Timp sau **`NULL`** dacă nu este disponibil.

`struct tm* localtime(const time_t* tp);` - întoarce timpul calendaristic ***`tp`** convertit la timpul local.

`size_t strftime(char* s, size_t smax, const char* fmt, const struct tm* tp);` - formatează ***`tp`** în **`s`** conform **`fmt`** care este analog şirului format din **`printf()`**. Caracterele de conversie însă sunt altele şi vor fi prezentate în tabelul care urmează, caracterele obişnuite vor fi copiate direct în **`s`**. Cel mult **`smax`** caractere sunt copiate în **`s`**. Întoarce numărul de caractere, exclusiv '**`\0`**', sau zero dacă au fost produse mai multe de **`smax`** caractere.

Caracter de conversie	Semnificaţie
`%a`	numele zilei abreviat.
`%A`	numele complet al zilei din săptămână.
`%b`	numele abreviat al lunii.

`%B`	numele complet al lunii.
`%c`	reprezentarea locală a datei şi timpului.
`%d`	ziua din lună (01-31).
`%H`	ora (reprezentată prin 24 de ore) (00-23).
`%I`	ora (reprezentată prin 12 ore) (01-12).
`%j`	ziua din an (001-366).
`%m`	luna (01-12).
`%M`	minutul (00-59).
`%p`	echivalentul local al lui AM sau PM.
`%S`	secunda (00-61).
`%U`	numărul săptămânii din an (*duminica* este ziua 1 a săptămânii) (0-53).
`%w`	numărul zilei din săptămână (0-6, duminică este 0).
`%W`	numărul săptămânii din an (*luni* este ziua 1 a săptămânii) (0-53).
`%x`	reprezentarea locală a datei.
`%X`	reprezentarea locală a timpului.
`%y`	an fără secol (00-99).
`%Y`	an cu secol.
`%Z`	numele zonei de timp, dacă există.
`%%`	caracterul %.

Conf. dr. ing. ANTAL Tiberiu Alexandru
Universitatea Tenhică din Cluj-Napoca
Facultatea Construcţii de Maşini
Catedra de Mecanică şi Programare
Curs de limbaj C
Copyright 2001. Toate drepturile sunt rezervate autorului.
Multiplicarea acestui document în scop comercial este interzisă.
Sudenţii participanţi la orice formă de învăţământ superior bugetar pot multiplica acest document pentru uzul personal.
Sudenţii participanţi la orice formă de învăţământ cu plată sau alte persoane doritoare pot multiplica documentul numai contra cost şi cu acordul scris al autorului. În acest scop mă puteţi contactă la:
ANTAL Tiberiu Alexandru
tel.: 0040-264-530918
e-mail: antaltiberiu@pcnet.ro

Index

B

C

G

I

K

L

M

O

P

T

U

V

W

Bibliografie

[1] Niklaus Wirth, ALGORITHMS+DATA STRUCTURES = PROGRAMS, PRENTICE-HALL, INC., 1976.

[2] Brian W. Kernighan, Dennis M. Ritchie, THE C PROGRAMMING LANGUAGE, SECOND EDITION, PRENTICE HALL SOFTWARE SERIES, 1988.

[3] Steven Holzner with the Peter Norton Computing Group, C PROGRAMMNIG, Brady Publishing, 1991.

[4] Turbo C reference guide, Borland International Inc., 1987.

[5] Turbo C user's guide, Borland International Inc., 1987.

[6] Herbert Schild, C manual complet, Teora, 1998.

[7] Popescu, M., Floricică, I., Jitaru, M., Metode avansate de utilizare a tehnicii de calcul, Editura Militară, Bucureşti, 1989.

[8] M., Sargent, R., L., Shoemaker, The IMB PC from the inside out, Addison-Wesley Publishing Company, 1986.

[9] A., S., Tanenbaum, Organizarea structurată a calculatoarelor, ediţia a IV-a, Computer Press AGORA, 1999.

[10] Giumale, C. Kalisz, E., Structura datelor şi tehnici de programare, Lito. IPB, 1981.

[11] A., V., Aho, J., D. , Ullman, THE THEORY OF PARSING, TRANSLATION, AND COMPILING, VOLUME I: PARSING, PRENTICE-HALL, INC.

[12] C. H. Smedema, P. Medema, M. Boasson, The programming languages Pascal, Modula, CHILL, Ada, Prentice Hall Inc., 1983.

[13] Leon Livovschi, Bazele informaticii, Editura Albatros, 1979.

Sudenţii participanţi la orice formă de învăţământ cu plată sau alte persoane doritoare pot multiplica documentul numai contra cost şi cu acordul scris al autorului. În acest scop mă puteţi contactă la:

ANTAL Tiberiu Alexandru
tel.: 0040-264-530918
e-mail: antaltiberiu@pcnet.ro

ANEXA 1 - Scurtă comparaţie C - FORTRAN în domeniul calculului numeric

Această anexă apare într-o carte de C numai pentru că în lumea calculelor numerice prostia a început să se manifeste din plin. Firme serioase au cedat în faţa presiunilor şi a modei venite din America intoxicând lumea calculelor numerice cu tot felul de rutine scrise în C şi C++. Această tendinţă ciudată este probabil întreţinută şi de persoane care nu au experienţă în calculul numeric, domeniu în care bătrânul FORTRAN predomină, deşi în ultima vreme a cam început să piardă teren din motivele descrise mai sus. Iată câteva dintre calităţile FORTRAN-ului culese de mine de pe Internet:

A. FORTRAN satisface mai bine nevoile oamenilor de ştiinţă decât C pentru că:

- permite lucrul cu tablouri de dimensiune variabilă în subrutine, din acest motiv pot fi scrise rutine cu scop general, fără a specifica explicit dimensiunea tabloului care este transferat. C standard nu are această facilitate importantă, iar preţul plătit constă în complicarea codului care să poată trata aceste cazuri. Unele compilatorare, de exemplu `gcc`, au această extensie nestandard;

- are un grup mare de funcţii intrinseci cu precizie generică. Aceste funcţii pot fi optimizate foarte bine (prin scrierea lor în limbaj de asamblare cu optimizarea folosirii cache-ului) şi asigură o oarecare standardizare a programelor la un nivel mai înalt, crescând portabilitatea;

- asigură intern lucrul cu aritmetica numerelor complexe;

- are implementat un operator exponenţial infix care este generic în raport cu precizia şi tipul care tratează foarte eficient cazuri speciale frecvente de genul `număr flotant ** întreg mic`;

- indicii de tablouri pot începe de la un întreg arbitrar;

- FORTRAN 90 suportă o notaţie de tablou care permite operaţii pe porţiuni de tablou, acesta fiind un avantaj în cazul calculului paralel;

- FORTRAN 90 suportă selecţia automată a tipurilor de date numerice având specificate precizia şi domeniul ceea ce face programele încă şi mai portabile;

- extensiile FORTRAN pentru programare paralelă sunt strandardizate de

consorţiul **`High Performance Fortran`**.

FORTRAN 90 suportă majoritatea facilităţilor specifice limbajelor C (poantori, alocare dinamică de memorie etc.) şi C++ (supraîncărcarea operatorilor, obiecte simple) având în plus o sofisticată capacitate de manipulare a tablourilor. Are facilităţi privind precizia numerică şi controlul domeniului, crescând astfel portabilitatea programelor.

B. Modul de proiectare a limbajului FORTRAN permite viteză maximă în execuţie:

- FORTRAN 77 nu lucrează explicit cu poantori, motiv pentru care optimizarea automată a codului este simplificată; FORTRAN 90 permite poantori expliciţi, însă cu restricţii care să nu dăuneze optimizării automate de cod;

- FORTRAN a fost proiectat pentru a permite alocarea statică a memoriei, astfel nu se pierde timpul cu crearea şi distrugerea înregistrărilor de activare de pe stivă, pentru fiecare apel de procedură. Proceduri recursive nu sunt posibile cu alocarea statică, dar pot fi simulate eficient dacă este cazul;

- FORTRAN nu permite folosirea unor nume alternative între variabilele globale (COMMON) şi argumentele fictive.

Toate aceste restricţii asigură o mai bună optimizare a codului faţă de C.

C. Există deja scris un numeros cod FORTRAN, care în mare parte este public şi de foarte bună calitate.

D. C standard specifică numai existenţa bibliotecilor matematice în dublă precizie (**`double`**). Standardul FORTRAN obligă la existenţa bibliotecilor în simplă şi dublă precizie, în plus mulţi producători furnizează biblioteci şi pentru quad-precizie (**`long double`**, **`REAL*16`**). Codul în simplă precizie va fi mai rapid decât cel în dublă precizie întrucât datele în simplă precizie ocupă mai puţin spaţiu de RAM. Calculele în quad-precizie (**`long double`**) sunt uneori necesare pentru a minimiza erorile de rotunjire. În C există rutine matematice numai pentru **`double`**, acestea vor consuma un timp mai lung decât cel necesar în calcule de simplă precizie, iar exactitatea lor va lăsa de dorit dacă sunt folosite cu **`long double`**.

E. FORTRAN este proiectat să efectueze calculele numerice foarte clar şi precis definite pentru că:

1) Ordinea evaluării expresiilor aritmetice este precis definită şi poate fi controlată prin paranteze;

2) Declaraţiile implicite de tip scurtează timpul de scrierea a programelor, dar în acelaşi timp fac programele mai vulnerabile la erori şi complică depanarea;

3) Nu este dependent de scrierea cu litere mari sau mici;

4) Nu are cuvinte cheie rezervate, programatorul având libertatea deplină în alegerea numelor de identificatori;

5) Principiul o linie pe rând creşte mult lizibilitatea programelor;

6) Este un limbaj mult mai uşor de învăţat decât limbajul C.

F. Compilatoarele FORTRAN emit mesaje de eroare şi de diagnosticare mult mai bune decât cele C.

Conf. dr. ing. ANTAL Tiberiu Alexandru
Universitatea Tenhică din Cluj-Napoca
Facultatea Construcţii de Maşini
Catedra de Mecanică şi Programare
Curs de limbaj C

Copyright 2001. Toate drepturile sunt rezervate autorului.

Multiplicarea acestui document în scop comercial este interzisă.

Sudenţii participanţi la orice formă de învăţământ superior bugetar pot multiplica acest document pentru uzul personal.

Sudenţii participanţi la orice formă de învăţământ cu plată sau alte persoane doritoare pot multiplica documentul numai contra cost şi cu acordul scris al autorului. În acest scop mă puteţi contactă la:

ANTAL Tiberiu Alexandru
tel.: 0040-264-530918
e-mail: antaltiberiu@pcnet.ro

Universitatea Tenhică din Cluj-Napoca
Facultatea Construcţii de Maşini
Catedra de Mecanică şi Programare
Curs de limbaj C

ANEXA 2 - Citirea declaraţiilor C

Declaraţiile folosite în curs au fost relativ simple: `int`, `float`, tablouri de `char`, structuri de tipuri de bază şi poantori la acestea. Citirea unei declaraţii C complexe se poate descompune în câţiva paşi elementari. Pentru a putea citi însă declaraţii complexe trebuie să cunoaştem regulile pe care compilatorul le foloseşte pentru interpretarea declaraţiilor. Etapele de parcurs pentru înţelegerea unei declaraţii sunt:

❶ Identificaţi numele variabilei care se declară;
❷ Rotiţi-vă în afară în sens trigonometric (antiorar, yang);
❸

La întâlnirea lui	Citiţi
*	`poantor la`
[ ]	`tablou de`
( )	`funcţie care are ... şi care întoarce ...`

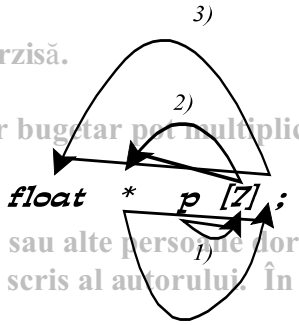
❹ Citiţi `struct S`, `union U` sau `enum E` dintr-o singură bucată;
❺ Citiţi colecţiile adiacente de [ ][ ] dintr-o singură bucată.

Conf. dr. ing. ANTAL Tiberiu Alexandru
Universitatea Tenhică din Cluj-Napoca
Facultatea Construcţii de Maşini
Catedra de Mecanică şi Programare
Curs de limbaj C

Exemplul 1

Fie declaraţia:

```
float *p[7];
```



Declaraţia de mai sus poate să specifice:

1) un tablou de `7` poantori la `float` sau
2) un poantor la un tablou de `7 float`.

Începem citirea cu numele variabilei care se declară, adică cu `p`. Ne rotim spre în afară în sens trigonometric şi găsim `[7]`. Acum putem spune deja `tablou de 7`. Continuând cu rotirea dăm peste * care se citeşte `poantor la`; pentru că avem deja 7 bucăţi vom spune mai precis `7 poantori la`, trecem mai departe printre] şi ; ajungând în final la `float`. Sistematizând cele citite mai sus, putem scrie:

`p` este un

1) `tablou de 7`

2) **poantori la**

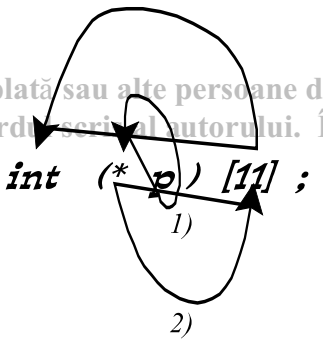
3) **float**

Deci, variabila **p** este un tablou care are 7 elemente, fiecare dintre acestea fiind un poantor la **float**.

Exemplul 2

Fie declaraţia:

```
int (*p) [11];
```



Diferenţa esenţială faţă de exemplul anterior este un set de paranteze rotunde în jurul lui ***p**. Deşi s-ar părea că efectul acestora este nesemnificativ, acestea vor modifica ordinea de parcurgere în spirală yang. Se începe cu **p** care este variabila declarată, se continuă spirala în interiorul parantezelor şi întâlnim un ***** care ne face să spunem **poantor la**. Apoi continuăm rotirea în spirală în afara parantezelor, întâlnim **[11]** şi vom putea spune **tablou de 11**. Apoi terminăm spirala la **int**. Sistematizând, putem spune:

p este

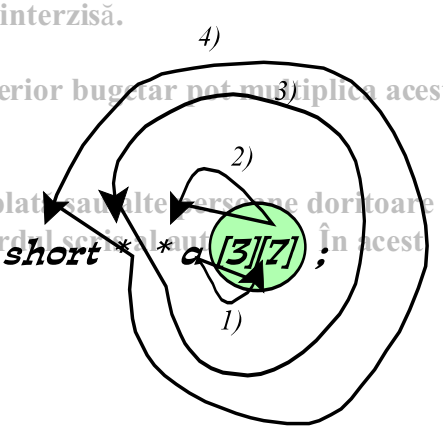
1) **un poantor la**
2) **un tablou de 11**
3) **int**

Deci variabila **p** este un singur poantor. La capătul poantorului, după iniţializare, se va afla un tablou de **11** întregi (**int**).

Exemplul 3

Fie declaraţia:

```
short **a[3][7];
```



Mai întâi găsim variabila care se declară, adică pe **a**. Apoi ne rotim în spirală yang şi găsim pe **[3][7]**, care se citeşte conform regulilor expuse dintr-o bucată, adică **tablou de 3 tablouri de 7**. Continuând rotirea găsim cel mai apropiat ***** de **a** şi zicem **poantor la**, apoi găsim celălalt ***** spunând din nou **poantor la**. La ultima spirală, ieşim între **]** şi **;** şi dăm peste **short**. Deci putem zice:

a este un

1) **un tablou 3 tablouri de 7**
2) **poantori la**

3) `poantori la`

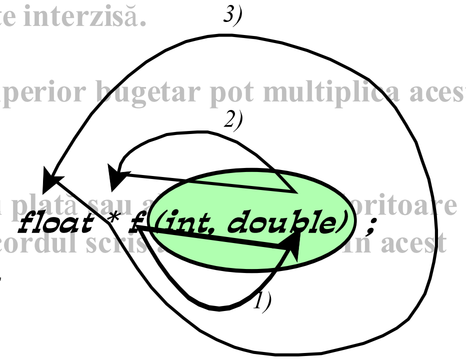
4) `short`

`a` este un tablou de `3` tablouri de `7` poantori la poantori la `short`.

Exemplul 4

Fie declaraţia:

```
float *f(int, double);
```



Începem cu spirala din `f`, întâlnim mai întâi `(int, double)` şi spunem `funcţie care are ca parametri de intrare un int şi un double şi întoarce` , la următoarea spirală găsim pe * şi spunem `poantor la`, apoi, în final, ajungem la `float`. Deci,

`f` este:

1) `o funcţie, care are ca parametri de intrare un int şi un float şi întoarce`

2) `un poantor la`

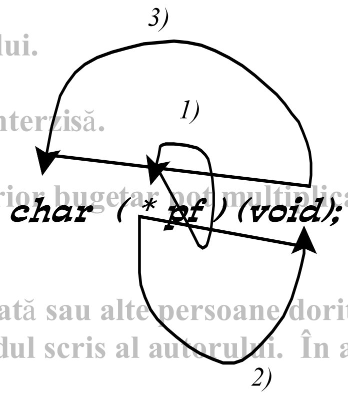
3) `float`

Avem de a face deci cu prototipul unei funcţii cu numele `f` care are ca parametri de intrare un `int` şi un `double` şi întoarce un poantor la un `long`.

Exemplul 5

Fie declaraţia:

```
char (*pf)(void);
```



Aici apare un set de paranteze rotunde în jurul lui `*pf`. Este declarată variabila `pf`; parantezele rotunde ne obligă să realizăm spirala yang în interioul lor întâlnind pe * pentru care citim `poantor la`. Apoi găsim pe `(void)` şi citim `funcţie care nu are parametri şi care întoarce`, continuând spirala ne oprim în final la `char`. Deci `pf` este:

1) `un poantor la o`

2) `funcţie care nu are parametri şi întoarce un`

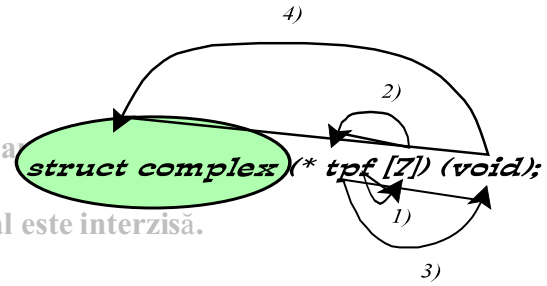
3) `char`

Astfel, `pf` nu este un prototip de funcţie, ci declaraţia unui singur poantor cara va stoca o funcţie. Un astfel de tip de poantor va putea poanta, de exemplu în segmentul de cod.

Exemplu 6

Fie declaraţia:

```
struct complex (*tpf[7]) (void);
```



Acest exemplu este imposibil de înţeles fără aplicarea regulilor. Începem cu `tpf`. Apoi parcurgând spirala yang, ajungem la `[7]` şi citim `tablou de 7`. Continuând găsim pe `*` şi citim `poantor la`, apoi ajungem la `(void)` citind `funcţie care nu are parametri şi întoarce`. Acum se ajunge la `struct complex`, care fiind un caz special, se citeşte aşa cum este scris. Punând toate cele zise cap la cap, avem:

`tpf` este un

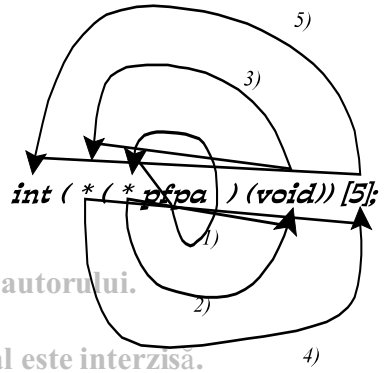
1) `tablou de 7`
2) `poantori la`
3) `funcţii care nu au parametru de intrare şi întorc`
4) `structuri complex`

`tpd` este un tablou de `7` poantori la funcţii care întorc prin valoare câte o structură `complex`. Fiecare funcţie întoarce câte o singură structură de tipul `complex`.

Exemplul 7

Fie declaraţia:

```
int (*(*pfpa)(void))[5];
```



Iată o declaraţie care probabil v-ar putea face să picaţi un examen. Totuşi, să aplicăm regulile deja prezentate. `pfpa` este variabila declarată. Realizăm parcurgerea în spirală din interiorul parantezelor, ceea ce ne conduce la `*`, citind `poantor la`. Parcurgerea în spirală ne conduce mai departe la `(void)`, citind `funcţie care nu are parametri şi întoarce`. În continuare se ajunge la `[5]` care se va citi `tablou de 5` şi terminăm cu `int`. Recapitulând, `pfpa` este:

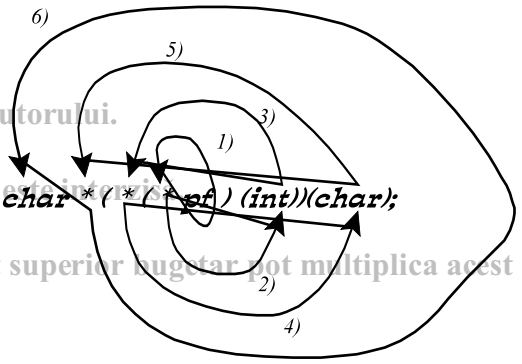
1) `un poantor la`
2) `o funcţie care nu are parametri de intrare şi care întoarce`
3) `un poantor la`
4) `un tablou de 5`
5) `int`

Astfel, `pfpa` declară un singur poantor. Restul declaraţiei se foloseşte numai pentru a specifica ce anume se află la capătul poantorului după iniţializare. Este un poantor la un cod, poantând la o funcţie fără parametri şi care întoarce un poantor ce poantează la un tablou de `5` valori `int`.

Exemplul 8

Fie declaraţia:

```
char *(*(* pf)(int))(char);
```



pf este un **poantor la**, datorită parantezelor rotunde şi a lui *, urmează **(int)** şi citim **funcţie care are un parametru de tipul int şi întoarce.** Continuând parcurgerea în spirală yang, întâlnim pe * cu semnificaţia **poantor la**, urmează apoi **(char)** şi citim **funcţie care are un parametru de tipul char şi întoarce.** Urmează încă un * citit **poantor la**, apoi în final **char.**

pf este

1) **un poantor la**
2) **o funcţie care are un parametru int şi întoarce un**
3) **poantor la**
4) **o funcţie care are un parametru char şi întoarce un**
5) **poantor la**
6) **char**

Astfel, **pf** este declaraţia unui singur poantor din nou. Restul declaraţiei serveşte numai la specificarea tipului poantat de poantor. Avem un poantor la o funcţie care are un parametru de tipul **int** şi care întoarce un poantor. Poantorul întors poantează la o altă funcţie care are ca parametru un **char** şi care întoarce o valoare tot de tipul **char**.

Înţelegerea declaraţiilor complexe poate fi mult uşurată prin folosirea lui **typedef**. Dacă dorim să simplificăm declaraţia **float* p[7];** care declară un tablou **p** de **7** poantori la **float**, vom începe întotdeauna de la capăt. Creăm un nou tip poantor la **float** cu numele de **ptf.** Tabelul care urmează prezintă câteva dintre variantele de folosire ale lui **typedef** pentru exemplele deja parcurse.

Declaraţie iniţială	cu **typedef**
float *p[7]; - tablou **p** de **7** (1) poantori la **float.**	(1) **typedef float * ptf;** **ptf p[7];**
short **a[3][7]; - **a** este un tablou de **3** tablouri de **7** (1) poantori la poantori la **short;** putem în continuare crea un (2) tablou de 3 poantori la poantori la **short**, după care crearea lui **a** va fi simplă.	(1) **typedef short ** pps;** **pps a[3][7];** sau (2) **typedef pps a3[3];** **a3 a[7];**
char (*pf)(void) - **pf** este un poantor la (1) o funcţie fără parametri.	(1) **typedef char f(void);** **f *pf;**

`struct complex (*tpf[7])(void);` - **`tpf`** este un tablou de **`7`** `(1)` poantori la `(2)` funcţii fară parametri care întorc structuri **`complex`**.	`(2)` **`typedef struct complex f(void);`** `(1)` **`typedef f *pf;`** **`pf tpf[7];`**
`int (*(*pfpa)(void))[5];` - **`pfpa`** este `(1)` un poantor la o `(2)` funcţie fără parametri care întoarce `(3)` un poantor la un tablou de **`6 int`**.	`(3)` **`typedef int (*pt6)[6];`** `(2)` **`typedef pt6 f(void);`** `(1)` **`typedef f *pfpa;`**
`char *(*(* pf)(int))(char);` - **`pf`** este un `(1)` poantor la o funcţie cu un parametru **`int`** care întoarce `(2)` un poantor la `(3)` o funcţie care are un parametru **`char`** care întoarce `(4)` un poantor la **`char`**.	`(4)` **`typedef char *ptc;`** `(3)` **`typedef ptc f(char);`** `(2)` **`typedef f *ptf;`** `(1)` **`ptf (*pf) (int);`**

Conf. dr. ing. ANTAL Tiberiu Alexandru
Universitatea Tenhică din Cluj-Napoca
Facultatea Construcţii de Maşini
Catedra de Mecanică şi Programare
Curs de limbaj C

Copyright 2001. Toate drepturile sunt rezervate autorului.

Multiplicarea acestui document în scop comercial este interzisă.

Sudenţii participanţi la orice formă de învăţământ superior bugetar pot multiplica acest document pentru uzul personal.

Sudenţii participanţi la orice formă de învăţământ cu plată sau alte persoane doritoare pot multiplica documentul numai contra cost şi cu acordul scris al autorului. În acest scop mă puteţi contactă la:

ANTAL Tiberiu Alexandru
tel.: 0040-264-530918
e-mail: antaltiberiu@pcnet.ro

Glosar

ASCII American Standard Code for Information Interchange este un standard pe 7 biţi pentru codificarea setului de caractere folosit de sistemele de calcul.

BIOS Basic Input Output System este o interfaţă software de nivel foarte jos cu hardware-ul unui PC. Este implementată în ROM BIOS şi încărcată la boot-area PC-ului.

Bit (*bit*) binary digit, denumirea unităţii de informaţie; în tehnica de calcul este cea mai mică unitate de stocare, suficientă pentru stocarea unui bit.

Boolean o variabilă care poate lua numai două valorile: **adevărat** sau **fals**. Denumirea vine de la **George Boole**, matematician britanic, care a creat un sistem matematic ce lucra numai cu două valori de adevăr.

Boot-are termenul vine din "to pull oneself up by one's bootstraps" şi semnifică procesul de încărcare şi iniţializare a sistemului de operare.

Buffer o zonă de memorie care asigură stocarea temporară a datelor pentru dispozitivele de I/E.

Byte (*octet*) o componentă în ierarhia de date a unui sistem de calcul, tipică pentru stocarea unui caracter (în sistemele de calcul pe 36 de biţi are 9 biţi, iar pe unele sisteme mai vechi, 6 sau 7 biţi).

Cache o zonă de memorie cu informaţie salvată pentru referiri ulterioare.

Device driver (*driver de dispozitiv*) un software folosit pentru comanda unui dispozitiv hardware.

FAT File Allocation Table este sistemul de fişiere nativ al sistemului de operare DOS.

GUI Graphical User Interface este o denumire utilizată pentru interfeţele grafice prin care utilizatorul poate comanda un sistem de calcul.

Interrupt (*întrerupere*)	un eveniment care suspendă procesul curent activ şi produce comutarea UCP pe un proces nou.
Interrupt driven	un sistem care este controlat prin întreruperi.
Interrupt handler (*rutină pentru tratarea întreruperii*)	un program care prelucrează întreruperi.
Library (**bibliotecă**)	o colecţie de rutine software disponibilă pentru toate programele.
Machine language (*limbaj maşină*)	limbaj cu instrucţiuni înţelese direct de UCP. Cel mai apropiat limbaj de cel maşină este limbajul de asamblare.
Macro sau **macrodefiniţie**	o referinţă simbolică la un grup de instrucţiuni sau date. Simbolul este înlocuit prin instrucţiuni sau valori actuale în faza de expandare a macrodefiniţiei.
Octet (*octet*)	8 biţi, termenul este folosit în domeniul reţelelor de calculatoare în locul lui **byte**, întrucât pe unele sisteme de calcul acesta nu are 8 biţi.
Type (*tip*)	o mulţime de valori din care o variabilă sau o constantă poate lua valori.

Conf. dr. ing. ANTAL Tiberiu Alexandru
Universitatea Tehnică din Cluj-Napoca
Facultatea Construcţii de Maşini
Catedra de Mecanică şi Programare
Curs de limbaj C

Copyright 2001. Toate drepturile sunt rezervate autorului.

Multiplicarea acestui document în scop comercial este interzisă.

Sudenţii participanţi la orice formă de învăţământ superior bugetar pot multiplica acest document pentru uzul personal.

Sudenţii participanţi la orice formă de învăţământ cu plată sau alte persoane doritoare pot multiplica documentul numai contra cost şi cu acordul scris al autorului. În acest scop mă puteţi contactă la:

ANTAL Tiberiu Alexandru
tel.: 0040-264-530918
e-mail: antaltiberiu@pcnet.ro