IB.

INTRODUCERE IN PROGRAMAREA CALCULATOARELOR

CUPRINS

Cuvânt inainte

Noțiunea de limbaj de programare este definită ca fiind ansamblul format de un vocabular şi un set de reguli gramaticale, care permit programatorului specificarea exactă a acțiunilor pe care trebuie să le execute calculatorul asupra unor date în scopul obținerii anumitor rezultate. Specificarea constă practic în întocmirea/scrierea programelor necesare ("programare").
Altfel spus, un limbaj de programare oferă o notație sistematică prin care poate fi descris un proces de calcul. Notația constă dintr-un set de reguli sintactice şi sematice. Sintaxa reprezintă un set de reguli ce guvernează alcătuirea propozițiilor dintr-un limbaj. În cazul limbajelor de programare echivalentul propoziției este programul. Semantica este un set de reguli ce determină „înțelesul" sau semnificația propozițiilor într-un limbaj.

Putem defini două mari categorii de limbaje de programare:

1. **Limbaje de nivel coborât, dependente de calculator. Aici avem:**
   - **Limbajul maşină**
   - **Limbajul de asamblare**

Limbajul maşină este limbajul pe care calculatorul îl înțelege în mod direct; în acest limbaj programele se scriu în cod binar ca succesiuni de 0 şi 1, fiecare instrucțiune din limbajul maşină fiind o combinație de 4 biți (exemple: 0000, 0001). Pentru aceasta programatorul trebuie să cunoască detaliat structura hardware a calculatorului, trebuie să gestioneze fără greşeală alocarea adreselor de memorie pentru un program. Pot apărea multe erori datorate concepției programului, sintaxei, suprapunerii adreselor de memorie, etc.

Limbajele de asamblare introduc cuvinte cheie pentru desemnarea operațiilor (de exemplu: LOAD pentru operația cu codul binar 0000, ADD pentru operația cu codul 0001, etc) precum şi simboluri pentru adrese (exemplu....), simplificând astfel programarea. Pentru execuția unui program scris în limbaj de asamblare este necesară o fază preliminară prin care programul este transformat într-unul echivalent în limbaj maşină. Transformarea este realizată automat de un program numit assembler (asamblor). Asamblorul înlocuieşte codarea mnemonică (cum ar fi ADD) cu coduri binare corespunzătoare limbajului maşină şi alocă adrese de memorie pentru toate variabilele simbolice utilizate (A, B, C). Astfel, limbajele de asamblare uşurează procesul de programare dar sunt la fel de apropiate de hardware ca şi limbajele maşină. În prezent, limbajele de asamblare sunt utilizate pentru unele programe critice, care necesită controlul exact al resurselor hardware ale calculatorului (procesorul central şi memoria internă).

Limbaje de nivel înalt, independente de structura calculatorului. Câteva exemple în ordinea apariției lor:

- **Fortran (FORmula TRANslation) – 1955, IBM, pentru probleme tehnico-ştiințifice**
- **Cobol – 1959, pentru probleme economice**
- **Pascal, C, s.a. – anii 1970, odată cu apariția conceptelor de Programare structurată**
- **C++, Java, s.a. – anii 1980, odată cu apariția conceptelor de Programare orientată pe obiecte**

Programarea structurata se bazeaza pe teorema programării structurate (structured program theorem) a lui Böhm şi Jacopini, pe care am folosit-o deja în elaborarea algoritmilor. Această teoremă spune că orice algoritm poate fi compus din numai trei structuri de calcul:

1. **structura secvențială - secvența;**
2. **structura alternativă - decizia;**
3. **structura repetitivă - ciclul.**

Bazele programării structurate au fost puse de Dijkstra şi Hoare. Structura unui program se obțține printr-o abordare "top-down" (de regulă) şi orientată pe prelucrări: o problemă care presupune o prelucrare complexă este descompusă în subprobleme/prelucrări mai simple; fiecare subproblemă poate fi descompusă la rândul său în prelucrări şi mai simple, până când se ajunge la un nivel de complexitate coborât, la care fiecare prelucrare obținută este descrisă printr-o unitate program (funcție, procedură). Tehnicile de programare structurată pot fi aplicate în majoritatea limbajelor de programare, dar ele sunt adecvate limbajelor de programare procedurală (în care unitatea program este procedura/funcția) cum sunt Pascal, C şi altele.

Programarea orientata pe obiecte introduce ideea structurării programelor în jurul obiectelor. Fiecare obiect aparține unei clase de obiecte care este descrisă în cadrul unui program. Toate obiectele dintr-o clasă au aceeaşi structură (descrisă prin variabile şi constante) şi un acelaşi comportament (descris prin operații). Obiectele sunt entități dinamice, care apar, interacționează cu alte obiecte (prin intermediul operațiilor) şi dispar, in timpul execuției programului.
Obiectele din programele cu structură orientată obiect sunt, de obicei, reprezentări ale obiectelor din viața reală, astfel încât programele realizate prin tehnica POO sunt mai uşor de înțeles, de testat şi de extins decât programele procedurale. Această constatare este adevărată mai ales în cazul sistemelor software complexe şi de dimensiuni mari, a căror dezvoltare trebuie să fie ghidată de principii ale Ingineriei Programelor (Software Engineering).

După modul de transformare a programelor ("translatare") în vederea execuției pe un calculator, limbajele de programare de nivel înalt pot fi împărțite în:

- Limbaje compilate: C, C++, Pascal, Java;
- Limbaje interpretate: PHP, Javascript, Prolog, Matlab

La limbajele compilate translatorul se numeşte compilator; acesta transformă programul sursa (scris în limbjul de programare de nivel înalt) într-un program exprimat în limbajul maşină, rezultatul fiind un fişier executabil. Viteza de execuție a programului compilat este mare, întrucât programul este deja transpus în întregime în cod maşină.
La limbajele interpretate translatorul poartă denumirea de interpretor şi funcționează în felul următor: preia prima comandă din codul sursă, o traduce în limbajul maşină şi o execută, apoi a doua comandă şi tot aşa. De aceea, viteza de executie a unui program interpretat este mult mai mica decat a unui program compilat.
Multe limbaje moderne combină compilarea cu interpretarea: codul sursă este *compilat* într-un limbaj binar numit *bytecode*, care la executie este *interpretat* de către o maşină virtuală. De remarcat faptul că unele interpretoare de limbaje pot folosi compilatoare aşa-numite *just-in-time*, care transformă codul în limbaj maşină chiar înaintea executării.

Capitolul IB.01. Rezolvarea algoritmică a problemelor

Cuvinte-cheie
Algoritm, date, constante, variabile, expresii, operații, schemă logică, pseudocod

IB.01.1. Introducere în programare

Să considerăm următorul exemplu. Se defineşte funcția F(x), unde x este număr real, astfel:

$$F(x)=\begin{cases} x^2-2, & x<0 \\ 3, & x=0 \\ x+2, & x>0 \end{cases}$$

Se cere să se scrie un program care să calculeze valoarea acestei funcții pentru următoarele 100 de valori ale lui x: x = {-3, 0,1,7, 2.23, etc} – 100 valori.
Pentru a putea rezolva această cerință trebuie să vedem mai întâi care sunt etapele rezolvării unei probleme utilizând un program informatic.

Etapele rezolvării unei probleme utilizând un program informatic:

1. **Formularea clară a problemei:**
   - **date disponibile**
   - **prelucrări necesare**
   - **rezultate dorite**
2. **Elaborarea algoritmului ce implică analiza detaliată a problemei:**
   - **date: sursa (consola/ suport magnetic/…), semnificație, tip (numeric/ alfanumeric), restricții asupra valorilor**
   - **rezultate: destinație (ecran/imprimantă/suport magnetic /…), mod de prezentare**
   - **principalele etape de rezolvare (schemă logică sau pseudocod)**
   - **eventuale restrictii impuse de limbajul de programare în care vom transpune algoritmul**
3. **Transpunerea algoritmului în limbajul de programare utilizat**
4. **Rulare şi ... din nou etapa 2 dacă nu am obținut ceea ce trebuia.**

În cadrul acestui capitol ne vom ocupa de primele două etape, şi anume cea de formulare a problemei şi cea de elaborare a algoritmului, pentru ca pe parcursul următoarelor capitole să detaliem modalitățile de implementare a programelor utilizând limbajul de programare C.
Să revenim acum la problema noastră şi să parcurgem prima etapă, cea de formulare clară a problemei:

- datele disponibile sunt cele 100 de valori de intrare x
- prelucrări necesare sunt calculul lui F(x) pentru cele 100 de valori ale lui x
- rezultatele dorite sunt cele 100 de valori ale lui F(x) calculate prin prelucrări.

În acest moment putem spune că ştim foarte bine ce avem de făcut. Urmează să vedem mai departe cum anume facem aceasta.
Pentru a trece la etapa a doua a rezolvării problemei nostre vom detalia în cele ce urmează noțiunea de algoritm precum şi obiectele acestuia.

IB.01.2. Algoritm

Algoritm - succesiune de etape de calcul ce se poate aplica pentru rezolvarea unei clase de probleme.

Cerințele pe care trebuie să le îndeplinească un algoritm sunt următoarele:

- **Claritate – să nu existe ambiguități in descrierea etapelor de calcul**
- **Generalitate – să poată fi aplicat pentru o clasă de probleme si nu pentru o problema particulara (de exemplu, algoritmul pentru rezolvarea ecuatiilor de gradul 2 trebuie sa descrie modul de rezolvare a oricarei ecuatii de gadul 2 si nu a unei ecuatii particulare de gradul 2).**
- **Finitudine – să furnizeze rezultatul în timp finit**

Descrierea unui algoritm poate fi efectuată utilizând:

- **Scheme logice**
- **Pseudocod**

În momentul în care vom căpăta suficientă experiență în programare iar problema de rezolvat nu este foarte complexă putem să ne reprezentăm mental algoritmul ei de rezolvare. Totuşi, în fazele de început sau pentru un algoritm mai complex este foarte indicat să schițăm algoritmul de rezolvare a unei probleme înainte de implementarea rezolvării într-un limbaj de programare. Se practică descrierea algoritmului fie sub formă grafică (organigrame sau scheme logice), fie folosind un "pseudocod", ca un text intermediar între limbajul natural şi un limbaj de programare.
O problemă poate avea mai mulți algoritmi de rezolvare. Cum îl alegem pe cel mai bun şi ce înseamnă cel mai bun algoritm? Pentru a răspunde la această întrebare se va analiza eficiența algoritmului (timpul de execuție, memoria internă necesară,alte resurse de calcul necesare) şi se va alege, dintre algoritmii identificați cel mai eficient (timp minim de executie, resurse de calcul minime), ținând cont şi de scopul şi natura problemei rezolvate (de exemplu, timpul minim de execuție poate fi mai important decat dimensiunea memoriei interne necesare). Această etapă este o etapă ce implică o analiză complexă a algoritmilor pe care deocamdată, pe parcursul acestui prim curs de programare, nu o vom lua decât arareori în considerare.

IB.01.3. Obiecte cu care lucrează algoritmii

Date

Principalele obiecte ale unui algoritm sunt datele. Ele pot fi:

- **Date de intrare - care sunt cunoscute**
- **Date de ieşire - rezultate furnizate**

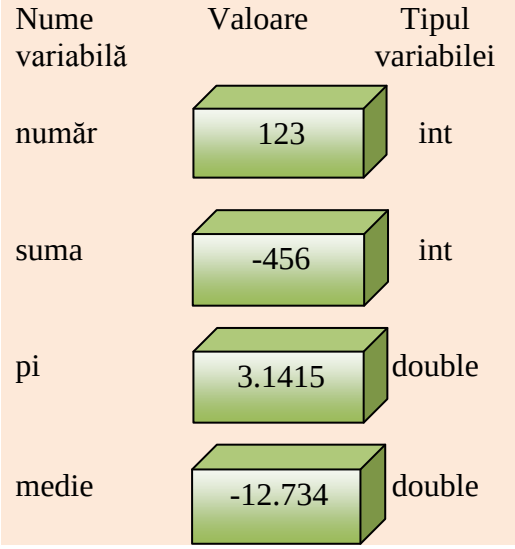
După tipul lor, datele pot fi:

- **Întregi: 2, -4**
- **Reale: 3.25, 0.007**
- **Logice: true şi false – adevărat şi fals**
- **Caracter: 'y', 'a'**
- **Şir de caractere: "ab23_c"**

Constante
În descrierea unui algoritm pot apare **constantele**; acestea sunt date conținute în program, care nu sunt citite sau calculate. Un exemplu este constanta π din matematică.

Variabile
Programele, şi implicit algoritmii, lucreaza cu date. O **variabilă** este utilizată pentru a stoca (a păstra) o dată. Se numeşte **variabilă** pentru că valoarea stocată se poate schimba pe parcursul execuției algoritmului. O variabilă are un nume unic şi un conținut care poate să difere de la un moment la altul al execuției algoritmului. Mai precis, o variabilă este o locație de memorie care are un nume şi care păstrează o valoare de un anumit tip.

O *variabilă* este caracterizată prin:
- **Nume**
- **Tip**
- **Valoarea la un moment dat**
- **Locul în memorie (adresa)**



Orice variabilă are un nume, conține o valoare declarată de un anumit tip, valoare memorată mereu la o aceeaşi adresă de memorie

De exemplu, în problema propusă anterior, putem păstra valorile datelor de intrare (1.5, 3.6, 4.2, etc) într-o variabilă numită x, care va lua pe rând fiecare dintre datele de intrare. Variabila x este de tip real, se află în memorie, de exemplu la adresa 0xFF32 (adresă care rămâne fixă)- care însă nu are importanță pentru un programator ăncepator- şi poate avea valoarea 3.6 la un moment dat.
În mod analog, rezultatele (F(1.5), F(3.6), F(4.2), etc) le vom stoca într-o variabilă F, tot de tip real.

Expresii
Expresiile sunt construite utilizând constante, variabile şi operatori. Ele pot fi de mai multe tipuri, la fel ca şi datele. Exemplu 3*x+7, x<y, etc.
Operatorii sunt: matematici (+, - , *, /, mod – restul împărțirii întregi), relaționali (<, >, <=, >=, != - diferit, == - comparație la egalitate) şi logici (şi – ambele adevărate, sau – cel puțin una adevărată, not – negarea unei expresii).

IB.01.4. Etapele specifice unui algoritm şi fluxul de execuție al acestora

Un algoritm poate efectua operații de:

- **Intrare: preluarea unei date de la un dispozitiv de intrare (consola, suport magnetic, etc)**
- **Ieşire: transferul unei date din memoria internă către un dispozitiv de ieşire (ecran, imprimanta, suport magnetic, etc)**
- **Atribuire: x=3; y=x; y=x+y (variabila ia valoarea(=) expresiei)**
  - **Operația de atribuire se realizează astfel:**
    - **Se evaluează expresia din partea dreapta a semnului =**
    - **Valoarea obținută este atribuită variabilei din stânga (stocată în locația sa de memorie), care îşi pierde vechea valoare**
- **Decizie: o întrebare ridicată de programator la un moment dat în program, operație prin care, în funcție de valoarea curentă a unei condiții, se alege următoarea etapă a algoritmului**

În programarea structurată apare teorema de structură a lui Bohm şi Jacopini:

Orice algoritm poate fi compus din numai trei structuri de calcul:

1. **structura secvențială - secvența;**
2. **structura alternativă - decizia;**
3. **structura repetitivă - ciclul.**

Secvența este cea mai întâlnită, în cadrul ei instrucțiunile sunt executate în ordinea în care sunt scrise, de sus în jos, secvențial.

Decizia implică o întrebare ridicată de programator la un moment dat în program. In funcție de răspunsul la întrebare - care poate fi ori "Da", ori "Nu" - programul se continuă pe una din ramuri, executându-se blocul corespunzător de operații.

Ciclul exprimă un calcul (compus din una sau mai multe etape) care poate fi executat de mai multe ori. Numărul de execuții este controlat de valoarea unei expresii care se evaluează fie înainte fie după execuția calculului.
De exemplu, să presupunem că vrem să calculăm funcția F(x) din exemplul anterior pentru toate numerele întregi de la 1 la 1000. Pentru aceasta ar trebui să folosim o *structură repetitivă* pentru a descrie operațiile care trebuie executate în mod repetat:citirea unei date de intrare, evaluarea funcției pentru acea data de intrare şi transferul valorii functiei la un dispozitiv de ieşire. Iată, de exemplu, la ce sunt bune calculatoarele! Totuşi, ține de priceperea noastră să scriem algoritmi ce conțin structuri repetitive care să poată uşura foarte mult rezolvarea unei clase întregi de probleme; de exemplu, evaluarea funcției F(x) nu doar pentru o valoare particulară, ci pentru orice număr real şi pentru oricâte numere!!

În programarea structurată sunt definite trei structuri repetitive:

1. **Structura repetitivă cu conditie inițială, formată din:**
   - **O condiție, definită la începutul structurii**
   - **Un bloc de instrucțiuni, care se execută dacă rezultatul evaluării condiției este adevărat. Evaluarea condiției are loc <u>înainte</u> de execuția blocului de instrucțiuni, de aceea, dacă rezultatul primei evaluări a condiției este fals, blocul de instrucțiuni nu este executat (nici o dată).**

Se mai numeşte şi structură repetitivă de tip *while.*

2. Structura repetitivă cu condiție finală, în care condiția este definită după blocul de instrucțiuni

Condiția este evaluată <u>după</u> fiecare execuție a blocului de instrucțiuni. De aceea, blocul de instrucțiuni se execută cel puțin o dată.

Se mai numeşte şi structură repetitivă de tip *do-while.*

3. Structura repetitivă cu contor

Caz particular al structurii repetitive cu condiție inițială.

Condiția este exprimată folosind o variabilă cu rol de contor (numărător de repetări).

Se definesc, pentru variabila contor:

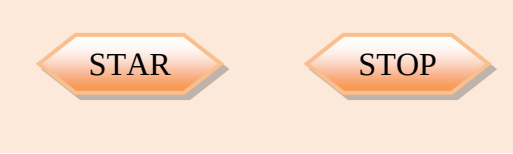
- **Valoarea inițiala (înaintea primei execuții a blocului de instrucțiuni);**
- **Valoarea finală(corespunzătoare ultimei execuții a blocului de instrucțiuni);**
- **Pasul: valoarea care se adaugă la valoarea variabilei contor după fiecare execuție a blocului de instrucțiuni.**

Se mai numeşte şi structură repetitivă de tip *for.*

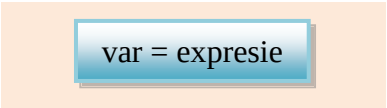
IB.01.5. Scheme logice

Schemele logice sunt reprezentări grafice ale algoritmilor. Fiecărei operații îi corespunde un simbol grafic:

- **Start/Stop**: marchează începutul/ sfârşitul schemei logice (execuției algoritmului)



- **Atribuirea**: operația prin care unei variabile i se atribuie o valoare.



Atribuirea:
Se evaluează expresia, apoi se atribuie valoarea expresiei variabilei din stânga semnului =

- **Operația de intrare (citire)**: programatorul preia de la tastatură una sau mai multe valori (intrări)

Programul preia de la un dispozitiv (extern, de exemplu tastatura) una sau mai multe valori (date de intrare), pe care le atribuie in ordine variabilelor din lista specificată în operația de citire.

- **Operația de ieşire (scriere)**: programul transmite la un dispozitiv (extern, de exemplu, ecran sau imprimantă) valorile variabilelor/expresiilor specificate in operația de scriere.

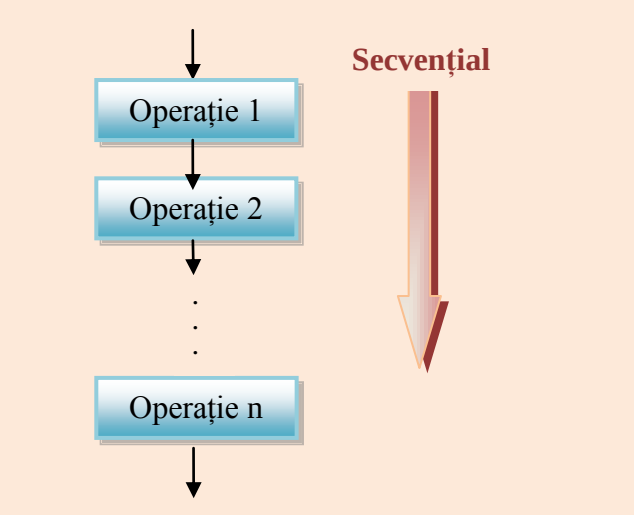
Operația de scriere (denumită şi operația de ieşire) presupune evaluarea în ordine a expresiilor specificate şi afişarea pe ecran a valorilor lor pe aceeaşi linie.

- **Acțiuni nedetaliate (bloc de operații)**: un bloc de operații nedetaliat

Se execută operația specifică blocului, fără ca această operație să fie detaliată. Este utilizat în general pentru operații mai complexe, pentru a nu încărca schema logică principală. Acest bloc va fi detaliat după realizarea schemei logice principale.

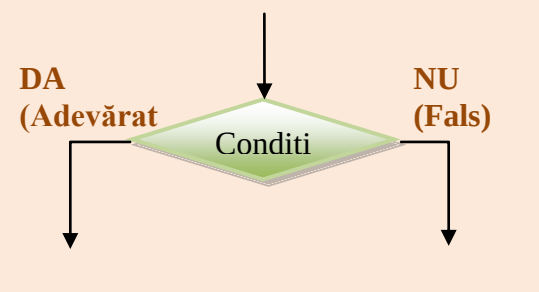
Exemplu: dacă vrem să afişăm primele n numere prime, putem avea un bloc de operatii nedetaliat care testează dacă un număr este prim sau nu.

- **Secvența** se reprezintă prin simboluri grafice conectate prin săgeți ce sugerează fluxul operațiilor (secvențial):



Se execută în ordine *Operație 1*, apoi *Operație 2*, ş.a.m.d. până la *Operație n*.

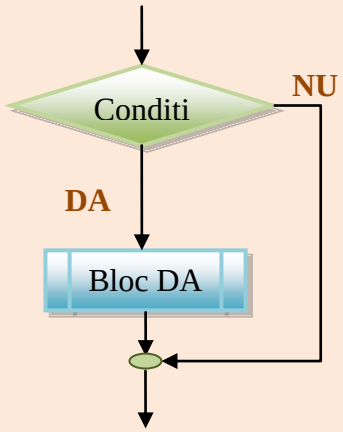
- **Decizia**



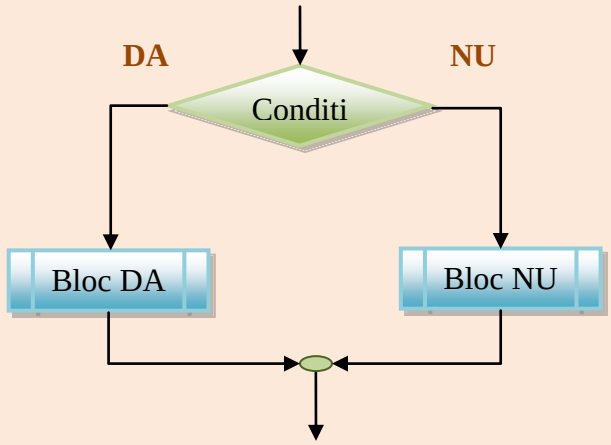
Se evaluează Condiție (o expresie cu valoare logică);

- **Dacă valoarea Condiției este adevărat, atunci execuția se continuă pe ramura DA;**
- **Dacă valoarea Condiției este fals, se continuă execuția pe ramura NU.**

Decizia poate avea una din următoarele forme:



Forma 1



Forma 2

Forma 1:
Se evaluează Condiție;

- **Dacă valoarea Condiției este adevărat, atunci se execută Blocul DA;**
- **Dacă valoarea Condiției este fals, executia se continuă cu operația care urmează imediat după blocul DA.**

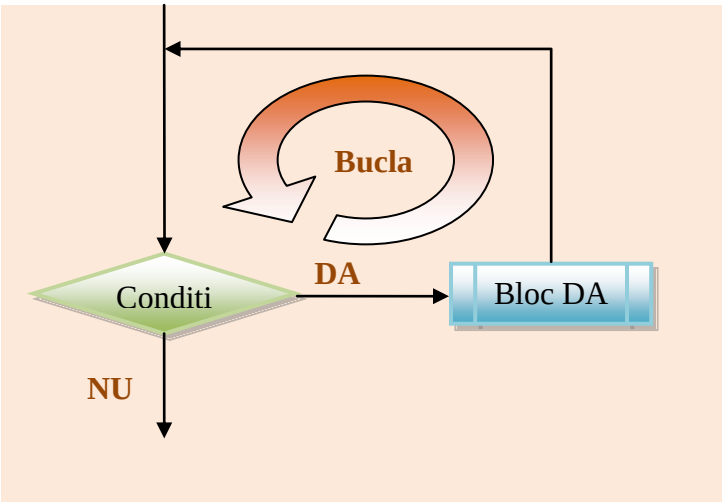
Forma 2:
Se evaluează Condiție;

- **Dacă valoarea Condiției este adevărat se execută Blocul DA;**
- **Dacă valoarea Condiției este fals se execută Blocul NU.**

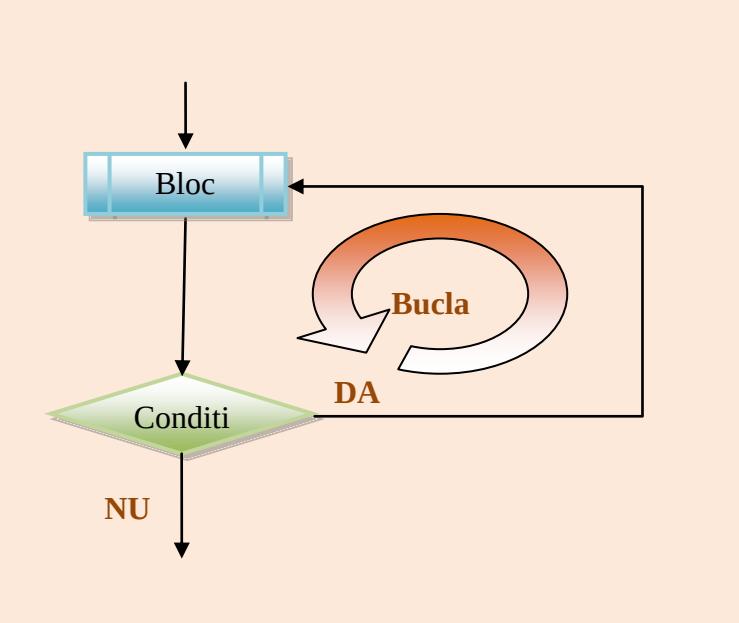
- **Structura repetitivă cu condiție inițială (structură repetitivă de tip *while*)**

Pas 1 : se evaluează *Conditie* ;
Pas 2:

- **dacă valoarea *Conditiei* este fals (NU), se iese din structura repetitivă;**
- **dacă valoarea expresiei este adevărat (DA), se execută Bloc DA , apoi se reia execuția de la Pas 1**

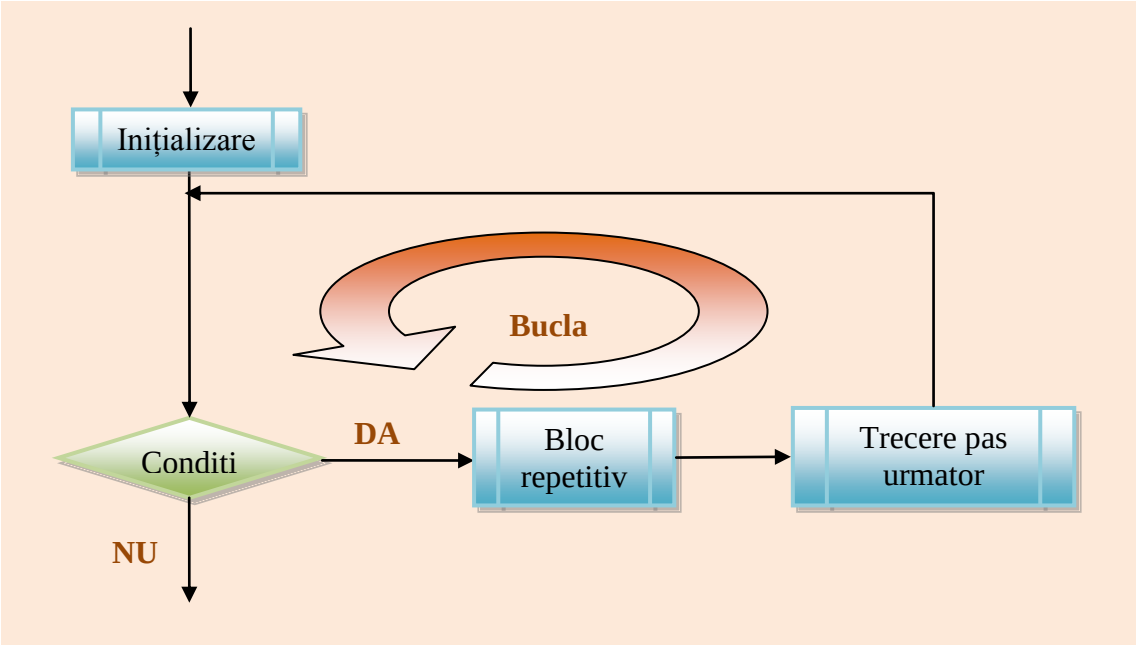


- **Structura repetitivă cu condiție finală (structură repetitivă de tip *do-while*)**



Pas 1 : se execută instrucțiunea sau instrucțiunile din *Bloc*;
Pas 2: se evaluează *Conditie*;
Pas 3: dacă valoarea *Conditiei* este fals (NU), se iese din instrucțiunea repetitivă;
dacă valoarea expresiei este adevărat (DA) se reia de la Pas 1

- **Structura repetitivă cu contor (structură repetitivă de tip *for*)**



- **Pas 1: se execută instrucțiunea sau instrucțiunile din blocul *Inițializare*. În general, aceasta înseamnă atribuirea unei valori inițiale unei variabile contor, folosită pentru numărarea (contorizarea) execuțiilor repetitive efectuate. Fie această valoare inițială *expresie_1*;**
- **Pas 2: se evaluează *Conditie*;**
  - **În general, această condiție (care poate fi dată sub forma unei expresii) verifică dacă valoarea variabilei contor este mai mică decât valoarea corespunzătoare ultimei execuții a Blocului repetitiv. Dacă valoarea expresiei *Condiție* este fals, atunci se iese din structura repetitivă.**
  - **dacă valoarea expresiei *Condiție* este adevărat, atunci :**
    - **se execută *Blocul repetitiv***
    - **se execută blocul *Trecere la pas următor* (care de obicei constă în modificarea valorii variabilei contor cu o valoare specificata)**
    - **se reia execuția de la Pas 2.**

IB.01.6 Exemple de algoritmi reprezentați în schemă logică

Problema 1:
Se defineşte funcția F(x), unde x este număr real, astfel:

$$F(x)=\begin{cases} x^2-2, & x<0 \\ 3, & x=0 \\ x+2, & x>0 \end{cases}$$

Să se scrie un program care calculează valoarea acestei funcții pentru 100 de valori ale lui x.

Rezolvare:
A două etapă a rezolvării acestei probleme este elaborarea algoritmului ce implică analiza detaliată a problemei:
Date:

- **x** - valoare reală preluată de la consolă

Rezultate:

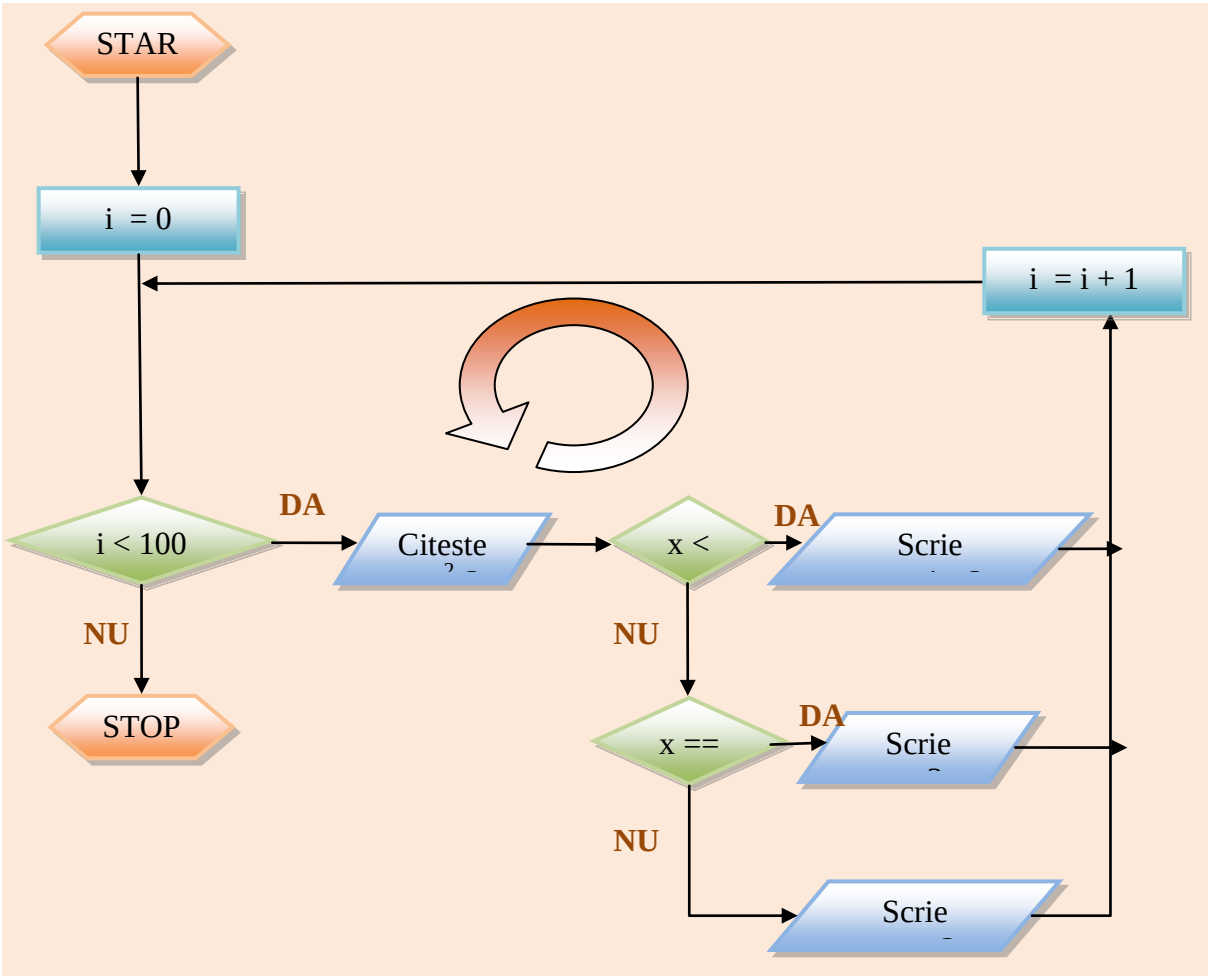
- ***y*** - valoare reală care va fi afişată pe ecran (însoțită de un text explicativ)

Principalele etape de rezolvare

- citirea datelor (se presupun corecte)
- calculul valorii lui y în funcție de intervalul în care se încadrează valoarea x citită
- repetăm aceste etape pentru cele 100 de valori ale lui x.

Pentru a şti câte valori am citit vom folosi o variabilă numită *contor* (deoarece contorizează/numără). Este o variabilă întreagă şi o vom numi **i**. Ea porneşte cu valoarea inițială 0 (Atenție! Să nu uităm acest pas, este important!) şi la fiecare valoare citită îi vom mări valoarea cu 1. Vom continua citirile atâta timp cât **i** va fi mai mic decât 100.

Urmează **schema logică** ce descrie în detaliu algoritmul propus:



Problema 2: Actualizarea unei valori intregi cu un procent dat.
Să se actualizeze o valoare naturală cu un procent dat.

Etapele elaborării algoritmului sunt :

1. Formularea problemei:

- date:
  - valoare curenta (***v***)
  - procent actualizare (***p***)
- prelucrare:
  - calculul valorii ***actualizate***
- rezultat:
  - valoarea calculată (***r***)

2. Analiza detaliată

Date:

- 2 valori preluate de la consolă:
- ***v*** - valoare întreagă, strict pozitivă
- ***p*** - valoare reala, pozitivă (majorare) sau negativă (diminuare)

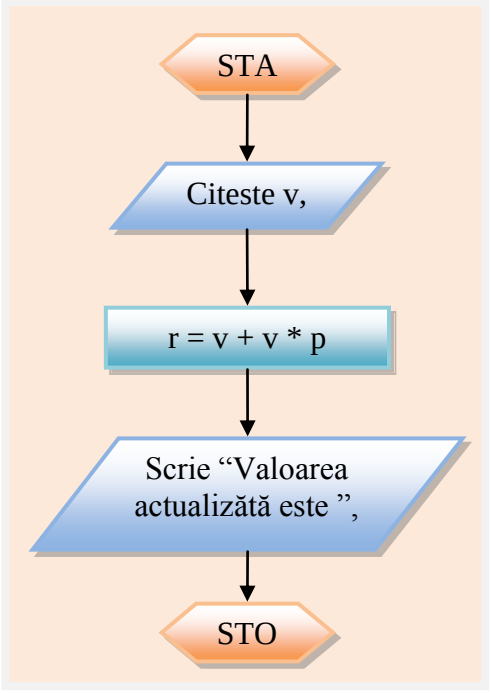
Rezultat:

- ***r*** - valoare reală, strict pozitivă, afişată pe ecran (însoţită de un text explicativ)

Principalele etape de rezolvare

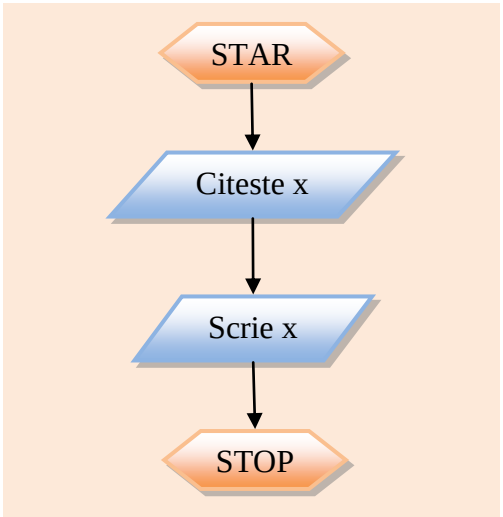
- citirea datelor (se presupun corecte)
- calculul valorii actualizate cu formula **v + v * p** sau **v * (1 + p)**
- afişarea rezultatului

Urmează **schema logică** ce descrie în detaliu algoritmul propus:



Pentru problemele care urmează vom oferi doar schema logică ce descrie algoritmul.

Problema 3: Una dintre cele mai simple probleme este citirea şi scrierea unei valori reale.
Date de intrare: x variabilă reala
Date de ieşire: acelaşi x.
În acest caz rezultatul este chiar data de intrare.



IB.01.7. Pseudocod

Un pseudocod este un text intermediar între limbajul natural şi un limbaj de programare. Are mai puține reguli decât un limbaj de programare şi descrie numai operațiile de prelucrare (nu şi variabilele folosite). Nu există un pseudocod standardizat sau unanim acceptat. De asemenea, descrierea unor prelucrări în pseudocod se poate face la diferite niveluri de detaliere.
Vom descrie în continuare operațiile unui algoritm într-un posibil pseudocod pe care îl vom folosi în continuare (tabel):

Operația	Pseudocod
Start	start
Terminare	stop.
Atribuire	variabila=valoare;
Citire	citeste var;
Scriere	scrie var;
Decizie	daca *conditie* instructiuni_1; altfel instructiuni_2;
Structura repetitivă cu conditie inițială - *while*	atata timp cat *conditie* instructiuni;
Structura repetitivă cu condiție finală - *do*	executa instructiuni; atata timp cat *conditie*
Structura repetitivă cu contor - *for*	pentru *contor* de la *val_initiala* la *val_finala* cu pasul *pas* instructiuni;

Vom face următoarea convenție: liniile ce conțin instrucțiuni care se execută în interiorul unciclu sau a unei decizii vor fi indentate (scrise deplasat în interior cu cateva spații) față de linia pe care începe ciclul sau decizia de care aparțin, pentru a pune în evidență execuția acestora în cadrul ciclului (deciziei).
O altă posibilitate de a marca, şi mai puternic, un bloc de instrucțiuni este aceea de a delimita blocul utilizând acolade:
{
instrucțiune 1
instrucțiune 2
….
instrucțiune 3
}
Această ultimă convenție este utilizată şi în limbajul C.
O altă observație legată de notațiile din tabelul anterior este legată de simbolul punct şi virgulă „**;**”, care apare la sfârşitul instrucțiunilor. În limbajul C prezența acestuia este obligatorie, de aceea, pentru a uşura trecerea de la pseudocod la C l-am introdus şi în pseudocod, fără însă ca aici prezența lui să fie neapărat obligatorie. Totuşi, în cele exemplele ce urmează va fi folosit.

IB.01.8 Exemple de algoritmi descrişi în pseudocod

Problema 1: Calculul funcției F(x)

```
start
```

```
pentru i de la 0 la 100 cu pasul 1
                    citeste x;
                    daca x < 0
                              scrie x * x – 2;
                    altfel
                      daca x = 0
                                        scrie 3;
                       altfel scrie x + 2;
stop.
```

Problema 2: Actualizarea unei valori naturale cu un procent dat

```
start
citeste v, p;
r = v + v * p;
scrie r;
stop.
```

Problema 3: Citirea şi scrierea unei valori.

```
start
citeste x;
scrie x;
stop.
```

Problema 4: Rezolvarea ecuației de grad 1: ax+b=0

```
start
citeste a,b;
daca a = 0
                    daca b = 0
                              scrie “Ecuatia are o infinitate de solutii”;
                    altfel
                      “scrie Ecuatia nu are nici o solutie”;
altfel
          x = -b / a;
          scrie x;
stop.
```

Problema 5: Să se afişeze suma primelor n numere naturale, n citit de la tastatură.

```
start
citeste n;
s = 0;
pentru i de la 0 la n cu pasul 1
```

```
                s = s + i;
scrie s;
stop.
```

Problema 6: Algoritmul lui Euclid care determină cel mai mare divizor comun a doi întregi, prin împărțiri repetate:

```
start
citeste a,b;
r = a mod b;
atata timp cat r > 0
        a = b;
        b = r;
        r = a mod b;
  scrie b;
  stop.
```

Observație: operatorul *mod* întoarce restul împărțirii întregi a lui a la b.

Capitolul IB.02. Introducere în limbajul C. Elemente de bază ale limbajului

Cuvinte-cheie
Limbajul C, istoric, caracteristici, compilare, rulare, main, alfabet,
atomi lexicali, identificatori, cuvinte cheie, tipuri de date,
constante, variabile, comentarii, operatori, expresii, conversii de tip

IB.02.1 Limbajul C - Scurt istoric

Limbajul C a fost proiectat şi implementat de Dennis Ritchie între anii 1969 şi 1973 la AT&T Bells Laboratories, pentru programe de sistem (programe care gestionează direct resursele hardware ale calculatorului, de exemplu, sistemul de operare), care până atunci erau dezvoltate doar în limbaje de asamblare. A fost numit "C" deoarece este un succesor al limbajului B, limbaj creat de Ken Thompson.
Originile limbajului C sunt strâns legate de cele ale sistemului de operare UNIX, care inițial fusese implementat în limbajul de asamblare PDP-7 tot de Ritchie şi Thompson. Deoarece limbajul B nu putea folosi eficient unele din facilitățile sistemului de calcul PDP-11, pe care vroiau să porteze UNIX-ul, au avut nevoie de un limbaj simplu pentru scrierea nucleului sistemului de operare UNIX.

- În 1973, sistemul de operare UNIX este aproape în totalitate rescris în C, fiind astfel unul din primele sisteme de operare implementate în alt limbaj decât cel de asamblare.
- Cartea de referință care defineşte un standard minim al limbajului este scrisă de Brian W. Kernighan şi Dennis Ritchie, "The C Programming Language" şi a apărut în 1978 .
- Între 1983-1989 a fost dezvoltat un standard international -- ANSI C (ANSI - American National Standards Institute).
- În 1990, ANSI C a fost adoptat de International Organization for Standardization (ISO) ca ISO/IEC 9899:1990, care mai este numit şi C90. Acest standard este suportat de compilatoarele curente de C. Majoritatea codului C scris astăzi are la bază acest standard. Orice program scris doar în ANSI C va rula corect pe orice platformă ce are instalată o variantă de C.
- În anii următori au fost dezvoltate medii de programare C performante sub UNIX şi DOS, care au contribuit la utilizarea largă a limbajului.
- Necesitatea grupării structurilor de date cu operațiile care prelucrează respectivele date a dus la apariția noțiunilor de o*biect* si *clasă*. In 1980, Bjarne Stroustrup elaborează "*C with Classes"*.
- Acest limbaj a dus la îmbunătățirea C-ului prin adăugarea unor noi facilități, printre care şi lucrul cu clase. În vara anului 1983, "C-with-classes" a pătruns şi în lumea academică şi a instituțiilor de cercetare. Astfel, acest limbaj a putut să evolueze datorită experienței acumulate de către utilizatorii săi. Denumirea finală a acestui limbaj a fost *C++*.
- Succesul extraordinar pe care l-a avut limbajul C++ a fost asigurat de faptul că a extins cel mai popular limbaj al momentului, C.
- Programele scrise în C funcționează şi în mediile de programare C++, şi ele pot fi transformate în C++ cu eforturi minime.
- Cea mai recentă etapă în evoluția acestui limbaj o constituie limbajul *JAVA* (*1995)* realizat de firma *SUN* (firmă cumpărată în 2010 de către *Oracle).*

În concluzie, sintaxa limbajului C a stat la baza multor limbaje create ulterior şi încă populare azi: C++, Java, JavaScript, C#.

IB.02.2 Caracteristicile limbajului C

Caracteristicile limbajului C, care i-au determinat popularitatea, sunt prezentate pe scurt mai jos şi vor fi analizate pe parcursul cursului:

- **limbaj de nivel înalt, portabil, structurat, flexibil**
- **produce programe eficiente (lungimea codului scăzută, viteză de execuție mare)**
- **set bogat de operatori**
- **multiple facilități de reprezentare şi prelucrare a datelor**
- **utilizare extensivă a apelurilor de funcții şi a pointerilor**
- **verificare mai scăzută a tipurilor - *loose typing* - spre deosebire de PASCAL**
- **permite programarea la nivel scăzut - *low-level* - , apropiat de hardware**

Este utilizat în multe aplicații:

- programe de sistem
- proiectare asistată de calculator
- grafică
- prelucrare de imagini
- aplicații de inteligență artificială.

IB.02.3 Procesul dezvoltării unui program C

Procesul dezvoltării unui program C

Principalele etape în dezvoltarea unui program C sunt:

1. Analiza problemei şi stabilirea cerințelor pe care trebuie să le satisfacă programul (formatul şi suportul datelor de intrare şi al celor de ieşire, cerințe de memorie internă, timp de execuție şi altele).
2. Proiectarea programului, utilizând conceptele programării structurate.
   Rezultatul este o structură ierarhică de unități program care vor fi codificate în limbajul C.
3. Codificarea (implementarea) programului: reprezentarea algoritmilor prin instrucțiuni ale limbajului C.
4. Compilarea programului (eliminarea erorilor de sintaxă).
5. Testarea programului: execuția sa pentru date de intrare semnificative şi compararea rezultatelor cu cele asteptate.

Etapele 3, 4 şi 5 sunt efectuate, de regulă, cu ajutorul unui mediu integrat de dezvoltare (IDE - Interactive Development Environment) precum CodeBlocks, DevCpp, MS Visual Studio, Eclipse sau Netbeans. Un mediu integrat de dezvoltare include un editor de text (program care permite şi uşurează editarea textului sursă al programului), compilatorul, editorul de legături (care asambleaza într-un singur program maşină modulele rezultate din compilarea separată a unităților program), precum şi facilități de depanare a programelor(de exemplu, execuția linie cu linie pentru

descoperirea cauzei unei erori). În absența unui mediu integrat de dezvoltare (ceea ce este mai putin uzual în prezent) putem face aceste operații în mod linie de comandă.

Detaliat, etapele necesare pentru introducerea și execuția unui program C sunt următoarele:
Etapa 1: Editarea programului sursă (codificarea algoritmului într-un limbaj de programare) - Edit Această codificare se va realiza într-un program sursă.
Se salvează fișierul sursă, de exemplu cu numele "Hello.c". Un fișier C++ trebuie salvat cu extensia ".cpp", iar un fișier C cu extensia ".c". Trebuie ales un nume care să reflecte scopul programului.
Această etapă se poate realiza utilizând comenzile respective ale IDE-ului – de exemplu N*ew*, S*ave* - sau un editor de texte – *Notepad*.

Etapa 2: Compilarea și editarea legăturilor
Sunt compilate separat fișierele sursă apoi prin editarea legăturilor se obține programul în limbaj mașină salvat într-un fișier 'executabil'. Exemplu: Hello.exe.
Această etapă se poate realiza utilizând comenzile corespunzătoare ale IDE-ului – de exemplu, *Compile*, *Build* - sau o linie de comandă, dacă se utilizează compilatorul GNU GCC:

În Windows (CMD shell) - generează fișierul executabil *Hello.exe*	**> gcc Hello.c -o Hello.exe**
În Unix or Mac (Bash shell) - generează fișierul executabil *Hello*	**$ gcc Hello.c -o Hello**

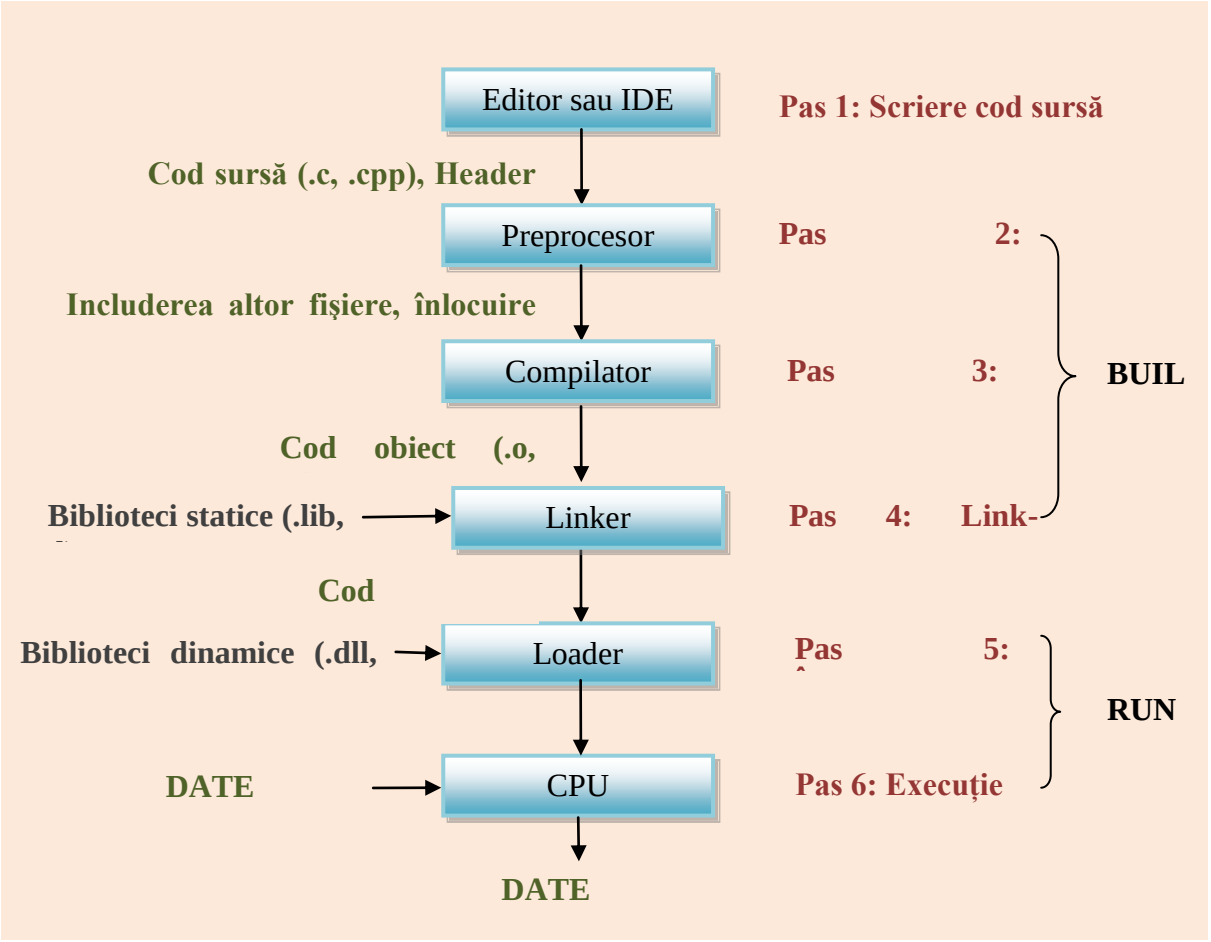
Etapa 3: Execuția programului
Această etapă se poate realiza utilizând comanda respectivă a IDE-ului – *Run* - sau linie de comandă. De exemplu, **rularea** (sau lansarea în execuție) în mod linie de comandă se poate face astfel:

În Windows (CMD shell) - Rulare "Hello.exe" (.exe este opțional)	**> Hello**
În Unix or Mac (Bash shell) - Rulare "Hello"	**$ ./Hello**

Tabelul conține o descriere sintetică a pașilor detaliați:

Pas	Descriere
1	Crearea fișierului sursă (cu extensia *.c* sau *.cpp* în cazul limbajului C/C++) folosind un editor de texte sau un IDE. Rezultă un fișier sursă. Poate fi creat și un fișier antet (*header*), cu extensia *.h*
2	Preprocesarea codului sursă în concordanță cu directivele de preprocesare *(#include, #define* în C). Aceste directive indică operații (includerea unui alt fișier, înlocuire de text etc) care se vor realiza ÎNAINTE de compilare.
3	Compilarea codului sursă preprocesat. Rezultă un fișier obiect (*.obj*, *.o*).
4	Legarea (linked-itarea) codului obiect rezultat cu alte fișiere obiect și biblioteci pentru a furniza fișierul executabil (.exe).
5	Încărcarea codului executabil în memoria calculatorului (RAM).
6	Rularea codului executabil.

Detaliarea acestor etape este reflectată în figura următoare: în stânga avem tipul de fișiere, la mijloc cine realizează acel pas, pentru ca în partea dreaptă a figurii să apară pașii efectivi:



Pentru mai multe detalii vom include aici legături către tutoriale ale unor medii integrate de dezvoltare C:

Tutorial CodeBlocks
Tutorial NetBeans

IB.02.4 Structura unui program C

Un program C este compus dintr-o ierarhie de funcţii, orice program conţinând cel puţin funcţia **main**, prima care se execută la lansarea programului C. Funcţiile pot face parte dintr-un singur fişier sursă sau din mai multe fişiere sursă. Un fişier sursă C este un fişier text care conţine o succesiune de funcţii şi, eventual, declaraţii de variabile.

Structura unui program C este, în principiu, următoarea:

- **directive preprocesor**
- **definiţii de tipuri**
- **prototipuri de funcţii - tipul funcţiei şi tipurile parametrilor funcţiei**
- **definiţii de variabile externe**
- **definiţii de funcţii**

O funcţie C are un antet şi un bloc de instrucţiuni (prin care se execută anumite acţiuni) încadrat de acolade. În interiorul unei funcţii există de obicei declaraţii de variabile precum şi alte blocuri de instrucţiuni.

Primul program C

Urmează un exemplu de program C minimal, ce afişează un text – *Hello World!* – pe ecran. Exemplul conține o funcție *main* cu o singură instrucțiune (apelul funcției *printf*). Sunt prezentate trei variante pentru funcția *main*:

Varianta	Cod

1	`#include<stdio.h>`
	`void main(void)`
	`{`
	`  printf("Hello World!");`
	`}`

2 – fără void în antetul funcției main	`#include<stdio.h>`
	`void main()`
	`{`
	`  printf("Hello World!");`
	`}`

3 - fără warning la compilare (în Code::Blocks, de exemplu)	`#include<stdio.h>`
	`int main()`
	`{`
	`  printf("Hello World!");`
	`  return 0;`
	`}`

Execuția programului începe cu prima linie din *main*. Cuvântul din fața numelui funcției (*main*) reprezintă tipul funcției (*void* arată că această funcție nu transmite nici un rezultat prin numele său, *int* arată că trimite un rezultat de tip întreg, prin instrucțiunea *return*). Parantezele care urmează cuvântului *main* arată că *main* este numele unei funcții (şi nu este numele unei variabile), dar o funcție fără parametri (acel void care poate lipsi – varianta 2).
Acoladele sunt necesare pentru a delimita corpul unei funcții, corp care este un bloc de instrucțiuni şi declarații. Funcția *printf* realizează afişarea pe ecran a textului *Hello World!*.
Directiva *#include* este o directivă de preprocesare, permite includerea unor funcții de bibliotecă. Înm acest caz, indică necesitatea includerii în programul sursă a fişierului *stdio.h*, în care este declarată funcția predefinită (de bibliotecă) *printf*. Fără această includere, compilatorul nu recunoaşte funcția *printf* şi semnalează eroare.

Alte observații care pot fi făcute:

- cuvintele cheie sunt scrise cu litere mici
- instrucțiunile se termină cu ';'
- şirurile de caractere sunt incluse între ghilimele
- limbajul C este **case sensitive** – face diferență între literele mici şi literele mari (*a* este diferit de *A*)
- \n folosit în funcția *printf* poziționează cursorul la începutul liniei următoare

IB.02.5 Elemente de bază ale limbajului C

Elementele de bază ale limbajului C sunt următoarele:

- **Alfabetul şi atomii lexicali**
- **Identificatorii**
- **Cuvintele cheie**
- **Tipurile de date**
- **Constantele şi variabilele**
- **Comentariile**
- **Operatorii şi expresiile**

IB.02.5.1 Alfabetul limbajului

Alfabetul limbajului este compus din toate caracterele care pot fi folosite într-un program C.
Caracterele se codifică conform *codului ASCII* (American Standard Code for Information Interchange), codificarea realizîndu-se pe 8 biți (un octet). Sunt 256 (codificate de la 0 la 255) de caractere în codul ASCII, alfabetul cuprinzând simboluri grafice şi simboluri fără corespondent grafic.
De exemplu, caracterul *A* cu codul ASCII 65 este codificat pe 8 biți – 1 octet astfel:

Puterea lui 2	7 6 5 4 3 2 1 0
Valoare	0 1 0 0 0 0 0 1

Pentru a înțelege mai bine ce înseamnă mod de codificare următoarele noțiuni auxiliare sunt descrise în anexa **Tutorial despre reprezentarea datelor**.

- Octet
- reprezentare în baza 2, 8, 10, 16

Limbajul C este case-sensitive (se face diferență între litere mici şi litere mari).
O altă observație este aceea că spațiul are codul mai mic decat simbolurile grafice (32), iar cifrele (în ordine crescătoare), literele mari şi literele mici (în ordine alfabetică) ocupă câte trei zone compacte - cu intervale între ele.
Pentru mai multe detalii legate de codul ASCII vă rugăm să urmați unul din următoarele link-uri:
Tabel Coduri ASCII
Tabel Coduri ASCII 2

Atomii lexicali pot fi:

- **identificatori**
- **constante (explicite) - numerice, caracter, şir**
- **operatori**
- **semne de punctuație.**

Un atom lexical trebuie scris integral pe o linie şi nu se poate extinde pe mai multe linii. În cadrul unui atom lexical nu se pot folosi spații (excepție fac spațiile dintr-o constantă şir), putem însă folosi caracterul '_'.
Respectarea acestei reguli poate fi mai dificilă în cazul unor şiruri constante lungi, dar există posibilitatea prelungirii unui şir constant de pe o linie pe alta folosind caracterul '\'.

Atomii lexicali sunt separați prin separatori, care pot fi:

- **spațiul**
- **caracterul de tabulare orizontală *\t***
- **terminatorul de linie *\n***
- **comentariul**

Un program este adresat unui calculator pentru a i se cere efectuarea unor operații, dar programul trebuie citit şi înțeles şi de către oameni; de aceea se folosesc **comentarii** care explică de ce se fac anumite acțiuni. Comentariile nu sunt incluse în codul executabil al programului.

Inițial în limbajul C exista un singur tip de comentariu, comentariul multilinie, care începe cu secvența /* şi se termină cu secvența */. Ulterior s-au adoptat şi comentariile din C++, care încep cu secvența // şi se termină la sfârşitul liniei curente.

Identificatorii pot fi :

- **nume utilizator - nume de variabile, constante simbolice, funcții, tipuri, structuri, uniuni - este bine sa fie alese cât mai sugestiv pentru scopul utilizării;**
- **cuvinte cheie ale limbajului C - pot fi folosite doar cu înțelesul cu care au fost definite**
- **cuvinte rezervate - înțelesul poate fi modificat, de evitat acest lucru;**

Limbajul C impune următoarele reguli asupra identificatorilor:

- Un identificator este o secvență de caractere, de o lungime maximă ce depinde de compilator (în general are maxim 255 de caractere). Secvența poate conține litere mici şi mari(a-z, A-Z), cifre (0-9) şi simbolul '_' (underscore, liniuța de subliniere).
- Primul caracter trebuie să fie o literă sau '_'. Pentru că multe nume rezervate de compilator, invizibile programatorului, încep cu '_', este indicat a nu utiliza '_' pentru începutul numelor utilizator.
- Un identificator nu poate fi un cuvânt cheie (int, double, if, else, for).
- Deoarece limbajul C este case-sensitive identificatorii sunt şi ei la rândul lor case-sensitive. Astfel, *suma* este diferit de *Suma* şi de *SUMA*.
- Identificatorii sunt atomi lexicali, deci în cadrul lor nu e permisă utilizarea spațiilor şi a altor caractere speciale (ca +, -, *, /, @, &, virgulă, etc.), putem însă folosi caracterul '_' în locul spațiului şi a caracterelor speciale.

Exemple de identificatori: *suma, _produs, x2, X5a, PI, functia_gauss* etc.

Recomandări:

- Este important să alegem un nume , care reflectă foarte bine semnificația identificatorului respectiv (este *self-descriptive),* de exemplu, vom alege numele nrStudenti sau numarDeStudenti ca identificator pentru o variabilă care memorează numărul de studenți.
- Încercați să nu folosiți identificatori care nu spun nimic (lipsiți de un sens clar): a, b, c, d, i, j, k, i1, j99.
- Evitați numele de un singur caracter care sunt mai uşor de folosit dar de cele mai multe ori lipsite de vreun înțeles. Acest lucru este însă utilizat dacă sunt nume commune cum ar fi x, y, z pentru coordonate sau i, j pentru indici.
- Nu folosiți nume foarte lungi, sunt greu de utilizat, încercați să optimizați lungimea numelui cu înțelesul acestuia.
- Folosiți singularul şi pluralul pentru a diferența. De exemplu, putem folosi numele linie pentru a ne referi la numărul unei singure linii şi numele linii pentru a ne referi la mai multe linii (de exemplu un vector de linii).

Cuvintele cheie se folosesc în declarații şi instrucțiuni şi nu pot fi folosite ca nume de variabile sau de funcții (sunt cuvinte rezervate ale limbajului). Exemple de cuvinte cheie:

int	**extern**	**double**
char	**register**	**float**
unsigned	**typedef**	**static**
do	**else**	**for**

while	**struct**	**goto**
switch	**union**	**return**
case	**sizeof**	**default**
short	**break**	**if**
long	**auto**	**continue**
Standardul ANSI C a mai adăugat:		
signed	**const**	**void**
enum	**volatile**	

Numele de funcții standard (*scanf, printf, sqrt* etc.) nu sunt cuvinte cheie, dar nu se recomandă utilizarea lor în alte scopuri, deoarece aceasta ar produce schimbarea sensului inițial, atribuit în toate versiunile limbajului.

IB.02.5.2 Tipuri de date

În C există tipuri de date fundamentale şi tipuri de date derivate. Sunt prezentate mai jos principale tipuri de date şi numărul de octeți pe care acestea le ocupă pe un sistem Unix de 32 de biți.

Tipurile de date fundamentale sunt:

- caracter (char – 1 octet)
- întreg (int – 4 octeți)
- virgulă mobilă (float – 4 octeți)
- virgulă mobilă dublă precizie (double – 8 octeți)
- nedefinit (void)

Tipurile de date derivate sunt:

- tipuri structurate (tablouri, structuri)
- tipul pointer (4 octeți)

Pentru a utiliza eficient memoria şi a satisface necesitățile unei multitudini de aplicații există în C mai multe tipuri de întregi şi respectiv de reali, ce diferă prin memoria alocată şi deci prin numărul de cifre ce pot fi memorate şi prin domeniul de valori.

Tipurile întregi

Numerele intregi se pot reprezenta în C folosind următoarele tipuri: *char , short , int , long, long long*. Implicit toate numerele întregi sunt numere cu semn (*signed*), dar prin folosirea cuvântului cheie *unsigned* la declararea lor se poate cere interpretarea ca numere fără semn. Utilizarea cuvintelor *long* sau *short* face ca tipul respectiv să aibă un domeniu mai mare, respectiv mai mic de valori.

Tipuri în virgulă mobilă (reale)

Există două tipuri, *float* şi *double*, pentru numere reale reprezentate în virgulă mobilă cu simplă şi respectiv dublă precizie. Unele implementări suportă şi *long double* (numai cu semn). Trebuie semnalat faptul că nu toate numerele reale pot fi reprezentate, întrucât memorarea valorilor reale, fiind realizată pe un anumit număr de biți, nu poate reține decât o parte dintre cifrele semnificative. Deci numai anumite valori reale au reprezentarea exactă în calculator, restul confundându-se cu reprezentarea cea mai apropiată.

Tipul caracter

Caracterele (exemplu 'a', 'Z', '0', '9') sunt codificate ASCII sub formă de valori întregi, şi păstrate în tipul char. De exemplu, caracterul '0' are codul 48 (zecimal, adică în baza 10) sau 30H (hexazecimal – baza 16); caracterul 'A' are codul 65 (zecimal) sau 41H (hexazecimal); caracterul 'a' are codul 97 (zecimal) sau 61H (hexazecimal). Se observă că tipul char poate fi folosit pentru a reprezenta întregi fără semn în domeniul 0 – 255 sau pentru a reprezenta caractere în codul ASCII.

Standardul C din 1999 prevede şi tipul boolean _Bool (sau bool), reprezentat pe un octet.

Reprezentarea internă şi numărul de octeți necesari pentru fiecare tip nu sunt reglementate de standardul limbajului C, dar limitele fiecărui tip pentru o anumită implementare a limbajului pot fi aflate din fişierul "limits.h" (care conține şi nume simbolice pentru aceste limite - INT_MAX şi INT_MIN) sau utilizând operatorul *sizeof*; de exemplu: *sizeof(short)* ne dă numărul de octeți pe care se memorează o valoare de tip short.

De obicei tipul *int* ocupă 4 octeți, iar valoarea maximă este de cca. 10 cifre zecimale pentru tipul *int*. Depăşirile la operații cu întregi de orice lungime nu sunt semnalate deşi rezultatele sunt incorecte în caz de depăşire.
De obicei valorile de tipul int se memoreaza pe 4 octeți, iar valoarea maximă este de circa 10 cifre zecimale. Depăşirile la operații cu întregi de orice lungime nu sunt semnalate deşi rezultatele sunt incorecte în caz de depăşire.
Reprezentarea numerelor reale în diferite versiuni ale limbajului C este mai uniformă deoarece urmează un standard IEEE de reprezentare în virgulă mobilă. Pentru tipul *float* domeniul de valori este între 10E-38 şi 10E+38 iar precizia este de 6 cifre zecimale exacte. Pentru tipul *double* domeniul de valori este între 10E-308 şi 10E+308 iar precizia este de 15 cifre zecimale.

Tabelul următor prezintă dimensiunea tipică, valorile minimă şi maximă pentru tipurile primitive (în cazul general). Încă o dată, dimensiunea este dependentă de implementarea C folosită.

Categorie	Tip	Descriere	Octeți	Valoare minimă	Valoare Maximă
Numere Întregi	**int** **signed int**	Întreg cu semn (cel puțin 16 biți)	4 (2)	-2147483648	2147483647 (=2^{31}-1)
	unsigned int	Întreg fără semn (cel puțin 16 biți)	4 (2)	0	4294967295 (=2^{32}-1)
	char	Caracter (poate fi cu semn sau fără semn, depinde de implementare)	1		
	signed char	Caracter sau întreg mic cu semn (garantează că e cu semn)	1	-128	127 (=2^{7}-1)
	unsigned char	Caracter or sau întreg mic fără semn (garantează că e fără semn)	1	0	255 (=2^{8}-1)
	short **short int** **signed short** **signed short int**	Întreg scurt cu semn (cel puțin 16 biți)	2	-32768	32767 (=2^{15}-1)

	unsigned short **unsigned short int**	Întreg scurt fără semn (cel puțin 16 biți)	2	0	65535 ($=2^{16}$-1)
	long **long int** **signed long** **signed long int**	Întreg lung cu semn (cel puțin 32 biți)	4 (8)	-2147483648	2147483647 ($=2^{31}$-1)
	unsigned long **unsigned long int**	Întreg lung fără semn (cel puțin 32 biți)	4 (8)	0	4294967295 ($=2^{32}$-1)
	long long **long long int** **signed long long** **signed long long int**	Întreg foarte lung cu semn (cel puțin 64 biți) (de la standardul C99)	8	-2^{63}	($=2^{63}$-1)
	unsigned long long **unsigned long long int**	Întreg foarte lung fără semn (cel puțin 64 biți) (de la standardul C99)	8	0	2^{64}-1
Numere Reale	**float**	Număr în virgulă mobilă, ≈7 cifre(IEEE 754 format virgulă mobilă simplă precizie)	4	3.4e-38	3.4e38
	double	Număr în virgulă mobilă dublă precizie, ≈15 cifre(IEEE 754 format virgulă mobilă dublă precizie)	8	1.7e-308	1.7e308
	long double	Număr lung în virgulă mobilă dublă precizie, ≈19 cifre(IEEE 754 format virgulă mobilă cvadruplă precizie)	12 (8)	3.4E-4932	1.1E4932
Numere booleene	**bool**	Valoare booleană care poate fi fie `true` fie `false` (de la standardul C99)	1	false (0)	true (1 sau diferit de zero)

IB.02.5.3 Valori corespunzătoare tipurilor fundamentale

Aceste valori constante, ca `123`, `-456`, `3.14`, `'a'`, `"Hello"`, pot fi atribuite direct unei variabile sau pot fi folosite ca parte componentă a unei expresii.

Valorile întregi se reprezintă implicit în baza 10 şi sunt de tipul `signed care este acoperitor` pentru reprezentarea lor.

- Pentru reprezentarea constantelor fără semn se foloseşte sufixul u sau U
- O constantă întreagă poate fi precedată de semnul plus (+) sau minus.
- Se poate folosi prefixul '0' (zero) pentru a arăta că acea valoare este reprezentată în octal, prefixul '0x' pentru o valoare în hexadecimal şi prefixul '0b' pentru o valoare binară (acceptată de unele compilatoare).
- O constanta de tip întreg long este identificată prin folosirea sufixului 'L' sau 'l'. Un long long int este identificat prin folosirea sufixului 'LL'. Putem folosi sufixul 'U' sau 'u' pentru unsigned int, 'UL' pentru unsigned long, şi 'ULL' pentru unsigned long long int.
- Pentru valori constante de tip short nu e nevoie de sufix.

Exemple:

```
-10000          // int
65000           // long
32780           // long
32780u          // unsigned int
1234;           // Decimal
01234;          // Octal 1234, Decimal 2322
0x1abc;         // hexadecimal 1ABC, decimal 15274
0b10001001;     // binar (doar în unele compilatoare)
12345678L;      // Sufix 'L' pentru long
123UL;          // int 123 auto-cast la long 123L
987654321LL;    // sufix 'LL' pentru long long int
```

Valorile reale

Sunt implicit de tipul double; sufixul f sau F aplicat unei constante, o face de tipul float, iar l sau L de tipul long double.
Un număr cu parte fracționară, ca `55.66` sau `-33.442`, este tratat implicit ca do`uble`.
O constanta reală se poate reprezenta şi sub forma ştiințifică.

Exemplu:

```
1.2e3       // 1.2*10^3
-5.5E-6     // -5.5*10^-6
```

unde E denotă exponentul puterii lui 10. Mantisa, partea de dinaintea lui E poate fi precedată de semnul plus (+) sau minus (-).

- mantisa are forma: *parte_intreagă.parte_zecimala* (oricare din cele două părți poate lipsi, dar nu ambele)
- exponentul are forma: e*val_exponent* sau E*val_exponent*, unde *val_exponent* este un număr întreg.

Valoarea constantei este produsul dintre mantisa si 10 la puterea daăa de val_exponent.

In tabelul de mai jos apar cateva exemple de constante reale:

Constante de tip float	Constante de tip double	Constante de tip long double
1.f	1.	1.L
.241f	.241	.241l

-12.5e5f	-12.5e5	-12.5e5l
98.E-12f	98.E-12	98.E-12L

Valori caracter

Valorile de tip caracter se reprezintă pe un octet şi au ca valoare codul ASCII al caracterului respectiv.
În reprezentarea lor se foloseşte caracterul apostrof : ‘A’ (cod ASCII 65), ‘b’, ‘+’.
Pentru caractere speciale se foloseşte caracterul \.
Exemple:

```
'\''  - pentru apostrof
'\\'  - pentru backslash
Este greşit:      ''   sau  '\'
```

Constantele caracter pot fi tratate în operațiile aritmetice ca întregi cu semn reprezentați pe 8 biți. Cu alte cuvinte, *char* şi *signed int* pe 8 biți sunt interschimbabile. De asemenea, putem atribui un întreg în domeniul [-128, 127] unei variabile de tip char şi [0, 255] unui unsigned char.
Caracterele non-tipăribile şi caracterele de control pot fi reprezentate fie prin *secvențe escape*, care încep cu un back-slash (\) urmat de o literă (‘\n’ = new line , ‘\t’ =tab , ‘\b’ = backspace etc), fie prin codul numeric al caracterului în octal sau în hexazecimal (\012 = \0x0a = 10 este codul pentru caracterul de trecere la linie nouă ‘\n’).
Cele mai utilizate secvențe escape sunt:

Secvența escape	Descriere	Hexa (Decimal)
\n	Linie nouă (Line feed)	0AH (10D)
\r	Carriage-return	0DH (13D)
\t	Tab	09H (9D)
\"	Ghilimele	22H (34D)
\'	Apostrof	27H (39D)
\\	Back-slash	5CH (92D)

Valori şir de caractere

Valorile şir de caractere sunt compuse din zero sau mai multe caractere precizate între ghilimele.

Exemple:

```
"Hello, world!"
"The sum is "
"" //reprezintă şirul vid
```

Fiecare caracter din şir poate fi un *simbol grafic*, o *secvență escape* sau un *cod ASCII* (în octal sau hexazecimal). Spațiul ocupat este un număr de octeți cu unu mai mare decât numărul caracterelor din şir, ultimul octet fiind rezervat pentru *terminatorul de şir*- caracterul cu codul ASCII 0, adica '\0'. Dacă se doreşte ca şi caracterul ghilimele să faca parte din şir, el trebuie precedat de \.

Exemple:

```
"CURS"   -   "\x43URS"
// scrieri echivalente ale unui şir ce ocupă 5 octeți

"1a24\t"   -  "\x31\x61\x32\x34\11"
//scrieri echivalente ale unui sir ce ocupă 6 octeți

 "'\""
//şir ce conține caracterele '" şi terminatorul - ocupă 3 octeți
```

Tipul void nu are constante (valori) şi este utilizat atunci când funcțiile nu întorc valori sau când funcțiile nu au parametri:

```
// o funcție nu întoarce o valoare:

void f ( int a)
{
      if (a) a = a / 2;
}

// sau când funcțiile nu au parametri:

void f (void);
// echivalent cu void f();

int f (void);
//echivalent cu int f();
```

IB.02.5.4 Constante simbolice

Acestea sunt constante definite cu ajutorul unor identificatori. Pot fi: predefinite sau definite de programator.
Literele mari se folosesc în numele unor constante simbolice predefinite: EOF, M_PI, INT_MAX, INT_MIN. Aceste constante simbolice sunt definite în fişiere header (de tip "h") : EOF în "stdio.h", M_PI în "math.h".

Definirea unei constante simbolice se face utilizând directiva *#define* astfel:

#define identificator [text]

Exemple:

```
#define begin {   // unde apare begin acesta va fi înlocuit cu {
#define end }     // unde apare end acesta va fi înlocuit cu }
#define N 100     // unde apare N acesta va fi înlocuit cu 100
```

Tipul constantelor C rezultă din forma lor de scriere, în concordanță cu tipurile de date fundamentale.
În C++ se preferă altă definire pentru constante simbolice, utilizând cuvântul cheie *const*. Pentru a face diferențierea dintre variabile şi constante este de preferat ca definirea constantelor să se facă prin scrierea acestora cu majuscule.
Exemple:

```
const double PI = 3.1415;
const int LIM = 10000;
```

IB.02.5.5 Variabile

Variabila este o entitate folosită pentru memorarea unei valori de un anumit tip, tip asociat variabilei. O variabilă se caracterizează prin nume, tip, valoare şi adresă:

- Orice variabilă are un **nume** (*identificator*), exemplu: *raza*, *area*, *varsta*, *nr*. Numele identifică în mod unic fiecare variabilă permiţând utilizarea acesteia, cu alte cuvinte, numele unei variabile este unic (nu pot exista mai multe variabile cu acelaşi nume în acelaşi domeniu de definiţie)
- Orice variabilă are asociat un ***tip*** *de date*. Tipul poate *fi orice tip fundamental* sau derivat, precum şi un tip definite de programator.
- O variabilă poate stoca o **valoare** de un anumit tip. De menţionat că în majoritatea limbajelor de programare, o variabilă asociată cu un anumit tip poate stoca doar valori aparţinând acelui tip. De exemplu, o variabilă de tipul int poate stoca valoarea 123, dar nu şirul "Hello".
- Oricărei variabile i se alocă (rezervă) un spaţiu de memorie corespunzător tipului variabilei. Acest spaţiu este identificat printr-o **adresă de memorie.**

Pentru a folosi o variabilă trebuie ca mai întâi să îi declarăm numele şi tipul, folosind una din următoarele forme sintactice:

tip nume_variabila;
// Declară o variabilă de un anumit tip

tip nume_variabila1, nume_variabila2,...;
// Declaraţie multiplă pentru mai multe variabile de acelaşi tip

tip var-name = valoare_iniţiala;
// Declară o variabilă de un anumit tip şi îi atribuie o valoare iniţială

tip nume_variabila1 = valoare_iniţiala1, nume_variabila2 = valoare_iniţiala2, ... ;
// Declară mai multe variabile unele putând fi iniţializate, nu e obligatoriu să fie toate!

Sintetizând, definirea variabilelor se face astfel:

Tip lista_declaratori;

- ***lista_declaratori*** cuprinde unul sau mai multi declaratori, despărţiţi prin virgulă
- ***declarator*** poate fi**:**
  - ***nume_variabila*** sau
  - ***nume_variabila = expresie_de_initializare***

În *expresie_de_initializare* pot apare doar constante sau variabile iniţializate!

Exemple:

```
int sum;                // Declară a variabilă sum de tipul int
int nr1, nr2;           // Declară două variabile nr1 şi nr2 de tip int
double media;           // Declară a variabilă media de tipul double
int height = 20;        /* Declară a variabilă de tipul int şi îi atribuie o
valoare iniţială */
char c1;                // Declară o variabilă c1 de tipul char
char car1 = 'a', car2 = car1 + 1;     // car2 se initializeaza cu 'b'
float real = 1.74, coef;  /*Declară două variabile de tip float prima din ele
este şi iniţializată*/
```

În definirea ***tip lista_declaratori;*** tip poate fi precedat sau urmat de cuvântul cheie ***const***, caz în care variabilele astfel definite sunt de fapt **constante** ce trebuie să fie inițializate şi care nu-şi mai pot modifica apoi valoarea:

```
const int coef1 = -2, coef2 = 14;
coef1 = 5; /* modificarea valorii variabilei declarate const e gresita,
apare eroare la compilare!! */
```

Odată ce o variabilă a fost definită îi putem atribui sau reatribui o valoare folosind *operatorul de atribuire* "=".

Exemple:

```
int number;
nr = 99;        //atribuie valoarea întreagă 99 variabilei nr
nr = 88;        //îi reatribuie valoarea 88
nr = nr + 1;    //evaluează nr+1 şi atribuie rezultatul lui nr
int sum = 0;
int nr;         //ERROR: O variabilă cu numele nr e deja definită
/* WARNING: Variabila sum este de tip int (va primi valoarea 55)*/
sum = "Hello";   /* ERROR: Variabila sum este de tip int. Nu i se poate
atribui o valoare şir */
```

Observații:

- O variabilă poate fi declarată o singură dată.
- În fişiere cu extensia *cpp* putem declara o variabilă oriunde în program, atâta timp cât este declarată înainte de utilizare.
- Tipul unei variabile nu poate fi schimbat pe parcursul programului.
- Numele unei variabile este un substantiv, sau o combinație de mai multe cuvinte. Prin convenție, primul cuvânt este scris cu literă mică, în timp ce celelalte cuvinte pot fi scrise cu prima literă mare, fără spații între cuvinte. Exemple: *dimFont*, *nrCamera*, *xMax*, *yMin*, *xStangaSus* sau *acestaEsteUnNumeFoarteLungDeVariabila*.

IB.02.5.6 Comentarii

Comentariile sunt utilizate pentru documentarea şi explicarea logicii şi codului programului. Comentariile nu sunt instrucțiuni de programare şi sunt ignorate de compilator, dar sunt foarte importante pentru furnizarea documentației si explicațiilor necesare pentru înțelegerea programului nostru de către alte persoane sau chiar şi de noi înşine peste o săptămână.
Există două tipuri de comentarii:

Comentarii Multi-linie:
încep cu /* şi se termină cu */, şi se pot întinde pe mai multe linii

***Comentarii în linie*:**
încep cu // şi țin până la sfârşitul liniei curente.

Exemple:

```
/* Acesta este
            un comentariu în C */
```

```
// acesta este un alt comentariu C (C++)
```

În timpul dezvoltării unui program, în loc să ştergem o parte din instrucţiuni pentru totdeauna, putem să le transformăm in comentarii, astfel încât să le putem recupera mai târziu dacă vom avea nevoie.

IB.02.5.7 Operatori, operanzi, expresii

Limbajul C se caracterizează printr-un set foarte larg de operatori.
Operatorii sunt simboluri utilizate pentru precizarea operațiilor care trebuie executate asupra operanzilor.
Operanzii pot fi*:* constante, variabile, nume de funcţii, expresii.
Expresiile sunt entități construite cu ajutorul operanzilor şi operatorilor, respectând *sintaxa (regulile de scriere)* şi *semantica (sensul, înţelesul)* limbajului. Cea mai simplă expresie este cea formată dintr-un singur operand.

Cu alte cuvinte, putem spune că o expresie este o combinație de *operatori* ('+', '-','*', '/', etc.) şi *operanzi* (variabile sau valori constante), care poate fi evaluată ca având o valoare fixă, de un anumit tip.
Expresiile sunt de mai multe tipuri, ca şi variabilele: 3*x+7, x<y etc.
La evaluarea expresiilor se ține cont de precedență şi asociativitatea operatorilor!

Exemple

```
1 + 2 * 3           // rezultă int 7

int sum, number;
sum + number;        // evaluată la o valoare de tip int

double princ, rata;
princ * (1 + rata);  // evaluată la o valoare de tip double

sum  +  princ                  //  evaluata  la  o  valoare  de  tip  double
```

Operatorii

Clasificarea operatorilor se poate face:

- **după numărul operanzilor prelucrați:**
  - **unari**
  - **binari**
  - **ternari - cel condițional;**

- **după ordinea de succedare a operatorilor şi operanzilor:**
  - **prefixati**
  - **infixati**
  - **postfixati**

- **după tipul operanzilor şi al prelucrării:**
  - **aritmetici**
  - **relaționali**
  - **logici**
  - **la nivel de bit.**

Operatorii se împart în *clase de precedență*, fiecare clasă având o *regulă de asociativitate*, care indică ordinea aplicării operatorilor consecutivi de aceeaşi precedență (prioritate). Regula de

asociativitate de la stânga la dreapta înseamnă că, de exemplu, expresia 1+2+3-4 este evaluată astfel: ((1+2)+3)-4.

Tabelul operatorilor C indică atât regula de asociativitate, implicit de la stânga la dreapta (s-a figurat doar unde este dreapta-stânga), cât şi clasele de precedență, de la cea mai mare la cea mai mică precedență:

Operator	Semnificație	Utilizare	Asociativitate
( ) **[ ]** **.** ->** **--** **++**	paranteze indexare selecție selecție indirectă postdecrementare postincrementare	(e) t[i] s.c p->c a-- a++	
- **+** **--** **++** **!** **~** * **&** **sizeof** **( )**	schimbare semn plus unar (fără efect) predecrementare preincrementare negație logică complementare (negare bit cu bit) adresare indirectă preluare adresă determinare dimensiune (în octeți) conversie de tip (cast)	-v +v --a ++a !i ~i *p &a sizeof(x) (d) e	dreapta - stânga ←
* / **%**	înmulțire împărțire rest împărțire (modulo)	v1 * v2 v1 / v2 v1 % v2	
+ -	adunare scădere	v1 + v2 v1 - v2	
<< >>	deplasare stânga deplasare dreapta	i1 << i2 i1 >> i2	
< <= > >=	mai mic mai mic sau egal mai mare mai mare sau egal	v1 < v2 v1 <= v2 v1 > v2 v1 >= v2	
== **!=**	egal diferit	v1 == v2 v1 != v2	
&	şi pe biți	i1 & i2	
^	sau exclusiv pe biți	i1 ^ i2	
|	sau pe biți	i1 | i2	
&&	şi logic (conjuncție)	i1 && i2	
||	sau logic (disjuncție)	i1 || i2	
? :	operator condițional (ternar)	expr ? v1 : v2	dreapta - stânga ←
=	atribuire	a = v	

***= /= %= += -= &= ^= |= <<= >>=**	variante ale operatorului de atribuire	a *= v	dreapta - stânga ←
,	secvențiere	e1, e2	

Legendă:

a - variabila întreagă sau reală
c - câmp
d - nume tip
e - expresie
f - nume de funcție
i - întreg
v - valoare întreagă sau reală
p - pointer
s - structură sau uniune
t - tablou
x - nume tip sau expresie

Operatorii aritmetici

Limbajul C pune la dispoziție următorii operatori aritmetici pentru numere de tip întreg: `short`, `int`, `long`, `long long`, `char` (tratat ca 8-bit signed integer), `unsigned short`, `unsigned int`, `unsigned long`, `unsigned long long`, `unsigned char`, `float`, `double` şi `long double`.
expr1 şi *expr2* sunt două expresii de tipurile enumerate mai sus.

Operator	Semnificație	Utilizare	Exemple
-	schimbare semn	*-expr1*	-6 -3.5
+	plus unar (fără efect)	*+expr1*	+2 +2.5
*	înmulțire	*expr1 * expr2*	2 * 3 → 6; 3.3 * 1.0 → 3.3
/	împărțire	*expr1 / expr2*	1 / 2 → 0; 1.0 / 2.0 → 0.5
%	rest împărțire (modulo)	*expr1 % expr2*	5 % 2 → 1; -5 % 2 → -1
+	adunare	*expr1 + expr2*	1 + 2 → 3; 1.1 + 2.2 → 3.3
-	scădere	*expr1 - expr2*	1 - 2 → -1; 1.1 - 2.2 → -1.1

Este important de reținut că `int / int` produce un `int`, cu rezultat trunchiat, şi anume partea întreagă a împărțirii: `1/2 → 0` (în loc de `0.5`), iar operatorul modulo (%) este aplicabil doar pentru numere întregi.

În programare, următoarea expresie aritmetică:

$$\frac{1+2a}{3}+\frac{4(b+c)(5-d-e)}{f}-6\left(\frac{7}{g}+h\right)$$

Trebuie scrisă astfel: (1+2*a)/3 + (4*(b+c)*(5-d-e))/f - 6*(7/g+h). Simbolul pentru înmulțire '*' nu se poate omite (cum este în matematică).

Ca şi în matematică, înmulţirea şi împărţirea au precedenţă (prioritate) mai mare decât adunarea şi scăderea. Aceasta înseamnă că la evaluarea unei expresii în care apar operatori de adunare, scădere, înmulţire şi împărţire, se vor efectua mai întâi operaţiile de înmulţire şi împărţire în ordinea lor de la stânga la dreapta, apoi operaţiile de adunare şi scădere, tot de la stânga la dreapta. Parantezele sunt însă cele care au cea mai mare precedenţă: într-o expresie cu paranteze se evaluează mai întâi conţinutul fiecărei paranteze (ţinând cont de precedenţa operatorilor) apoi se evaluează expresia fără paranteze (în care fiecare paranteză a fost înlocuită cu rezultatul evaluării sale).
Depăşirile la calculele cu valori reale sunt semnalate, dar nu şi cele de la calculele cu valori întregi (valoarea rezultată este trunchiată). Se semnalează, de asemenea, eroarea la împărţirea cu 0.

Operatori relaţionali şi logici
Deseori, e nevoie să comparăm două valori înainte să decidem ce acţiune să realizăm. De exemplu, dacă nota este mai mare decât 50 afişează "Admis". Orice operaţie de comparaţie implică doi operanzi (x <= 100).

În C există şase **operatori de comparaţie** (mai sunt numiţi şi **operatori relaţionali**):

Operator	Semnificaţie	Utilizare	Exemple (x=5, y=8)
==	egal cu	*expr1 == expr2*	(x == y) → false
!=	diferit	*expr1 != expr2*	(x != y) → true
>	mai mare	*expr1 > expr2*	(x > y) → false
>=	mai mare egal	*expr1 >= expr2*	(x >= 5) → true
<	mai mic	*expr1 < expr2*	(y < 8) → false
<=	mai mic egal	*expr1 >= expr2*	(y <= 8) → true

Aceste operaţii de comparaţie returnează valoarea 0 pentru fals şi o valoare diferită de zero pentru adevărat.

Convenţie C:

Valoarea 0 este interpretată ca *fals* şi orice valoare diferită de zero ca *adevărat*.

În C există patru **operatori logici**:

Operator	Semnificaţie	Utilizare	Exemple (expr1=0, expr2=1)
&&	şi logic	*expr1 && expr2*	**0**
||	sau logic	*expr1 || expr2*	Diferit de 0
!	negaţie logică	*!expr1*	Diferit de 0
^	sau exclusiv logic	*expr1 ^ expr2*	1

Tabelele de adevăr sunt următoarele:

AND (&&)	true	false
true	true	false
false	false	false

OR (||)	true	false
true	true	true
false	true	false

NOT (!)	true	false
	false	true

XOR (^)	true	false
true	false	true
false	true	false

Este incorect să scriem 1 < x < 100; operația trebuie separată în două operații de comparație, x>1, x < 100, pe care le unim cu un operator logic ŞI: (x > 1) && (x < 100).

Exemple:

```
// Returnează true dacă x este între 0 şi 100 (inclusiv)
(x >= 0) && (x <= 100)          // greşit 0 <= x <= 100

// Returnează true dacă x nu este între 0 şi 100 (inclusiv)
(x < 0) || (x > 100)    //sau
 x < 0 || x > 100      /* operatorii relationali au prioritate mai mare
                          decat cei logici */
!((x >= 0) && (x <= 100))

/* Returnează true dacă year este bisect. Un an este bisect dacă este
divizibil cu 4 dar nu cu 100, sau este divisibil cu 400.*/
((year % 4 == 0) && (year % 100 != 0)) || (year % 400 == 0)
//sau:
year % 4 == 0 && year % 100 != 0 || year % 400 == 0
/* operatorii aritmetici au prioritate mai mare decat cei relationali */
```

Operatorii de atribuire

Pe lângă operatorul de atribuire '=', utilizat în exemplele anterioare, limbajul C mai pune la dispoziție așa numiții operatori de atribuire compuşi:

Operator	Semnificație	Utilizare	Exemple
=	atribuire	var = expr	x=5
+=	*var = var + expr*	var + = expr	x+=5 (echivalent cu x=x+5)
-=	*var = var – expr*	var - = expr	x-=5 (echivalent cu x=x-5)
*=	*var = var * expr*	var * = expr	x*=5 (echivalent cu x=x*5)
/=	*var = var / expr*	var / = expr	x/=5 (echivalent cu x=x/5)
%=	*var = var % expr*	var %= expr	x%=5 (echivalent cu x=x%5)

Limbajul C mai introduce şi cei doi operatori aritmetici pentru incrementare şi decrementare cu 1:

Operator	Semnificație	Exemple	Rezultat

--var	predecrementare	y = --x;	echivalent cu
			x=x-1;
			y=x;

var--	post decrementare	y=x--;	echivalent cu
			y=x;
			x=x-1;

++var	preincrementare	y=++x;	echivalent cu x=x+1; y=x;

var++	post incrementare	y=x++;	echivalent cu
			y=x;
			x=x+1;

Exemple:

```
int i, j, k;
//   variabilele   sunt   initializate   cu   aceeasi   valoare,   0
i = j = k = 0;

float lungime, latime, inaltime, baza, volum;
// calculeaza baza si volumul unui paralelipiped
volum = inaltime * ( baza = lungime * latime );
```

Operatorii pe biți

Se aplică fiecărui bit din reprezentarea *operanzilor întregi* spre deosebire de restul operatorilor care se aplică valorilor operanzilor.

Din această categorie fac parte operatorii următori, care apar in ordinea descrescătoare a priorității:

Operatori pe biți	
~	**complementare**
>>	**deplasare la dreapta**
<<	**deplasare la stanga**
&	**si**
^	**sau exclusiv**
|	**sau**

Operatorii logici pe biți sunt cuprinşi în următorul tabel:

Operator	Descriere	Utilizare
&	ŞI pe biți	*expr1* & *expr2*
|	SAU pe biți	*expr1* | *expr2*
^	XOR pe biți	*expr1* ^ *expr2*

De asemenea, sunt disponibili şi operatorii compuşi de atribuire: &=, |= şi ^=.

Operatorii &, ^, | realizează operațiile *şi, sau exclusiv,* respectiv *sau,* între toate perechile de biți corespunzători ai operanzilor. Daca b1 si b2 reprezintă o astfel de pereche, tabelul următor prezintă valorile obținute prin aplicarea operatorilor &, ^, |.

b1	b2	b1&b2	b1^b2	b1|b2
0	0	0	0	0
0	1	0	1	1

1	0	0	1	1
1	1	1	0	1

Din tabela de mai sus se observă că aplicând unui bit:

- operatorul & cu 0, bitul este şters
- operatorul & cu 1, bitul este neschimbat
- operatorul | cu 1, bitul este setat (are valoarea 1)
- operatorul ^ cu 1, bitul este complementat.

Operatorul ~ transformă fiecare bit din reprezentarea operandului în complementarul său (biții 1 în 0 şi cei 0 în 1)
Exemplu:
char a,b;

a	b	~a	!a	~b	!b	a&b	a^b	a|b
00000000	00000001	11111111	1	11111110	0	00000000	00000001	00000001
11111111	10101010	00000000	0	01010101	0	10101010	01010101	11111111

Operatorii de deplasare sunt descrişi în tabelul următor:

Operator	Utilizare	Descriere
<<	operand << number	Deplasare la stânga
>>	operand >> number	Deplasare la dreapta

În cazul operatorilor de deplasare, care sunt binari, primul operand este cel al carui biți sunt deplasați, iar al doilea indică numărul de biți cu care se face deplasarea -- deci numai primul operand este prelucrat la nivel de bit: a<<n, a>>n.
La *deplasarea la stânga* cu o poziție, bitul cel mai semnificativ se pierde, iar în dreapta se completează cu bitul 0.
La *deplasarea la dreapta* cu o poziție, bitul cel mai puțin semnificativ se pierde, iar în stânga se completează cu un bit identic cu cel de semn.

În tabelul următor apar valorile obținute (în binar şi zecimal) prin aplicarea operatorilor de deplasare:
char a;

a	a<<1	a<<2	a>>1	a<<2
00000001 **1**	00000010 2	00000100 4	00000000 0	00000000 0
00001110 **14**	00011100 28	00111000 56	00000111 7	00000011 3
11111111 **-1**	11111110 -2	11111100 -4	11111111 -1	11111111 -1
11011101 **-35**	10111010 -70	01110100 116(depasire)	11101110 -18	11110111 -9

Cu excepția cazurilor când se produce depăşire, deplasarea la stânga cu n biți echivalează cu înmulțirea cu 2 la puterea n. Deplasarea la dreapta cu n biți echivalează cu împărțirea cu 2 la puterea n.
Este indicat să se realizeze înmulțirile şi împărțirile cu puteri ale lui 2 prin deplasări (necesită un timp mult mai scurt):

Exemple:
Cele doua secvențe din tabelul următor conduc la aceleaşi rezultate, unde i este de tipul int (int i;)

operație	echivalent cu
i*=8;	i<<=3;
i/=4;	i>>=2;
i*=10;	i=i<<3+i<<1;

Se consideră n, p întregi. Să se seteze pe 1 bitul p din reprezentarea lui n, (ceilalți biți rămân nemodificați).

```
unsigned int n=5, p=1;
n = n | (1<<p);
```

Se consideră n şi p întregi. Să se seteze pe 0 bitul p din reprezentarea lui n.

```
unsigned int n=7,p=1;
n&= ~(1<<p);
```

Operatorul sizeof

Rezultatul aplicării acestui operator unar este un întreg reprezentând *numărul de octeți* necesari pentru stocarea unei valori de tipul operandului sau pentru stocarea rezultatului expresiei dacă operandul este o expresie. Operatorul are efect la compilare, pentru că atunci se stabileşte tipul operanzilor.

Sintaxa:

sizeof (tip)
sizeof (expresie)

Exemple:

```
sizeof('a')       // =1
sizeof(int)       // =2
sizeof(2.5 + 3)   // =4
```

Operatorul condițional

Operatorul condițional ? : este singurul *operator ternar*. Are prioritatea mai ridicată doar decât a operatorilor de atribuire şi a celui secvențial (virgulă), iar asociativitatea este de la dreapta spre stânga.

El se foloseşte în situațiile în care există două variante de obținere a unui rezultat, dintre care se alege una singură, funcție de îndeplinirea sau neîndeplinirea unei condiții. Cei trei operanzi sunt expresii, prima reprezentand condiția testată.

expr0 ? expr1 : expr2

Dacă valoarea expr0 este adevarată (!=0), se evaluează expr1, altfel expr2, rezultatul expresiei evaluate fiind rezultatul final al expresiei condiționale.

Exemple:

```
/* Expresia de mai jos determină valoarea maximă dintre a si b, pe care o memorează în max:*/
max = a > b ? a : b;
/* În funcție de ora memorată în variabila hour, funcția puts va tipări mesajul corespunzător.
Evaluare dreapta -> stânga a expresiilor condiționale multiple */
```

```
puts( hour < 0 || hour > 24 ? "Ora invalida" : hour < 12 ? "Buna dimineata!" : hour < 18 ? "Buna ziua!" : hour < 22 ? "Buna seara" : "Noapte buna!");
```

Operatorul secvențial

Operatorul secvențial ‘ **,**’ (virgulă) este cel cu prioritatea cea mai scăzută. Se foloseşte atunci când sintaxa limbajului impune prezența unei singure expresii, iar prelucrarea presupune evaluarea a două sau mai multe expresii; acestea se evaluează de la stânga la dreapta, iar rezultatul întregii expresii este cel al ultimei expresii (exprn):

expr1, expr2, ..., exprn

Exemplu:

```
/* Expresia de mai jos memorează în max valoarea maximă dintre a si b,
realizând şi ordonarea descrescătoare a acestora (le interschimbă dacă
a<b). A se observa că interschimbarea presupune utilizarea unei variabile
auxiliare. Operatorul secvențial e necesar pentru a avea o singură expresie
după : */

            int          a,          b,          aux,          max;
    max = a >= b ? a : (aux = b, b = a, a = aux);
```

IB.02.6 Conversii de tip. Operatorul de conversie explicita (cast)

Conversia de tip se efectueaz[asupra unui operand de un anumit tip şi are ca rezultat o valoare echivalentă de un alt tip.

Există două tipuri de conversii de tip:

1. **Implicită, realizată automat de compilator**
2. **Explicită, utilizând operatorul unar de conversie de tip în forma:**
   (tip_nou) ***operand***

Conversii de tip implicite

În limbajul C, dacă atribuim o valoare de tip double unei variabile întregi, compilatorul realizează o conversie de tip implicit**ă**, returnând o valoare întreagă. Partea fracționară se va pierde. Unele compilatoare generează o avertizare (warning) sau o eroare "possible loss in precision"; altele nu.

La evaluarea expresiilor pot apare *conversii implicite*:

- dacă o expresie are doar operanzi întregi, ei se convertesc la int;
- dacă o expresie are doar operanzi reali sau intregi, ei se convertesc la double;
- dacă o expresie are operanzi de tipuri diferite, compilatorul converteste valoarea tipului mai *mic* la tipul mai *mare*. Operația se realizează apoi în domeniul tipului mai mare. De exemplu, int / double → double / double → double. Deci, 1/2 → 0, 1.0/2.0 → 0.5, 1.0/2 → 0.5, 1/2.0 → 0.5.

- în expresia variabila=expresie se evaluează prima dată expresia, fără a ține cont de tipul variabilei; dacă tipul rezultatului obținut este diferit de cel al variabilei, se realizează conversia implicită la tipul variabilei astfel:

Exemple:

Tip	Exemplu	Operație
int	2 + 3	int 2 + int 3 → int 5
double	2.2 + 3.3	double 2.2 + double 3.3 → double 5.5
mixt	2 + 3.3	int 2 + double 3.3 → double 2.0 + double 3.3 → double 5.3
int	1 / 2	int 1 / int 2 → int 0
double	1.0 / 2.0	double 1.0 / double 2.0 → double 0.5
mixt	1 / 2.0	int 1 / double 2.0 → double 1.0 + double 2.0 → double 0.5

Exemple conversii implicite:

```
int     i;     char     c     =     'c';     long     l;     float     f;
i = 2.9;          //  2.9  e  convertit  implicit  la  int,  deci  i  va  fi  2
f = 'A';          //    f    va    fi    65.0    (    codul    ASCII    )
i = 30000 + c;    // expresia se calculeaza in domeniul int, i va fi 30099
i = 30000 + 10000;// calcul in dom. int, depasire nesemnalata, i = 25536
l = 30000 + 10000;      //    -25536    va    fi    convertit    la    long
l=30000u+10000;         //                 rezultat                  corect
l=30000l+10000;         // rezultat corect
```

Conversii de tip explicite

Operatorul de conversie explicită (denumit cast) se utilizează atunci când se doreşte ca valoarea unui operand (expresie) să fie de un alt tip decât cel implicit. Operatorul este unar, are prioritate ridicată si are sintaxa:

(tip) expresie

Exemple de conversii explicite

```
float r;
r = 5 / 2;          // impartirea se face in domeniul int, deci r va fi 2.0

r = (float) 5 / 2; /* r va fi 2.5; pentru ca primul operand este de tip
float calculul se face în domeniul real */

int x = (int) r;        //x va fi 2
```

C++ suportă şi conversii de tip de genul *new-type*(*operand*):

```
medie = double(sum) / 100;  // echivalent cu (double)sum / 100
```

Capitolul IB.03. Funcții de intrare/ieşire în limbajul C

Cuvinte-cheie
Funcții de intrare/ieşire caractere, funcții de
intrare/ieşire şiruri de caractere, citire/scriere cu format

IB.03.1 Funcții de intrare/ieşire în C

În limbajul C, nu există instrucțiuni de intrare/ieşire (citire/scriere), tocmai pentru a mări portabilitatea limbajului. Pentru a realiza citiri şi scrieri se apelează funcții de intrare/ieşire din bibliotecile mediului de programare.

Pentru operații de citire a datelor inițiale şi de afişare a rezultatelor sunt definite funcții standard, declarate în fişierul antet *stdio.h.* Se vor prezenta şi funcțiile de intrare/ieşire nestandard cele mai uzuale, care sunt declarate în *conio.h*. Utilizarea acestor funcții într-un program, va presupune deci, includerea respectivelor fişiere antet.

În acest capitol vom prezenta numai acele funcții folosite pentru citire de la tastatură şi pentru afişare pe ecran, deci pentru lucru cu fişierele standard numite *stdin* şi *stdout*. Fişierele standard *stdin* şi *stdout* sunt fişiere text. Un fişier text este un fişier care conține numai caractere ASCII grupate în linii (de lungimi diferite), fiecare linie fiind terminată cu un terminator de linie format din unul sau două caractere. În sistemele Windows se folosesc două caractere ca terminator de linie: *\n* şi *\r*, adică *newline* (trecere la linie nouă) şi *return (*trecere la început de linie). În sistemele Unix/Linux se foloseşte numai caracterul *\n* (*newline*) ca terminator de linie, caracter generat de tasta *Enter*.

Funcțiile I/O (Input/Output - intrare/ieşire) din C pot fi grupate în câteva familii:
- **Funcții de citire/scriere caractere individuale: *getchar, putchar, getch, putch*;**
- **Funcții de citire/scriere linii de text: *gets, puts*;**
- **Funcții de citire/scriere cu format (cu conversie): *scanf, printf*.**

IB.03.2 Funcții de citire/scriere pentru caractere şi şiruri de caractere

Pentru operațiile I/O la nivel de caracter:
- **int getchar ();**
- **int putchar (int c);**
- **int getche ();**
- **int getch ();**
- **int putch (int c);**

Pentru operații I/O cu şiruri de caractere:
- **char * gets(char * s);**
- **int puts(const char * şir);**

Tabelul următor conține o descriere detaliată a funcțiilor pentru operațiile I/O la nivel de caracter:

Funcție	Exemple	Descriere	Rezultat
int getchar();	char c; c=getchar();	Citeşte un caracter	Returnează codul unui caracter citit de la tastatura sau valoarea *EOF* (**End Of File** - constantă simbolică definită în *stdio.h*, având valoarea -1) dacă s-a tastat Ctrl+Z.
int putchar(int c);	char c; putchar(c);	Afişează pe ecran caracterul transmis ca parametru	Returnează codul caracterului sau *EOF* în cazul unei erori.
int getche();	char c; c=getche();	Citeşte un caracter (aşteaptă apăsarea unei taste, chiar dacă în buffer-ul de intrare mai sunt caractere nepreluate) şi afişează caracterul pe ecran	Returnează *EOF* la tastarea lui Ctrl+Z, respectiv *CR* (*\r*, cu codul 13) la tastarea lui *Enter*.
int getch();	char c; c=getch();	Analog cu funcția de mai sus, dar caracterul nu se transmite în ecou (nu se afişează pe ecran).	
int putch(int c);	char c; putch(c);	Tipăreşte pe ecran caracterul transmis ca parametru	Returnează codul caracterului sau *EOF* în cazul unei erori.

Funcțiile *getch*, *putch*, *getche* nu sunt standard, de aceea este bine să se evite utilizarea lor!

Tabelul următor conține o descriere detaliată a funcțiilor pentru operațiile I/O cu şiruri de caractere:

Funcție	Exemple utilizare	Descriere	Rezultat
char * gets(char * s);	char sir[10]; gets(sir);	Citeşte caractere până la întâlnirea lui *Enter*; acesta nu se adaugă la şirul s. Plasează */0* la sfârşitul lui s. Obs: codul lui *Enter* e scos din buffer-ul de intrare.	Returnează adresa primului caracter din şir. Dacă se tastează CTRL+Z returnează *NULL*.
int puts(const char * s);	char sir[10]; puts(sir);	Tipăreşte şirul primit ca parametru, apoi *NewLine*, cursorul trecând la începutul rândului următor	Returnează codul ultimului caracter din şir sau *EOF* la insucces

IB.03.3 Funcții de citire/scriere cu format

Funcțiile *scanf* şi *printf* permit citirea cu format şi respectiv scrierea cu format pentru orice tip de date. Antetul şi descrierea acestor funcții sunt:

int printf (format, arg1, arg2, ...);

- **afişează pe ecran valorile expresiilor din lista argumentelor, conform formatului specificat;**
- **argumentele pot fi constante, variabile, expresii**
  - **dacă nu apare nici un argument, pe ecran se tipăresc doar caracterele din şirul format.**
- **formatul este un şir de caractere care trebuie să includă câte un descriptor de format pentru fiecare din argumente.**
  - **caracterele din format care nu fac parte din descriptori se tipăresc pe ecran.**

Returnează numărul de valori tipărite sau EOF în cazul unei erori.

int scanf (format, adr1, adr2, ...);

- **Citeşte caracterele introduse de la tastatură, pe care le interpretează conform specificatorilor din format, memorând valorile citite la adresele transmise ca parametri.**
- **Formatul este un şir de caractere care trebuie să includă câte un descriptor de format pentru fiecare dintre valorile citite.**
- **Adresele sunt pointeri sau adresele variabilelor ale căror valori se citesc;**
  - **Adresa unei variabile se obține folosind operatorul de adresare &, astfel: &nume_variabila**
  - **Valorile întregi sau reale consecutive introduse de la tastatură trebuie separate de cel puțin un spațiu (enter, spațiu, tab)**

Returnează numărul de valori citite sau EOF dacă s-a tastat Ctrl/Z.

IB.03.3.1 Funcția *printf*

Descriptorii de format admişi în funcția *printf* sunt:

Specificatori format	Descriere
%d	întreg zecimal cu semn
%i	întreg zecimal, octal (0) sau hexazecimal (0x, 0X)
%o	întreg în octal, fără 0 la inceput
%u	întreg zecimal fără semn
%x, %X	întreg hexazecimal, fără 0x/0X; cu a-f pt. %x, A-F pt. %X
%c	caracter
%s	şir de caractere, până la '\0' sau nr. de caractere dat ca precizie
%f, %F	real fără exponent; precizie implicită 6 pozitii; la precizie 0: fără punct real (posibil cu exponent)
%e, %E	numere reale cu mantisă şi exponent (al lui 10); precizie implicită 6 poz.; la precizie 0: fără punct
%g, %G	numere reale în format %f sau %e, funcție de valoare real, ca %e, %E dacă exp. < -4 sau precizia; altfel ca %f. Nu tipăreşte zerouri sau punct zecimal în mod inutil
%p	pointer, în formatul tipărit de printf
%ld, %li	numere întregi lungi

%lf, %le, %lg	numere reale în precizie dublă (*double*)
%Lf, %Le, %Lg	numere reale de tip *long double*
%%	caracterul procent

Exemple de utilizare *printf*:

```
printf ("\n");                      // trecere la o noua linie
printf ("\n Eroare \n");            // afişează „ Eroare ” pe o linie
                                    // şi trece la linia următoare

int a=3;
printf ("%d\n",a);
// afişează „3” ca întreg şi trece la linia următoare

int a=5, b=7;
printf ("a=%d  b=%d\n", a, b);
// afişează „a=5 b=7” şi trece la linia următoare

int g=30, m=5, s=2;
printf ("%2d grade %2d min %2d sec\n", g,m,s);
// afişează „30 grade  5 min   2 sec„ şi trece la linia următoare

int anInt = 12345;
float aFloat = 55.6677;
double aDouble = 11.2233;
char aChar = 'a';
char aStr[] = "Hello";

printf("The int is %d.\n", anInt);      // afişează:The int is 12345.
printf("The float is %f.\n", aFloat);   // afişează:The float is 55.667702.
printf("The double is %f.\n", aDouble);
     // afişează:The double is 11.223300.

printf("The char is %c.\n", aChar);     // afişează:The char is a.
printf("The string is %s.\n", aStr);    // afişează:The string is Hello.

printf("The int (in hex) is %x.\n", anInt);
    // afişează:The int (in hex) is 3039.

printf("The double (in scientic) is %e.\n", aDouble);
    // afişează: The double (in scientic) is 1.122330e+01.

printf("The float (in scientic) is %E.\n", aFloat);
   // afişează: The float (in scientic) is 5.566770E+01.
```

Programatorul trebuie să asigure concordanța dintre descriptorii de format şi tipul variabilelor sau expresiilor argument, deoarece funcțiile *scanf* şi *printf* nu fac nici o verificare şi nu semnalează neconcordanțe. Exemplele următoare trec de compilare (fără mesaj de eroare) dar afişează incorect valorile variabilelor:

```
int i=3;
float f=3.14;
printf ("%f \n", i);          // scrie 0.00 (în Dev-Cpp)
printf ("%i \n", f);          // scrie 1610612736  (Dev-Cpp)
```

Funcțiile *scanf* şi *printf* folosesc noțiunea de *câmp* (field): un câmp conține o valoare şi este separat de alte câmpuri prin spații. Fiecare descriptor de format poate conține mărimea câmpului, ca număr

întreg. Această mărime se foloseşte mai ales la afişare, pentru afişare numere pe coloane, aliniate la dreapta. În lipsa acestei informații mărimea câmpului rezultă din valoarea afişată.

Exemple:

```
printf("%d %d",a,b);          // 2 campuri separate prin blanc(spatiu)
printf("%8d8%d",a,b);         // 2 câmpuri alăturate de cate 8 caractere

int number = 123456;
printf("number=%d.\n", number);
// afişează number=123456. intr-un camp de 7 pozitii

printf("number=%8d.\n", number);
// afişează number=  123456. intr-un camp de 8 pozitii
printf("number=%3d.\n",  number);    /*  dimensiunea  campului  este  prea
mica;este ignorata.*/
// afişează number=123456.

double value = 123.14159265;
printf("value=%f;\n", value);
// afişează value=123.141593;
printf("value=%6.2f;\n", value);
// afişează value=123.14; intr-un camp de 6 pozitii, cu 2 zecimale
printf("value=%9.4f;\n", value);
// afişează value= 123.1416; intr-un camp de 9 pozitii, cu 4 zecimale
printf("value=%3.2f;\n", value);  /* dimensiunea campului este prea mica;
este ignorata. */
// afişează value=123.14;
```

Dacă nu se precizează mărimea câmpului şi numărul de cifre de la partea fracționară pentru numere, atunci funcția *printf* alege automat aceste valori:

- dimensiunea câmpului rezultă din numărul de caractere necesar pentru afişarea cifrelor, semnului şi altor caractere cerute de format;
- numărul de cifre de la partea fracționară este 6 indiferent dacă numerele sunt de tip *float* sau *double* sau *long double*, dacă nu este precizat explicit.

Se poate preciza numai mărimea câmpului sau numai numărul de cifre la partea fracționară. Exemple:

```
float a=1.; double b=0.0002; long double c=7.5; float d=-12.34;
printf ("%.0f  %20lf  %20.10Lf  %f  \n", a, b, c, d);
```

Specificând dimensiunea câmpului în care se afişează o valoare, putem realiza scrierea mai multor valori în coloane. Dacă valoarea de afişat necesită mai puține caractere decât este mărimea câmpului, atunci această valoare este aliniată la dreapta în câmpul respectiv.

Secvența următoare va scrie trei linii, iar numerele afişate vor apare într-o coloană cu cifrele de aceeaşi pondere aliniate unele sub altele:

```
int a=203, b=5, c=16;
printf ("%10d \n %10d \n %10d \n",a,b,c);
```

Formatul cu exponent (“%e”) este util pentru numere foarte mari, foarte mici sau despre ale căror valori nu se ştie nimic. Numărul este scris cu o mantisă fracționară (între 0 şi 10) şi un exponent al lui 10, după litera E (e).

La formatul “%g” *printf* alege între formatul “%f” sau “%e” în funcție de ordinul de mărime al numărului afişat: pentru numere foarte mari sau foarte mici formatul cu exponent, iar pentru celelalte formatul cu parte întreagă şi parte fracționară.

Între caracterul ‘%’ şi literele care desemnează tipul valorilor scrise mai pot apare, în ordine:

a) un caracter ce exprimă anumite opțiuni de scriere:
- (minus): aliniere la stânga în câmpul de lungime specificată
+ (plus): se afişează şi semnul '+' pentru numere pozitive
0 : numerele se completează la stânga cu zerouri pe lungimea *w*
 # : formă alternativă de scriere pentru numere

b)
- **un număr întreg *w* ce arată lungimea câmpului pe care se scrie o valoare, sau**
- **caracterul * dacă lungimea câmpului se dă într-o variabilă de tip int care precede variabila a cărei valoare se scrie.**

c) punct urmat de un întreg, care arată precizia (numărul de cifre de după punctul zecimal) cu care se scriu numerele neîntregi.

d) una din literele 'h', 'l' sau 'L' care modifică lungimea tipului numeric.

```
/* Exemplu de utilizare a opțiunii '0' pentru a scrie întotdeauna două
cifre, chiar şi pentru numere de o singură cifră: */
int ora=9, min=7, sec=30;
printf ("%02d:%02d:%02d\n",ora, min, sec); // scrie  09:07:30

//Exemplu ce afiseaza 10 spatii
printf("afisez 10 spatii: %*c",10,' ');

// Exemplu de utilizare a opțiunii '-' pentru aliniere şiruri la stânga:
char  a[] = "unu", b[] ="cinci", c[]= "sapte" ;
printf (" %-10s \n %-10s \n %-10s \n", a, b, c);

int i1 = 12345, i2 = 678;
printf("Hello, first int is %d, second int is %5d.\n", i1, i2);
//Hello, first int is 12345, second int is   678.
printf("Hello, first int is %d, second int is %-5d.\n", i1, i2);
//Hello, first int is 12345, second int is 678  .

char msg[] = "Hello";
printf("xx%10sxx\n", msg);    //xx     Helloxx
printf("xx%-10sxx\n", msg);   //xxHello     xx
```

În general trebuie să existe o concordanță între numărul şi tipul variabilelor şi formatul de citire sau scriere din funcțiile *scanf* şi *printf*, dar această concordanță nu poate fi verificată de compilator şi nici nu este semnalată ca eroare la execuție, dar se manifestă prin falsificarea valorilor citite sau scrise. O excepție notabilă de la această regulă generală este posibilitatea de a citi sau scrie corect numere de tip *double* cu formatul "%f" (pentru tipul *float*), dar nu şi numere de tip *long double* (din cauza diferențelor de reprezentare internă a exponentului).

IB.03.3.2 Funcția *scanf*

Descriptorii de format admişi în funcția *scanf* sunt:

Specificatori format	Descriere
%d	întreg zecimal cu semn
%i	întreg zecimal, octal (0) sau hexazecimal (0x, 0X)
%o	întreg în octal, precedat sau nu de 0
%u	întreg zecimal fără semn
%x, %X	întreg hexazecimal, precedat sau nu de 0x, 0X
%c	orice caracter; nu sare peste spații (doar " %c")

%s	şir de caractere, până la primul spațiu. Se adaugă '\0'.
%e, %E, %f, %F, %g, %G, %a, %A	real (posibil cu exponent)
%p	pointer, în formatul tipărit de printf
%ld, %li	numere întregi lungi
%lf	numere reale în precizie dublă (*double*)
%Lf	numere reale de tip *long double*
%[…]	şir de caractere din mulțimea indicată între paranteze
%[^…]	şir de caractere exceptând mulțimea indicată între paranteze
%%	caracterul procent

După cum spuneam mai sus, argumentele funcției *scanf* sunt de tip pointer şi conțin adresele unde se memorează valorile citite. Pentru variabile numerice aceste adrese se obțin cu operatorul de adresare (&) aplicat variabilei care primeşte valoarea citită.
Numerele citite cu *scanf* pot fi introduse pe linii separate sau în aceeaşi linie dar separate prin spații sau caractere *Tab*. Între numere succesive pot fi oricâte caractere separator (‘\n’,’\t’,’ ‘). Un număr se termină la primul caracter care nu poate apare într-un număr.

Exemple:

```
int n, a,b;
scanf("%d", &n);        // citeşte un întreg în variabila n
scanf("%d%d", &a,&b);   // citeşte doi întregi in a şi b
float rad;
scanf("%f", &rad);      // citeşte un numar real in rad
char c;
scanf("%c", &c);        // citeşte un caracter in c

// program test scanf
#include <stdio.h>

int main() {
   int anInt;
   float aFloat;
   double aDouble;

   printf("Introduceti un int: "); // Mesaj afisat
   scanf("%d", &anInt);            // citeşte un întreg în variabila anInt
   printf("Valoarea introdusa este %d.\n", anInt);

   printf("Introduceti un float: ");// Mesaj afisat
   scanf("%f", &aFloat);            // citeşte un float în variabila aFloat
   printf("Valoarea introdusa este %f.\n", aFloat);

   printf("Introduceti un double: ");// Mesaj afisat
   scanf("%lf", &aDouble);        // citeşte un întreg în variabila aDouble
   printf("Valoarea introdusa este %lf.\n", aDouble);

   return 0;
}
```

La citirea de şiruri trebuie folosit un vector de caractere, iar în funcția *scanf* se foloseşte numele vectorului (echivalent cu un pointer). Exemplu:

```
char s[100];
scanf("%s", s);   //uneori funcționează, dar este greșit:  scanf("%s, &s);
```

Chiar şi spațiile trebuie folosite cu atenție în şirul format din *scanf*. Exemplu de citire care poate crea probleme la execuție din cauza blancului din format:

```
scanf("%d  ",&a);  // corect este  scanf ("%d",&a);
```

Funcția *scanf* nu poate afişa nimic, de aceea pentru a precede introducerea de date de un mesaj (prin care este anunțat utilizatorul că se aşteaptă introducerea unei date) trebuie folosită secvența *printf*, *scanf*. Exemplu:

```
printf ("n= ");
scanf ("%d", &n);    // NU: scanf("n=%d", &n);
```

De reținut diferența de utilizare a funcțiilor *scanf* şi *printf*!
Exemplu:

```
scanf("%d%d", &a, &b);  // citeşte două numere întregi în a şi b
printf("%d %d", a, b);  // scrie valorile din a şi b separate de un spațiu
```

Alte exemple de utilizare a funcțiilor *scanf* şi *printf*:

```
int i;
float f;
double d;
scanf("%d%f%lf",&i, &f, &d); //citeste un intreg, un real si un double

char c;
printf("Rezultatul este: %c\n",c); //afiseaza un mesaj si un character

float a;
double b;
scanf("%f", &a);         // citire variabila float
printf("%5.2f",a);       // sunt afisate minim 5 caractere, maxim 2 zecimale
scanf("%lf", &b);        // citire variabila double
printf("%-4.2lf",b);     // afisare cu aliniere stanga
printf("%+4.2lf",b);     // afisare cu adaugare semn (+,-)
```

Observații
Toate funcțiile de citire menționate (excepție *getch*, *getche*) folosesc o zonă tampon (*buffer*) în care se adună caracterele tastate până la apăsarea tastei *Enter*, moment în care conținutul zonei buffer este transmis programului. În acest fel este posibilă corectarea unor caractere introduse greşit înainte de apăsarea tastei *Enter*.
Caracterul *\n* este prezent în zona buffer numai la funcțiile *getchar* şi *scanf*, dar funcția *gets* înlocuieşte acest caracter cu un octet zero, ca terminator de şir în memorie (rezultatul funcției "gets" este un şir terminat cu zero).
Funcția *scanf* recunoaşte în zona buffer unul sau mai multe *câmpuri* (*fields*), separate şi terminate prin caractere spațiu (blanc, '\n', '\r','\f'); drept consecință, preluarea de caractere din buffer se opreşte la primul spațiu alb, care poate fi şi caracterul terminator de linie. La următorul apel al funcției *getchar* sau *scanf* se verifica dacă în zona buffer mai sunt caractere, înainte de a aştepta introducerea de la tastatura şi va găsi caracterul terminator de linie rămas de la citirea anterioară.
Pentru ca un program să citească corect un singur caracter de la tastatură avem mai multe soluții:

- apelarea funcției fflush(stdin) înainte de oricare citire; această funcție goleşte zona buffer asociată tastaturii şi este singura posibilitate de acces la această zonă tampon.

- introducerea unei citiri false care să preia terminatorul de linie din buffer (cu *getchar*(), de exemplu).
- utilizarea funcției nestandard *getch* (declarată în *conio.h*), funcție care nu foloseşte o zonă buffer la citire
- citirea unui singur caracter ca un şir de lungime 1:

```
char c[2];        // memorie ptr un caracter şi pentru terminator de şir
scanf ("%1s",c);  // citeşte şir de lungime 1
```

Unele medii integrate închid automat fereastra în care se afişează rezultatele unui program. Pentru menținerea rezultatelor pe ecran vom folosi fie o citire falsă (pune programul în aşteptare) fie instrucțiunea: system("pause"); din stdlib.h.

IB.03.4 Fluxuri de intrare/ieşire in C++

În C++ s-a introdus o altă posibilitate de exprimare a operațiilor de citire-scriere, pe lângă funcțiile standard de intrare-ieşire din limbajul C. În acest scop se folosesc câteva clase predefinite pentru fluxuri de I/O (declarate în fişierele antet *iostream.h* şi *fstream.h*).
Un flux de date (*stream*) este un obiect care conține datele şi metodele necesare operațiilor cu acel flux. Pentru operații de I/O la consolă sunt definite variabile de tip flux, numite *cin* (console input) respectiv *cout* (console output).
Operațiile de citire sau scriere cu un flux pot fi exprimate prin metode ale claselor flux sau prin doi operatori cu rol de extractor din flux (>>) sau insertor în flux (<<). Atunci când primul operand este de un tip flux, interpretarea acestor operatori nu mai este cea de deplasare binară ci este extragerea de date din flux (>>) sau introducerea de date în flux (<<).
Operatorii << şi >> implică o conversie automată a datelor între forma internă (binară) şi forma externă (şir de caractere). Formatul de conversie poate fi controlat prin cuvinte cheie cu rol de "modificator".
Exemplu de scriere şi citire cu format implicit:

```
#include <iostream.h>
void main ( ) {
int n; char s[20];
cout << " n= ";  cin >> n;
cout << " un şir: "; cin >> s;  cout << s << "\n";
}
```

Într-o expresie ce conține operatorul << primul operand trebuie să fie *cout* (sau o altă variabilă de un tip *ostream*), iar al doilea operand poate să fie de orice tip aritmetic sau de tip *char** pentru afişarea şirurilor de caractere. Rezultatul expresiei fiind de tipul primului operand, este posibilă o expresie cu mai mulți operanzi (ca la atribuirea multiplă).
Exemplu:

```
cout << "x= " << x << "\n";
```

În mod similar, într-o expresie ce conține operatori >> primul operand trebuie să fie *cin* sau de un alt tip *istream*, iar ceilalți operanzi pot fi de orice tip aritmetic sau pointer la caractere. Exemplu:

```
cin >> x >> y;
```

Este posibil şi un control al formatului de scriere prin utilizarea unor *modificatori*, însă nu vom detalia aici acest aspect, deoarece nu vom folosi aceste facilități care țin de C++, ele fiind date doar ca titlu informativ.

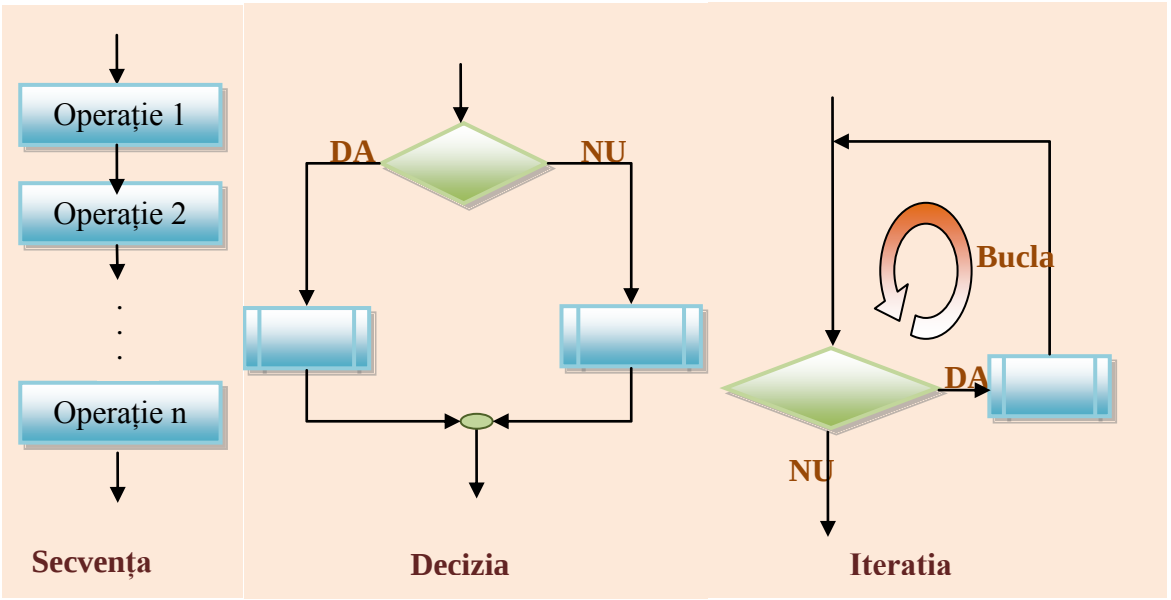
Capitolul IB.04. Instrucțiunile limbajului C

Cuvinte-cheie

Instrucțiunea expresie, instrucțiunea compusă – bloc, instrucțiunea if, instrucțiunea switch, instrucțiuni repetitive, instrucțiunea while, instrucțiunea for, instrucțiunea do, instrucțiunea break, instrucțiunea continue, terminarea programului: exit şi return

IB.04.1 Introducere

După cum spuneam, există trei tipuri de construcții de bază pentru controlul fluxului operațiilor: *secvența*, *decizia* (condiția) şi *iterația* (bucla, ciclu, repetiția), aşa cum sunt ilustrate mai jos.



IB.04.2 Instrucțiunea expresie

O expresie urmată de caracterul terminator de instrucțiune ’;’ devine o instrucțiune expresie.

Sintaxa:
expresie;

Cele mai importante cazuri de instrucțiuni expresie sunt:

Tip instrucțiune expresie	Descriere	Exemple
Instrucțiunea vidă	**conține doar terminatorul *’;’*** **este folosită pentru a marca absența unei prelucrări într-o altă instrucțiune (*if*, *while*, etc)**	for (i=0; i<10000; i++) ; /* temporizare, de 10000 de ori nu fac nimic */

Instrucțiunea de apelare a unei funcții	un apel de funcție terminat cu ‘;’	getchar();
		sqrt(a);
		system(”pause”);

Instrucțiunea de atribuire	o expresie de atribuire terminată cu ’;’	a=1;
		++a;
		c=r/(a+b);
		i=j=k=1;

Utilizarea neatentă a caracterului punct-şi-virgulă poate introduce uneori erori grave (nesemnalate de compilator), dar alteori nu afectează execuţia (fiind interpretat ca o instrucţiune vidă).

Exemple:

```
char  a = '1', b ='c';
printf (" %c \n %c \n", a, b);

int a,b,c,m,n,p=2;
// liniile de mai jos reprezinta instructiuni expresie
scanf("%d",&a);
// apel de functie, valoarea returnata de functie nu este folosita b = 5;
c = a > b ? a:b;
n = printf("%d %d %d\n",a,b,c); //valoarea returnata este memorata in n
p = a*b/c;
p++;
m = p+ = 5;
a+b;  /* valoarea expresiei nu este folosita - apare un avertisment ( warning ) la compilare: Code has no effect */
```

IB.04.3 Instrucţiunea compusă (bloc)

Pe parcursul elaborării programelor, intervin situaţii în care sintaxa impune specificarea unei singure operatii iar codificarea sa necesita mai multe – instrucţiuni; în acest caz se incadrează instrucţiunile între acolade, formând un bloc ce grupează mai multe instrucţiuni (şi declaraţii).

Sintaxa:

```
{
        declaraţii_variabile_locale_blocului   // opţionale, valabile doar în fişiere cpp!
        instrucţiuni
}
```

Observaţii:

- *Corpul* oricărei funcţii este un bloc;
- Instrucţiunile unui bloc pot fi de orice tip, deci şi alte instrucţiuni compuse; instrucţiunile bloc pot fi deci *incuibate;*
- *Un bl*oc corespunde structurii de control *secvenţă* de la schemele logice;
- O instrucţiune compusă poate să nu conţină nici o declaraţie sau instrucţiune între acolade; în general un astfel de bloc poate apare în faza de punere la punct a programului (funcţii cu corp vid);
- Acoladele nu modifică ordinea de execuţie, dar permit tratarea unui grup de instrucţiuni ca o singură instrucţiune în cadrul instrucţiunilor de control (*if, while, do, for* ş.a). Instrucţiunile de control au ca obiect, prin definiţie, o singură instrucţiune. Pentru a extinde domeniul de acţiune al acestor instrucţiuni la un grup de operaţii se foloseşte instrucţiunea compusă.
- Un bloc poate conţine şi doar o singură instrucţiune, aceasta pentru punerea în evidenţă a terminării acţiunii anumitor instrucţiuni.
- Un bloc nu trebuie terminat cu *';'* dar nu este *gre*şit *dacă se fol*oseşte (este interpretat ca instrucţiune vidă).

  Exemplu:
  if(a>b)
  {max=a;

if(c>a) max=c;
};

- Dacă un bloc conține doar instrucțiuni expresie, el se poate înlocui cu o instrucțiune expresie în care expresiile inițiale se separă prin operatorul secvențial.

Exemple:

```
{ int t; t=a; a=b; b=t; }     // schimba a şi b prin t
// sau:
{
      int t;
      t=a;
      a=b;
      b=t;
}

//Blocul:
{
      a++;
      c=a+ --b;
      printf("%d\n",c);
}
// este echivalent cu instructiunea expresie:
a++, c=a+ --b, printf("%d\n",c);
```

IB.04.4 Instrucțiuni de decizie (condiționale)

Există următoarele tipuri de instrucțini de decizie *if-then*, *if-then-else*, *if încuibat* (*if-elseif-elseif-...-else*), *switch-case.*

IB.04.4.1 Instrucțiunea *if*

Instrucțiunea introdusă prin cuvântul cheie *if* exprimă o decizie binară şi poate avea două forme: o formă fără cuvântul *else* şi o formă cu *else*:

Sintaxa:

```
If (expresie)
      instructiune1
else
      instructiune2

SAU:
If (expresie)
      instructiune
```

Semantica:

Se evaluează expresie; dacă valoarea ei este adevărat (diferită de 0) se execută instrucțiune1, altfel, dacă există ramura else, se execută instrucțiune2.

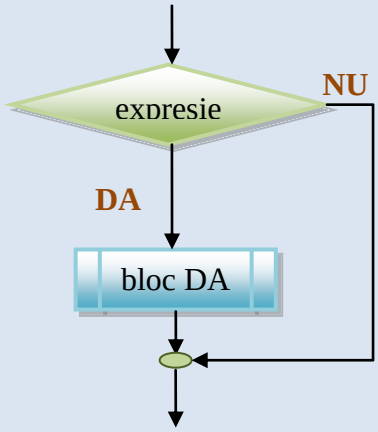
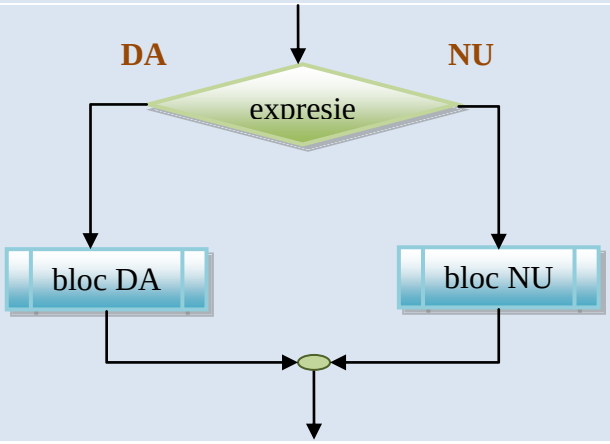
Observații:

- Instrucțiunea corespunde structurii de control *decizie* din schemele logice;
- Pentru ca programele scrise să fie cât mai clare este bine ca instrucțiunile corespunzătoare lui if si else să fie scrise pe liniile următoare şi deplasate spre dreapta, pentru a pune în evidentă structurile şi modul de asociere între *if* şi *else*. Acest mod de scriere permite citirea corectă a unor cascade de decizii.
- Valoarea expresiei dintre paranteze se compară cu zero, iar instrucțiunea care urmează se va executa numai atunci când expresia are o valoare nenulă. În general expresia din instrucțiunea *if* reprezintă o condiție, care poate fi adevărată (valoare nenulă) sau falsă (valoare nulă). De obicei expresia este o expresie relațională (o comparație de valori

numerice) sau o expresie logică în care sunt combinate mai multe expresii relaționale, dar poate fi orice expresie cu rezultat numeric.

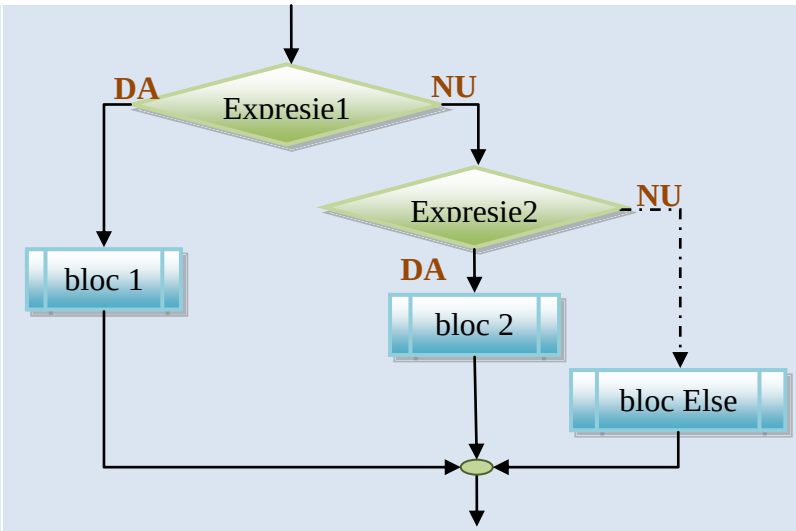
- Instrucțiunea corespunzătoare valorii adevărat sau fals, poate fi orice instrucțiune C:
  - instrucțiune expresie terminată cu simbolul ;
  - instrucțiunea bloc (atunci când trebuie executate mai multe prelucrări)
  - alta instrucțiune de decizie - deci instrucțiunile if-else pot fi incluse una în alta; fiecare else corespunzând ultimului if, fără pereche.
- O problemă de interpretare poate apare în cazul a două (sau mai multe) instrucțiuni *if* incluse, dintre care unele au alternativa *else*, iar altele nu conțin pe *else*. Regula de interpretare este aceea că *else* este asociat cu cel mai apropiat *if* fără *else* (dinaintea lui).

O sinteză a celor trei moduri de utilizare a lui *if* precum şi fluxul de operații în schemă logică sunt date în cele ce urmează:

Sintaxă	Flux operații
// if-then **if (*expresie*) {** **bloc_DA;** **}**	
// if-then-else **if (*expresie*)** **bloc_DA ;** **else** **bloc_NU ;**	

```
// if inclus

if (expresie_1 )
   bloc_1 ;
else
  if (expresie_2 )
    bloc_2 ;
  else
    if (expresie_3 )
     bloc_3 ;
    else
     if (expresie_4 )
       ......
        else
          bloc_Else ;
```



Exemple de utilizare a lui *if*:

Sintaxă	Exemple

`// if-then`
 	
`if ( expresie )`
`  bloc_DA;`
	`if (nota >= 5) {`
	`  printf("Congratulation!\n");`
	`  printf("Keep it up!\n");`
	`  }`

`// if-then-else`
 	
`if ( expresie )`
`  bloc_DA ;`
`else`
`  bloc_NU ;`
	`if (nota >= 5) {`
	`  printf("Congratulation!\n");`
	`  printf("Keep it up!\n");`
	`} else`
	`  printf("Try Harder!\n");`

`// if incuibat`
 	
`if (expresie_1 )`
`  bloc_1 ;`
`else`
`  if (expresie_2 )`
`  bloc_2 ;`
`  else`
`  if (expresie_3 )`
`  bloc_3 ;`
`  else`
`  if (expresie_4 )`
`  ......`
`  else`
`  bloc_Else ;`
	`if (nota >= 80)`
	`  printf("A\n");`
	`else`
	`  if (nota >= 7)`
	`  printf("B\n");`
	`  else`
	`  if (nota >= 6)`
	`  printf("C\n");`
	`  else`
	`  if (nota >= 5)`
	`  printf("D\n");`
	`  else`
	`  printf("E\n");`

Exemple:

```
/* urmatoarele trei secvente echivalente verifica daca trei valori pot reprezenta lungimile laturilor unui triunghi */
```

- ```
  if(a<b+c && b<a+c && c<a+b)
        puts("pot fi laturile unui triunghi");
  else
        puts("nu pot fi laturile unui triunghi");
  ```
- ```
  if(a<b+c && b<a+c && c<a+b)
        ;
  ```

```
      /*pentru conditie adevarata nu se executa nimic: apare instructiunea
      vida                                                                */
      else
            printf("nu ");
      puts("pot fi laturile unui triunghi");
```

- `if(!(a<b+c && b<a+c && c<a+b))  // sau if(a>=b+c || b>=a+c || c>=a+b)`

```
            printf("nu ");
      puts("pot fi laturile unui triunghi");

// Pentru gruparea mai multor instrucțiuni folosim o instrucțiune bloc:
      if ( a > b) { t=a; a=b; b=t; }

/* Pentru comparația cu zero nu trebuie neapărat folosit operatorul de
inegalitate (!=), deşi  folosirea lui poate face codul sursă mai clar:*/
      if (d) return;          //  if (d != 0) return;

// determinare minim dintre doua numere
      if ( a < b)
            min=a;
      else
            min=b;

/* Pentru a grupa o instrucțiune if-else care conține un if fără else
utilizăm o instrucțiune bloc:*/
      if ( a == b ) {
            if (b == c)
                  printf ("a==b==c \n");
      }
      else
            printf (" a==b şi b!=c \n");

// Expresia conținută în instrucțiunea if poate include şi o atribuire:
      if ( d = min2 - min1) printf(„%d”,d);
      //se atribuie lui d o valoare apoi aceasta se compara cu zero
/*Instrucțiunea anterioară poate fi derutantă la citire şi chiar este
semnalată cu avertisment de multe compilatoare, care presupun că s-a
folosit eronat atribuirea în loc de comparație la egalitate (o eroare
frecventă):*/

      if (i=0) printf( “Variabila i are valoarea 0”);
      else printf( “Variabila i are o valoare diferita de 0”);
```

IB.04.4.2 Instrucțiunea de selecție *switch*

Selecția multiplă (dintre mai multe cazuri posibile), se poate face cu mai multe instrucțiuni *if* incluse unele în altele sau cu instrucțiunea *switch*. Instrucțiunea *switch* face o enumerare a cazurilor posibile (fiecare precedat de cuvântul cheie *case*) folosind o expresie de selecție, cu rezultat întreg. Forma generală este:

Sintaxa:

```
switch (expresie) {
      case c1: prelucrare_1
      case c2: prelucrare_2
      case cn: prelucrare_n
      default: prelucrare_x
}
```

Unde:

- **expresie - de tip întreg, numită expresie selectoare**
- **c1, cn - constante sau expresii constante întregi (inclusiv „char”)**
- **orice prelucrare constă din 0 sau mai multe instrucțiuni**

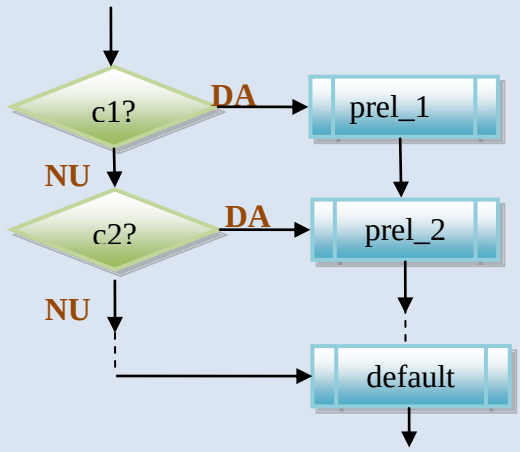
- **default e opțional, corespunde unor valori diferite de cele n etichete**

Semantica:
Se evaluează expresie; dacă se găseşte o etichetă având valoarea egală cu a expresiei, se execută atât secvența corespunzătoare acelui caz cât şi secvențele de instrucțiuni corespunzătoare tuturor cazurilor care urmează (chiar dacă condițiile acestora nu sunt îndeplinite) inclusiv ramura de *default*!
Această interpretare permite ca mai multe cazuri să folosească în comun aceleaşi operații. Cazul *default* poate lipsi; în cazul în care avem ramura *default*, se intră pe această ramură atunci când valoarea expresiei de selecție diferă de toate cazurile enumerate explicit.
Observație:
Deseori cazurile enumerate se exclud reciproc; pentru aceasta fiecare secvență de instrucțiuni trebuie să se termine cu *break*, pentru ca după selecția unui caz să se execute doar prelucrarea corespunzatoare unei etichete, nu şi cele următoare:

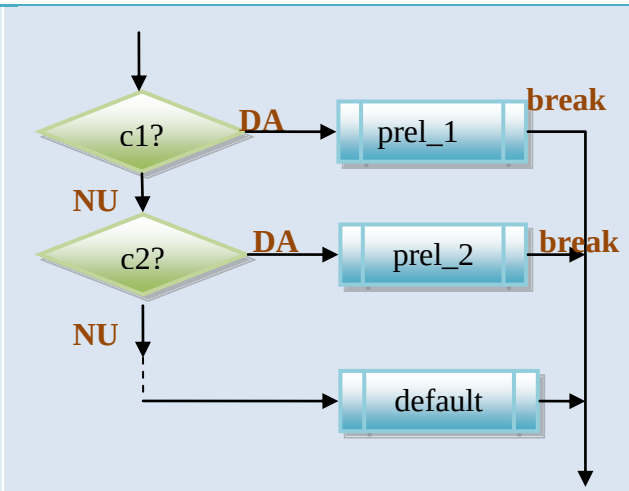
```
switch (expresie) {
        case c1:        prelucrare_1
                        break;
        case c2:        prelucrare_2
                        break;
        case cn:        prelucrare_n
                        break;
        default:        prelucrare_x ;
}
```

O sinteză a celor două moduri de utilizare a lui *switch* precum şi fluxul de operații în schemă logică este dat în cele ce urmează:

Sintaxă	Flux operații

`// switch-case`
`switch (expresie) {`
`  case c1:`
`  prelucrare_1`
`  case c2:`
`  prelucrare_2`
`  case cn:`
`  prelucrare_n`
`  default:`
`  prelucrare_x`
`}`
	

```
// switch-case
switch (expresie) {
   case c1:
            prelucrare_1
            break;
   case c2:
            prelucrare_2
            break;
   case cn:
            prelucrare_n
            break;
   default:
            prelucrare_x
}
```



Exemple

```
//calculeaza rezulatul expresiei num1 oper num2
char oper; int num1, num2, result;
......
switch (oper) {
   case '+':
      result = num1 + num2; break;
   case '-':
      result = num1 - num2; break;
   case '*':
      result = num1 * num2; break;
   case '/':
      result = num1 / num2; break;
   default:
      printf("Operator necunoscut");
}


// determina nr de zile dintr-o lună a unui an nebisect
switch (luna) {
      // februarie
      case 2: zile=28; break;
      // aprilie, iunie,..., noiembrie
      case 4: case 6: case 9: case 11: zile =30; break;
      // ianuarie, martie, mai,.. decembrie,celelalte (1,3,5,..)
      default: zile=31;
}
```

IB.04.5. Instrucțiuni repetitive

Există trei tipuri de instrucțiuni de ciclare (bucle, iterații): *while*, *for* şi *do-while*.

IB.04.5.1 Instrucțiunea *while*

Instrucțiunea *while* exprimă structura de ciclu cu condiție inițială şi cu număr necunoscut de paşi; are forma următoare:

Sintaxa:

```
while (expresie)
      instructiune
```

Semantica

Se evaluează expresie; dacă valoarea ei este adevărat (diferită de 0) se execută instructiune, după care se evaluează din nou expresie; daca valoarea este 0, se trece la instrucțiunea următoare. Efectul este acela de executare repetată a instrucțiunii conținute în instrucțiunea *while* cât timp expresia din paranteze are o valoare nenulă (este adevărată). Este posibil ca numărul de repetări să fie zero dacă expresia are valoarea zero de la început.

Observatii:

- Instrucțiunea while corespunde structurii repetitive cu test inițial de la schemele logice;
- În general *expresie* conține variabile care se modifică în instructiune, astfel încât expresie să devină falsă, deci ciclarea să nu se facă la infinit;
- În unele programe se poate să apară
  while(1)
  instructiune
  Atunci, corpul ciclului poate să conțină o instrucțiune de ieşire din ciclu, altfel tastarea Ctrl/Break întrerupe programul;
- Ca şi în cazul altor instrucțiuni de control, este posibil să se repete nu doar o instrucțiune ci un bloc de instrucțiuni;

Exemple:

```
// cmmdc prin incercari succesive de posibili divizori, presupunem a>b

d = b;                 // divizorul maxim posibil este minimul dintre a şi b
while ( a%d || b%d )   // repeta cat timp nici a nici b nu se divid prin d
d = d -1;        // incearca alt numar mai mic
}
```

În exemplul anterior, dacă a=8 şi b=4 atunci rezultatul este d=4 şi nu se execută niciodată instrucțiunea din ciclu (d=d-1).

```
// determinare cmmdc prin algoritmul lui Euclid. While cu instructiune bloc
while (a%b > 0) {
r = a % b;
a = b;
b = r;
}// la ieşirea din ciclu b este cmmdc
```

Expresia din instrucțiunea *while* poate să conțină atribuiri sau apeluri de funcții care se fac înainte de a evalua rezultatul expresiei:

```
// algoritmul lui Euclid rescris
while (r=a%b) {
a=b;
b=r;
}
```

IB.04.5.2 Instrucțiunea *for*

Instrucțiunea *for* din C permite exprimarea compactă a ciclurilor cu condiție inițială sau a ciclurilor cu număr cunoscut de paşi şi are forma:

Sintaxa:

for (expresie1; expresie2; expresie3)
instructiune

Semantica:
Se evaluează expresie1 care are rol de *inițializare*; se evaluează apoi *expresie2*, cu rol de condiție - dacă valoarea ei este adevărat (diferită de 0) se execută *instructiune - corpul ciclului*, după care se evaluează *expresie3*, cu rol de *actualizare*, apoi se evaluează din nou *expresie2*; dacă valoarea este 0, se trece la instrucțiunea următoare. Cu alte cuvinte, *instructiune* se execută atâta timp cât *expresie2* este adevărată, deci de 0 sau mai multe ori.
Efectul acestei instrucțiuni este echivalent cu al secvenței următoare:

```
expresie1;                  // operații de inițializare
while (expresie2) {         // cat timp exp2 !=0 repeta
  instrucțiune;                     // instrucțiunea repetata
  expresie3;                // o instrucțiune expresie
}
```

Observații:

- Instructiunea for permite o scriere mult mai compactă decât celelalte două instrucțiuni de ciclare, fiind foarte des utilizată în scrierea programelor;
- Oricare din cele trei expresii poate lipsi, dar separatorul ; rămâne. Absența expresie2 echivalează cu condiția adevărat, deci 1; in tabelul de mai jos sunt date echivalențele cu instrucțiunea while, pentru cazuri când expresii din sintaxa for lipsesc:

for	while
`for(;expresie;) instructiune`	`while(expresie) instructiune`
`for(;;) instructiune`	`while(1) instructiune`

- Cele trei expresii din instrucțiunea *for* sunt separate prin ';' deoarece o expresie poate conține operatorul virgulă. Este posibil ca prima sau ultima expresie să reunească mai multe expresii separate prin virgule;
- Este posibilă mutarea unor instrucțiuni din ciclu în paranteza instrucțiunii *for*, ca expresii, şi invers - mutarea unor operații repetate în afara parantezei.
- Un caz particular al instrucțiunii for este instrucțiunea de ciclare cu contor numărul de cicluri fiind (val_finala-val_initiala)/increment:

```
for ( var_contor = val_initiala; var_contor <= val_finala; var_contor += increment )
        instructiune
```

- Nu se recomandă modificarea variabilei *contor* folosită de instrucțiunea *for* în interiorul ciclului, prin atribuire sau incrementare;
- Pentru ieşire forțată dintr-un ciclu se folosesc instrucțiunile *break* sau *return*;

Exemple:

```
// ştergere linii ecran
for (k=1;k<=24;k++)
      putchar('\n');    // avans la linie noua

// alta secvenţa de ştergere ecran
for (k=24;k>0;k--)
      putchar('\n');

// calcul factorial de n
for (nf=k=1 ; k<=n ; k++)  nf = nf * k;

// alta varianta de calcul pentru factorial de n
for (nf=1, k=1 ; k<=n ; nf=nf * k, k++) ;  // repeta instr. vida
```

IB.04.5.3 Instrucţiunea *do*

Instrucţiunea *do-while* se foloseşte pentru exprimarea ciclurilor cu condiţie finală, cicluri care se execută cel puţin o dată. Forma uzuală a instrucţiunii *do* este următoarea:

Sintaxa:

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

Semantica:

Se execută *instrucţiune* - *corpul ciclului*, apoi se evaluează *expresie* care are rol de condiţie - dacă valoarea ei este adevărat (diferită de 0) se execută *instrucţiune*, după care se evaluează din nou expresie; dacă valoarea este 0, se trece la instrucţiunea următoare.

Cu alte cuvinte, *instrucţiune* se execută atâta timp cât *expresie* este adevărată; ca observaţie, *instrucţiune* se execută cel puţin o dată.

Observaţii:

- Instrucţiunea echivalează cu structura repetitivă cu test final de la scheme logice;
- Instrucţiunea do-while se utilizează în secvenţele în care se ştie că o anumită prelucrare trebuie executată cel puţin o dată;
- Spre deosebire de *while*, în ciclul *do* instrucţiunea se execută sigur prima dată chiar dacă expresia este falsă. Există şi alte situaţii când instrucţiunea *do* poate reduce numărul de instrucţiuni, dar în general se foloseşte mult mai frecvent instrucţiunea *while*.

Exemplu:

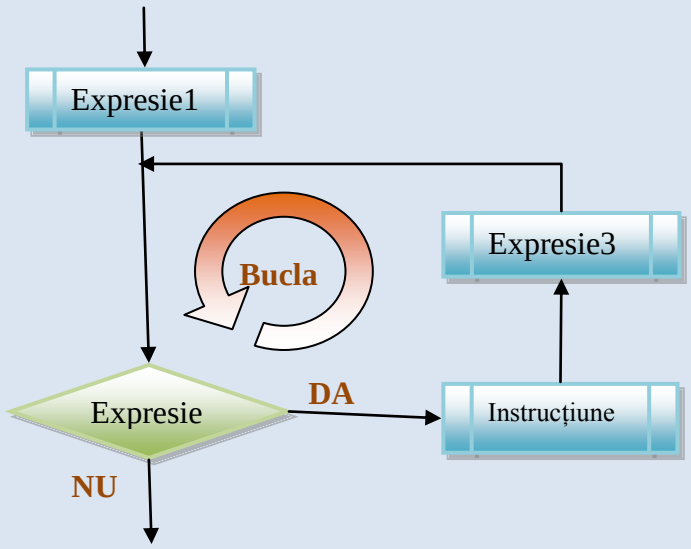
```
// calcul radical din x prin aproximatii succesive
r2=x;                          // aproximatia iniţiala
do {
      r1=r2;                   // r1 este aproximatia veche
      r2=(r1+x/r1) / 2;        // r2 este aproximatia mai noua
} while ( abs(r2-r1)) ;        // pana cand r2-r1 este o valoare foarte mica
```

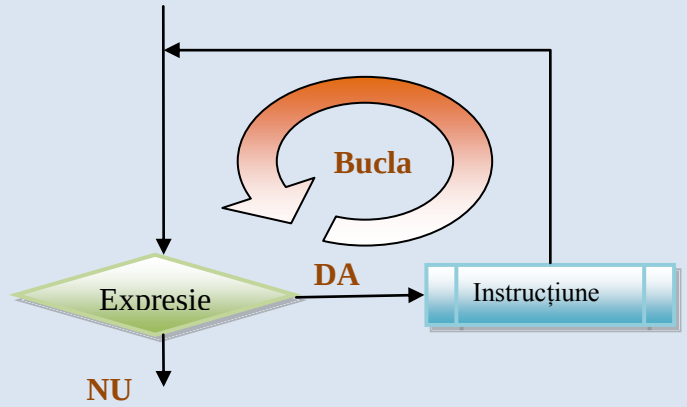
Un ciclu *do* tipic apare la citirea cu validare a unei valori, citire repetată până la introducerea corectă a valorii respective:

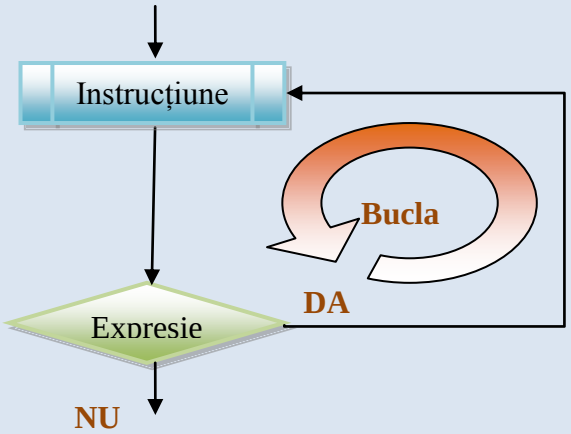
```
do {
printf ("n (<1000): "); // n trebuie sa fie sub 1000
scanf("%d", &n);
if ( n <=0 || n>=1000)
      printf (" Eroare la valoarea lui n ! \n");
} while (n>=1000) ;
```

IB.04.5.4 Sinteza instrucțiunilor repetitive

Sintaxă	Flux operații

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

`// while-do`
 	
`while ( expresie )`
`  instrucțiune`
	

`// do-while`
 	
`do {`
`  instrucțiune`
`}`
`while ( expresie ) ;`
	

Exemple scrise utilizând cele trei instrucțiuni repetitive:
1. Suma primelor 1000 de numere naturale

2. Secvențe echivalente care citesc cu validare o variabilă - în urma citirii, variabila întreagă trebuie să aparțină intervalului *[inf,sup]*:

Instrucțiune	Exemple

for	`// Suma de la 1 la 1000`
	`int suma = 0;`
	`for (int nr = 1; nr <= 1000; ++nr) {`
	`  suma += nr;`
	`}`
	
	`// Citire cu validare in intervalul [inf, sup]`
	`puts("Valoare");`
	`scanf("%d",&var);`
	`for(;var < inf || var > sup;){`
	`  puts("Valoare");`
	`  scanf("%d",&var);`
	`}`
	
	`// Citire cu validare in intervalul [inf, sup]`
	`for(puts("Valoare"),scanf("%d",&var);var < inf || var > sup;){`
	`  puts("Valoare");`
	`  scanf("%d",&var);`
	`}`
	
	`// Citire cu validare in intervalul [inf, sup]`
	`for(puts("Valoare"), scanf("%d",&var); var<inf || var>sup;`
	`  puts("Valoare"), scanf("%d",&var));`
	
	`// Citire cu validare in intervalul [inf, sup]`
	`for( ;puts("Valoare"), scanf("%d",&var), var<inf || var>sup;);`

while	`// Suma de la 1 la 1000`
	`int suma = 0, nr = 1;`
	`while (nr <= 1000) {`
	`  suma += nr;`
	`  ++nr;`
	`}`
	
	`// Citire cu validare in intervalul [inf, sup]`
	`puts("Valoare");`
	`scanf("%d",&var);`
	`while ( var < inf || var > sup){`
	`  //valoare invalida, se reia citirea`
	`puts("Valoare");`
	`scanf("%d",&var);`
	`}`
	
	`// Citire cu validare in intervalul [inf, sup]`
	`while( puts("Valoare"), scanf("%d",&var),var<inf ||var>sup);`

do while	`// Suma de la 1 la 1000`
	`int suma = 0, nr = 1;`
	`do {`
	`  suma += nr;`
	`  ++nr;`
	`} while (nr <= 1000);`
	
	`// Citire cu validare in intervalul [inf, sup]`
	`do{`
	`puts("Valoare");`
	`scanf("%d",&var);`
	`}while( var<inf || var>sup);`

IB.04.6. Instrucțiunile *break* şi *continue*

Instrucțiunea *break* determină ieşirea forțată dintr-un ciclu - adică ieşirea din corpul celei mai apropiate instrucțiuni *while, for, do-while* - sau dintr-un *switch* care o conține, şi trecerea la execuția instrucțiunii următoare.
Sintaxa instrucțiunii este simplă:

Sintaxa:
break;

Semantica
Efectul instrucțiunii *break* este un salt imediat după instrucțiunea sau blocul repetat prin *while, do, for* sau după blocul *switch*.

Observații:

- Un ciclu din care se poate ieşi fie după un număr cunoscut de paşi fie la îndeplinirea unei condiții (ieşire forțată) este de obicei urmat de o instrucțiune *if* care stabileşte cum s-a ieşit din ciclu: fie după numărul maxim de paşi, fie mai înainte, datorită satisfacerii condiției.

Exemplu:

```
// verifica daca un numar dat n este prim
for (k=2; k<n;k++)
      if ( n%k==0)  break;
//daca gasim un divizor, iesim, n nu este prim!
if (k==n) printf ("prim \n"); /*s-a iesit normal din ciclu-nu are
divizor*/
else printf ("neprim \n"); /*s-a iesit fortat prin break - are divizor */
```

- Utilizarea instrucțiunii *break* poate simplifica expresiile din *while* sau *for* şi poate contribui la urmărirea mai uşoară a programelor, deşi putem evita instrucțiunea *break* prin complicarea expresiei testate în *for* sau *while*. Secvențele următoare sunt echivalente:

```
//se iese cand e este diferita de 0
for (k=0 ; k<n; k++)
if (e) break;

for (k=0 ; k<n && !e ; k++);
```

Instrucțiunea *continue* este mai rar folosită față de *break.*

Sintaxa:
continue;

Semantica
Efectul instrucțiunii *continue* este opirea iterației curente a ciclului şi un salt imediat la prima instrucțiune din ciclu, pentru a continua cu următoarea iterație. Nu se iese în afara ciclului, ca în cazul instrucțiunii *break.*

În exemplu următor se citeşte repetat un moment de timp dat sub forma ora, minut, secundă până la introducerea unui momnet corect (care are toate cele 3 componente: h, m s şi pentru care ora (h) se incadrează între 0 şi 24, minutele şi secundele (m, s) între 0 şi 59. Este realizată validarea doar pentru oră:

```
int h,m,s;
int corect=0;  // initial nu avem date corecte - nu avem de fapt deloc date

while ( ! corect ) {
// atata timp cat nu s-au citit date corecte
```

```
      printf (" ore, minute, secunde: ");
      if ( scanf("%i%i%i", &h, &m, &s) !=3 ) {
      //nu s-au citit 3 nr intregi
            printf (" Eroare - insuficiente date numerice\n");
            fflush(stdin);    //stergere buffer de intrare
            continue;  // salt peste instructiunile urmatoare, reia citirea
      }
      if (h < 0 || h > 24) {
            printf (" Valoare incorecta pentru ora!\n");
            fflush(stdin);   //stergere buffer de intrare
            continue;  // salt peste instructiunile urmatoare, reia citirea
            }
      .... // testare m si s intre 0 si 59
      corect=1;
}
```

Observații:

Uneori se recomandă să evităm utilizarea instrucțiunilor *break* şi *continue* deoarece programele care le folosesc sunt mai greu de citit şi de înțeles. Întotdeauna putem scrie acelaşi program fără să folosim *break* şi *continue*.

Exemplu:

```
// Suma de la 1 la n, excluzand  11, 22, 33,...
int n = 100;
int suma = 0;
for (int nr = 1; nr <= n; nr++) {
   if (nr % 11 == 0) continue;  /* sare peste restul corpului buclei şi
                                   trece la urmat. iteraţie - nr+1 */
   suma += nr;          // aici ajung doar daca nr nu e divizibil cu 11
}

// Este mai bine să rescriem bucla for astfel:
for (int nr = 1; nr <= n; nr++) {
   if (nr % 11 != 0) suma += nr;
}
```

IB.04.7. Terminarea programului

Un program se termină în mod normal în momentul în care s-au executat toate instrucțiunile sale. Dacă dorim să forțăm terminarea lui, putem folosi funcția *exit* sau instrucțiunea *return*.

Funcția ***exit*** are următoarea sintaxă:

Sintaxa:

```
      exit();
sau:
      exit(int codIesire);
```

Semantica:

Termină programul şi returnează controlul sistemului de operare (OS). Prin convenție, returnarea codului 0 indică terminarea normală a programului, în timp ce o valoare diferită de zero indică o terminare *anormală*.

Exemplu:

```
if (nrErori > 10) {
   printf( "prea multe erori!\n");
```

```
    exit(1);          // Terminarea programului
}
```

Instrucțiunea ***return*** are următoarea sintaxă:

Sintaxa:
return;
sau:
return expresie;

Semantica:
Se revine din funcția care conține instrucțiunea, în cea apelantă, la instrucțiunea următoare apelului; se returnează valoarea expresiei pentru cazul al doilea.
Putem folosi instrucțiunea "`return valoareReturnata;`" în funcția `main()` pentru a termina programul.

Exemplu:

```
int main() {
   ...
   if (nrErori > 10) {
      printf( "prea multe erori!\n");
      return 1;         // Termina programul si reda controlul OS
   }
   ...
}
```

În continuare, găsiți două anexe, una cu sfaturi practice pentru devoltarea programelor C (good practices) şi modul de depanare a programelor C în Netbeans şi CodeBlocks, iar cea de-a doua cu programele din cursul 1 rezolvate în C.

IB.04.8. Anexa A. Sfaturi practice pentru devoltarea programelor C. Depanare

Este important să scriem programe care produc rezultate corecte, dar de asemenea este important să scriem programe pe care alții (şi chiar noi peste câteva zile) să le putem înțelege, astfel încât să poată fi uşor întreținute. Acesta este ceea ce se numeşte un program bun.
Iată câteva sugestii:

- Respectă convețtia stabilită deja la proiectul la care lucrezi astfel încât întreaga echipă să respecte aceleaşi reguli.
- Formatează codul sursă cu indentare potrivită, cu spații şi linii goale. Foloseşte 3 sau 4 spații pentru indentare şi linii goale pentru a marca secțiuni diferite de cod.
- Alege nume bune şi descriptive pentru variabile si funcții: coloană, linie, xMax, numElevi. Nu folosiți nume fără sens pentru variabile, cum ar fi a, b, c, d. Evitați nume de variabile formate doar dintr-o literă (mai uşor de scris dar greu de înțeles), excepție făcând nume uzuale cum ar fi coordonatele x, y, z şi nume de index precum i.
- Scrie comentarii pentru bucățile de cod importante şi complicate. Comentează propriul cod cât de mult se poate.
- Scrie documentația programului în timp ce scrii programul.
- Evită construcțiile nestructurate, cum ar fi break şi continue, deoarece sunt greu de urmărit.

Erori în programare
Există trei categorii de erori în progamare:

- Erori de compilare (sau de sintaxă): pot fi reparate uşor.
- Erori de rulare: programul se opreşte prematur fără a produce un rezultat – de asemenea se repară uşor.
- Erori de logică: programul produce rezultate eronate. Eroarea este uşor de găsit dacă rezultatele sunt eronate mereu. Dar dacă programul produce de asemenea rezultate corecte cât şi rezultate eronate câteodată, eroarea este foarte greu de identficat.

  Astfel de erori devin foarte grave dacă nu sunt detectate înainte de utilizarea efectivă a programului în mediul său de operare. Implementarea unor programe bune ajută la mimimizarea şi detectarea acestor erori. De asemenea, o strategie de testare bună este necesară pentru a certifica corectitudinea programului.

Programe de depanare
Exista câteva tehnici de depanare a programaelor:

1. Uită-te mult la cod (inspectează codul)! Din păcate, erorile nu o să-ți sară în ochi nici dacă te uiți destul de mult.
2. Nu închide consola de erori când apar mesaje pretinzând că totul este în regulă. Analizează mesajele de eroare! Asta ajută de cele mai multe ori.
3. Inserează în cod afişări de variabile în locuri potrivite pentru a observa valori intermediare. Este folositor pentru programe mici, dar la programe complexe îşi pierde din eficiență.
4. Foloseşte un depanator grafic. Aceasta este cea mai eficientă metodă. Urmăreşte execuția programului pas cu pas urmărind valorile variabilelor.
5. Foloseşte unelte avansate pentru a descoperi folosirea ineficientă a memoriei sau nealocarea ei.
6. Testează programul cu valori de test utile pentru a elimina erorile de logică.

Testarea programului pentru a vedea dacă este corect
Cum te poți asigura că programul tău produce rezultate corecte mereu? Este imposibil să încerci toate variantele chiar şi pentru un program simplu. Testarea programului foloseşte de obicei un set de teste reprezentative, concepute pentru a detecta clasele de erori majore.
În continuarea acestui material găsiți modul de depanare a programelor C în Netbeans:
Ecuatia de grad 1
Suma primelor n numere naturale
şi în CodeBlocks:
Inversul unui numar natural
Interschimbarea valorilor- Problema paharelor

IB.04.9. Anexa B. Programele din capitolul IB.01 rezolvate în C

Probleme rezolvate în limbajul C

Problema 1: Problema cu functia F(x)

```
/** Includem stdio.h pentru a putea folosi functiile de citire/scriere. */
#include <stdio.h>

int main()
{
    /* Declaram variabilele. */
```

```
    int i, x;

    /* Citim cele 100 de valori, utilizand functia scanf, si calculam valoarea
     * functiei. */
    for(i = 0; i < 100; i++) {
        /* Atentie la string-ul de formatare si la caracterul '&'! */
        scanf("%d", &x);
        if(x < 0)
            printf("%d\n", x * x - 2);
        else
            if(x == 0)
                printf("3\n");
            else
                printf("%d\n", x + 3);
    }

    /* Valoarea 0 semnaleaza faptul ca programul s-a incheiat cu succes. */
    return 0;
}
```

Link execuția programului

Observație

În filmulet avem for(i=0;i<3;i+), pentru a putea pune în evidență depanarea.

Problema 2: Actualizarea unei valori naturale cu un procent dat

```
/** Includem stdio.h pentru a putea folosi functiile de citire/scriere. */
#include <stdio.h>

int main()
{
    /* Declaram doua variabile de tip double. */
    int v, p;

    /* Citim valorile celor doua variabile utilizand functia scanf.
     * Specificatorul pentru tipul double este "%lf". Atentie la string-ul de
     * formatare si la caracterul "&"! */
    scanf("%d%d", &v, &p);

    /* Actualizam valoarea variabilei v. */
    v = v + v * p;

    /* Afisam noua valoare a variabilei v, utilizand printf. */
    printf("%d\n", v);

    /* Valoarea 0 semnaleaza faptul ca programul s-a incheiat cu succes. */
    return 0;
}
```

Link execuția programlui

Problema 3: Citirea şi scrierea unei valori.

```
/** Includem stdio.h pentru a putea folosi functiile de citire/scriere. */
#include <stdio.h>

int main()
{
    /* Declaram variabila ce trebuie citita si afisata. */
    int a;
```

```
    /* Citim valoarea variabilei a utilizand scanf. Atentie la string-ul de
     * formatare si la caracterul '&'! */
    scanf("%d", &a);

    /* Afisam valoarea variabilei a utilizand printf. */
    printf("%d\n", a);

    /* Valoarea 0 semnaleaza faptul ca programul s-a incheiat cu succes. */
    return 0;
}
```

Link execuţia programului

Problema 4: Rezolvarea ecuației de grad 1: ax+b=0

```
/** Includem stdio.h pentru a putea folosi functiile de citire/scriere. */
#include <stdio.h>

int main()
{
    /* Declaram doua variabile de tip double care reprezinta coeficientii
     * ecuatiei. */
    double a, b;

    /* Citim cei doi coeficienti, primul fiind cel dominant, utilizand scanf.
     * Atentie la sirul de formatare si la caracterul "&"! */
    scanf("%lf%lf", &a, &b);

    /* Rezolvam ecuatia luand in calcul toate cazurile posibile. */
    if(a == 0) {
        if(b == 0)
            printf("Ecuatia are o infinitate de solutii\n");
        else
            printf("Ecuatia nu are nicio solutie\n");
    }
    else
        /* Solutia ecuatiei este de forma -b/a si o afisam. */
        printf("%lf\n", -b / a);

    /* Valoarea 0 semnaleaza faptul ca programul s-a incheiat cu succes. */
    return 0;
}
```

Link execuţia programului

Problema 5: Să se afişeze suma primelor n numere naturale, n citit de la tastatură.

```
/** Includem stdio.h pentru a putea folosi functiile de citire/scriere. */
#include <stdio.h>

int main()
{
    /* Declaram trei variabile de tip int. */
    int n, s, i;

    /* Citim numarul n utilizand functia scanf. Atentie la sirul de foramtare
     * si la caracterul "&"! */
    scanf("%d", &n);

    /* In scop didactic vom utiliza o instructiune de tip for pentru a calcula
     * suma primeor n numere naturale. */
    s = 0;
```

```
    for(i = 1; i <= n; i++)
        s += i;

    /* Afisam variabila s utilizand functia printf. */
    printf("%d\n", s);

    /* Valoarea 0 semnaleaza faptul ca programul s-a incheiat cu succes. */
    return 0;
}
```

Link execuţia programului

Problema 6: Algoritmul lui Euclid care determină cel mai mare divizor comun a doi întregi, prin împărțiri repetate:

```
/** Includem stdio.h pentru a putea folosi functiile de citire/scriere. */
#include <stdio.h>

int main()
{
    /* Declaram cele doua numere si o variabila ce va retine restul
     * impartirii. */
    int a, b, r;

    /* Citim cele doua numere utilizand functia scanf. Atentie la string-ul de
     * formatare si la caracterul "&"! */
    scanf("%d%d", &a, &b);

    /* Calculam restul impartirii lui a la b. Cat timp acesta este diferit de 0
     * se modifica valorile lui a si b si se recalculeaza restul. Cel mai mare
     * divizor comun va fi ultimul rest diferit de 0 care, la final, va fi
     * retinut in variabila b. */
    r = a % b;
    while(r > 0) {
        a = b;
        b = r;
        r = a % b;
    }

    /* Afisam valoarea celui mai mare divizor comun utilizand functia printf. */
    printf("%d\n", b);

    /* Valoarea 0 semnaleaza faptul ca programul s-a incheiat cu succes. */
    return 0;
}
```

Link execuţia programului

Probleme propuse şi rezolvate în limbajul C

Problema 1. Interschimbul valorilor a două variabile a şi b.

Rezolvare: Atenție! Trebuie să folosim o variabilă auxiliară. Nu funcționează a=b şi apoi b=a deoarece deja am pierdut valoarea inițială a lui a!

```
#include <stdio.h>

int main()
{
    /* Declaram cele doua variabile ale caror valori vrem sa le intreschimbam
     * si o variabila auxliara. */
```

```
    int a, b, aux;

    /* Citim valorile variabilelor a si b utilizand scanf. Atentie la string-ul
     * de formatare si la caracterul '&'. */
    scanf("%d%d", &a, &b);

    /* Interschimbam valorile variabilelor. */
    aux = a;
    a = b;
    b = aux;

    /* Afisam variabilele a si b folosind fucntia printf. */
    printf("a=%d b=%d\n", a, b);
    return 0;
}
```

Link execuția programului

Problema 2. Rezolvarea ecuației de grad 2: $ax^2+bx+c=0$.

Rezolvare: Atenție la cazurile speciale! Dacă a este 0 ecuația devine ecuație de gradul 1!

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

int main()
{
    double a, b, c, delta;

    /* Citim coeficientii incpeand cu cel dominant utilizand scanf. Atentie la
     * string-ul de formatare si la caracterul '&'! */
    scanf("%lf %lf %lf", &a, &b, &c);

    /* Rezolvam ecuatia de gradul 2 luand in calcul toate posibilitatile. */
    if(a == 0) {
        if(b == 0) {
            if(c == 0)
                printf("Ecuatia are o infinitate de solutii\n");
            else
                printf("Ecuatia nu are nicio solutie\n");
        }
        else {
            /* Avem o ecuatie de gradul 1 si ii afisam solutia. */
            printf("Ecuatia de gradul 1 are solutia: %lf\n", -b / c);
        }
    }
    else {
        delta = b * b - 4 * a * c;
        if(delta < 0)
            printf("Ecuatia nu are solutie\n");
        else {
            if(delta == 0) {
                /* Ecuatia are solutie dubla. */
                printf("Ecuatia are doua radacini egale cu: ");
                printf("%lf\n", -b / (2 * a));
            }
            else {
                /* Ecuatia are doua solutii diferite. */
                printf("Ecuatia are doua radacini distincte: ");
                printf("%lf ", (-b - sqrt(delta)) / (2 * a));
                printf("%lf\n", (-b + sqrt(delta)) / (2 * a));
            }
        }
```

```
    }

    return 0;
}
```

Link execuţia programului

Problema 3. Să se afişeze în ordine crescătoare valorile a 3 variabile a, b şi c.

Rezolvare: Putem rezolva această problemă comparând două câte două cele 3 variabile. În cazul în care nu sunt în ordine crescătoare interschimbăm valorile lor.

```
#include <stdio.h>

int main()
{
    /* Declaram 3 varaibile ce vor retine cele 3 numere si o varaiabila
     * auxiliara. */
    int a, b, c, aux;

    /* Citim cele 3 numere utilizand functia scanf. Atentie la string-ul de
     * formatare si la caracterul '&'! */
     scanf("%d%d%d", &a, &b, &c);

     if(a > b) {
        aux = a;
        a = b;
        b = aux;
     }
     if(a > c) {
        aux = a;
        a = c;
        c = aux;
     }
     if(b > c) {
        aux = b;
        b = c;
        c = aux;
     }

     /* Afisam cele 3 numere folosinf printf. */
     printf("%d %d %d\n", a, b, c);

    return 0;
}
```

Link execuţia programului

O altă variantă este următoarea, în care valorile variabilelor nu se modifică ci doar se afişează aceste valori în ordine crescătoare:

```
#include <stdio.h>

int main()
{
   /* Declaram 3 varaibile ce vor retine cele 3 numere. */
    int a, b, c;

    /* Citim cele 3 numere utilizand functia scanf. Atentie la string-ul de
     * formatare si la caracterul '&'! */
     scanf("%d %d %d", &a, &b, &c);

     if(a < b && b < c) {
```

```
        printf("%d %d %d\n", a, b, c);
        /* Semnalam ca programul se incheie cu succes. */
        return 0;
     }
    if(a < c && c < b) {
        printf("%d %d %d\n", a, c, b);
        /* Semnalam ca programul se incheie cu succes. */
        return 0;
     }
    if(b < a && a < c) {
        printf("%d %d %d\n", b, a, c);
        /* Semnalam ca programul se incheie cu succes. */
        return 0;
     }
     if(b < c && c < a) {
        printf("%d %d %d\n", b, c, a);
        /* Semnalam ca programul se incheie cu succes. */
        return 0;
     }
     if(c < a && a < b) {
        printf("%d %d %d\n", c, a, b);
        /* Semnalam ca programul se incheie cu succes. */
        return 0;
     }
     if(c < b && b < a) {
        printf("%d %d %d\n", c, b, a);
        /* Semnalam ca programul se incheie cu succes. */
        return 0;
     }
    return 0;
}
```

Link execuţia programului

Problema 4. Să se calculeze şi să se afişeze suma: S=1+1*2+1*2*3+..+n!

Rezolvare: Vom folosi o variabilă auxiliară p în care vom calcula produsul parțial.

```
#include <stdio.h>

int main()
{
    int n, i, j;

    /* Deoarece valoarea lui n! poate depasi tipul int, declaram varaibila s
     * de tip long. Mai avem nevoie de o variabila in care sa calculam i!, cu
     * i =1:n. */
    long fact, s;

    /* Citim valoarea lui n utilizand scanf. */
    scanf("%d", &n);

    /* Calculam suma de factoriale. */
    s = 0;
    for(i = 1; i <= n; i++) {
        /* Calculam i!. */
        fact = 1;
        for(j = 1; j <= i; j++)
            fact *= j;
        s += fact;
    }

    /* Afisam valoarea variabilei s utilizand printf. Specificatorul de tip
```

```
     * pentru tipul long este "%ld". */
    printf("%ld\n", s);

    return 0;
}
```

Link execuția programului

O altă modalitate este următoarea, în care produsul parțial este actualizat la fiecare pas, fără a mai fi calculat de fiecare dată de la 1:

```
#include <stdio.h>

int main()
{
    int n, i;

    /* Deoarece valoarea lui n! poate depasi tipul int, declaram varaibila s
     * de tip long. Mai avem nevoie de o variabila in care sa calculam i!, cu
     * i =1:n. */
    long fact, s;

    /* Citim valoarea lui n utilizand scanf. */
    scanf("%d", &n);

    /* Calculam suma de factoriale. */
    s = 0; fact = 1;
    for(i = 1; i <= n; i++) {
        fact *= i;
        s += fact;
    }

    /* Afisam valoarea variabilei s utilizand printf. Specificatorul de tip
     * pentru tipul long este "%ld". */
    printf("%ld\n", s);

    return 0;
}
```

Link execuția programului

Problema 5. Să se calculeze şi să se afişeze suma cifrelor unui număr natural n.

Rezolvare: Vom folosi o variabilă auxiliară ***c*** în care vom calcula rând pe rând cifrele. Pentru aceasta vom lua ultima cifră a lui ca rest al împărțirii lui n la 10, după care micşorăm n împărțindu-l la 10 pentru a ne pregăti pentru a calcula următoarea cifră, şamd.

```
#include <stdio.h>

int main()
{
    int n, s, c;

    /* Citim valoarea variabilei n utilizand scanf. */
    scanf("%d", &n);

    /* Calculam suma cifrelor numarului n stiind ca putem obtine ultima cifra
     * a acestuia ca restul impartirii lui n la 10, iar numarul n fara ultima
     * cifra este egal cu, catul impartirii lui n la 10. */
    s = 0;
    while(n > 0) {
```

```
        c = n % 10;
        s += c;
        n /= 10;
    }

    /* Afisam variabila s utilizand printf. */
    printf("%d\n", s);

    return 0;
}
```

Link execuția programului

Problema 6. Să se calculeze şi să se afişeze inversul unui număr natural n.

Rezolvare: Vom folosi o variabilă auxiliară ***c*** în care vom calcula rând pe rând cifrele ca în problema anterioară, şi vom construi inversul pe măsură ce calculăm aceste cifre.

```
#include <stdio.h>

int main()
{
    int n, inv, c;

    /* Citim valoarea variabilei n utilizand scanf. */
    scanf("%d", &n);

    /* Calculam inversul numarului n stiind ca putem obtine ultima cifra
     * a acestuia ca restul impartirii lui n la 10, iar numarul n fara ultima
     * cifra este egal cu, catul impartirii lui n la 10. */
    inv = 0;
    while(n > 0) {
        c = n % 10;
        inv = inv * 10 + c;
        n /= 10;
    }

    /* Afisam variabila inv utilizand printf. */
    printf("%d\n", inv);

    return 0;
}
```

Link execuția programului

Problema 7. Să se afişeze dacă un număr natural dat ***n*** este prim.

Rezolvare: Pentru aceasta vom împărți numărul pe rând la numerele de la 2 la radical din n (este suficient pentru a testa condiția de prim, după această valoare numerele se vor repeta). Dacă găsim un număr care să-l împartă exact vom seta o variabilă auxiliară b (numită variabila flag, sau indicator) pe 0. Ea are rolul de a indica că s-a găsit un număr care divide exact pe n. Inițial presupunem că nu există un astfel de număr, şi deci, b va avea valoarea 1 inițial.

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

int main()
{
    int n, i, prim;
    /* Citim valoarea variabilei n utilizand scanf. */
```

```
    scanf("%d", &n);

    /* Pentru a verifica daca n este prim vom cauta un numar din intervalul
     * [0, sqrt(n)] un divizor al sau. Daca nu exista niciun astfel de numar
     * atunci numarul este prim. Pentru aceasta vom folosi o varaibila numita
     * "prim" care are valoarea 1 in caz ca numarul este prim sau 0 in caz
     * contrar. Initial presupunem ca n este prim. */
    prim = 1;
    for(i = 2; i <= sqrt(n); i++)
        if((n % i) == 0) {
            /* S-a gasit un divizor si cautarea se incheie. */
            prim = 0;
            break;
        }
    if(prim)
        printf("Numarul %d este prim", n);
    else
        printf("Numarul %d nu este prim", n);

    return 0;
}
```

Link execuţia programului

O altă variantă este cea în care nu mai folosim variabila prim:

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

int main()
{
    int n, i;
    /* Citim valoarea variabilei n utilizand scanf. */
    scanf("%d", &n);

    /* Pentru a verifica daca n este prim vom cauta un numar din intervalul
     * [0, sqrt(n)] un divizor al sau. Daca nu exista niciun astfel de numar
     * atunci numarul este prim. Pentru aceasta vom folosi o varaibila numita
     * "prim" care are valoarea 1 in caz ca numarul este prim sau 0 in caz
     * contrar. Initial presupunem ca n este prim. */
    for(i = 2; i <= sqrt(n); i++)
        if((n % i) == 0) {
            /* S-a gasit un divizor si cautarea se incheie. */
            break;
        }
    if(i > sqrt(n)) //s-a iesit normal din for
        printf("Numarul %d este prim", n);
    else //s-a iesit fortat din for, prin break
        printf("Numarul %d nu este prim", n);

    return 0;
}
```

Link execuţia programului

Problema 8. Să se afişeze primele n numere naturale prime.

Rezolvare: Folosim algoritmul de la problema anterioară la care adăugăm o variabilă de contorizare – numărare, k.

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

```
int main()
{
    int n, i, prim, nr, j;

    /* Citim valoarea variabilei n utilizand scanf. */
    scanf("%d", &n);

    nr = 0;
    i = 2;
    while(nr < n) {
        prim = 1;
        for(j = 2; j <= sqrt(i); j++)
            if((i % j) == 0) {
                /* S-a gasit un divizor si cautarea se incheie. */
                prim = 0;
                break;
            }
        if(prim) {
            printf("%d\n", i);
            nr++;
        }
        i++;
    }

    return 0;
}
```

Link execuția programului

Problema 9. Să se descompună în factori primi un număr dat n.

Rezolvare: Pentru aceasta vom împărți numărul pe rând la numerele începând cu 2. Dacă găsim un număr care să-l împartă exact vom împărți pe n de câte ori se poate la numărul găsit, calculând astfel puterea. În modul acesta nu va mai fi necesar să testăm că numerele la care împărțim sunt prime!

```
#include <stdio.h>

int main()
{
    int n, div, nr;

    scanf("%d", &n);

    /* vom împărți numărul pe rând la numerele începând cu 2. Dacă găsim un
     * număr care să-l împartă exact vom împărți pe n de câte ori se poate la
     * numărul găsit, calculând astfel puterea. */
    div = 2;
    printf("n = ");
    while(n != 1) {
            nr = 0;
            while((n % div) == 0) {
                nr++;
                n /= div;
            }
            if(nr != 0)
                printf("%d^%d * ", div, nr);
            div++;
    }

    return 0;
```

```
}
```

Link execuţia programului

Problema 10. Să se afişeze toate numerele naturale mai mici decât 10000 care se pot descompune în două moduri diferite ca sumă de două cuburi.

Rezolvare: Această problemă prezintă foarte bine avantajul utilizării unui calculator în rezolvarea anumitor probleme. Calculăm, pe rând, suma de cuburi a perechilor de numere de la 1 la 21 (cel mai apropiat întreg de radical de ordin 3 din 10000). Căutăm apoi, pentru fiecare sumă, o a doua pereche a cărei sumă de cuburi este aceeaşi. Dacă este o pereche diferită de prima, afişăm numerele.

```
#include <stdio.h>

int main()
{
    int a, b, c, d, x, y;

    /* Numerele pe care le cautam cor fi in intervalul [1, 21], deoarece
     * 22^3 > 10000. */

     for(a = 1; a < 22; a++)
        for(b = 1; b < 22; b++) {
            x = a * a * a + b * b * b;
            /* Cautam inca o pereche de numere cu aceeasi proprietate. */
            for(c = 1; c < 22; c++)
                for(d = 1; d < 22; d++) {
                    y = c * c * c + d * d * d;
                    if(x == y && c != a && d != b)
                        printf("%d=%d^3+%d^3=%d^3+%d^3\n", x, a, b, c, d);
                }
        }
    return 0;
}
```

Link execuţia programului

Problema 11. Să se calculeze valoarea minimă, respectiv maximă, dintr-o secvență de n numere reale.

Rezolvare: Vom utiliza două variabile, max şi min pe care le inițializăm cu o valoare foarte mică şi respectiv cu o valoare foarte mare. Vom compara pe rând valorile citite cu max şi respectiv cu min, iar dacă găsim o valoare mai mare, respectiv mai mică decât acestea modificăm max (sau min, după cum e cazul) la noua valoare maximă, respectiv minimă.

```
#include <stdio.h>

int main()
{
    int min, max, i, n, x;

    scanf("%d", &n);

    min = 32000;
    max = -32000;
    for(i = 0; i < n; i++) {
        scanf("%d", &x);
        if(x > max)
            max = x;
        if(x < min)
```

```
            min = x;
    }

    printf("Valoarea maxima este: %d\n", max);
    printf("Valoarea minima este: %d\n", min);

    return 0;
}
```

Link execuţia programului

Capitolul IB.05. Tablouri. Definire şi utilizare în limbajul C

Cuvinte-cheie
Tablou, tablouri unidimensionale, vector, indexare, tablouri multidimensionale, tablouri bidimensionale, matrice

IB.05.1 Tablouri

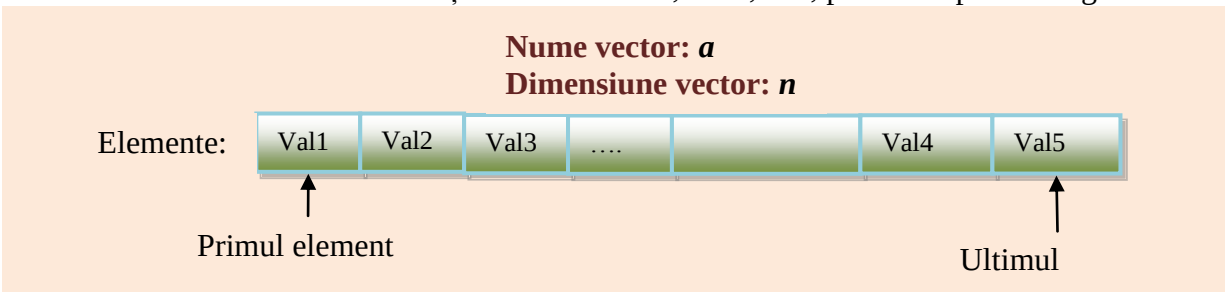
Să presupunem că avem următoarea problemă: *Să se afişeze numele tuturor studenţilor care au nota maximă; numele şi notele se citesc de la tastatură.*

Până acum, problemele rezolvate de genul acesta, presupuneau citirea unor date şi prelucrarea lor pe măsură ce sunt citite, fără a reține toate valorile citite (vezi problema 1 (capitolul IB.01) cu funcția F(x) – nu se reţineau toate valorile lui x, ci doar una la un moment dat!) . Acest lucru însă nu este posibil aici, deoarece trebuie ca mai întâi să aflăm nota maximă printr-o primă parcurgere a datelor de intrare şi apoi să mai parcurgem încă o dată aceste date pentru a afişa studenţii cu nota maximă. Pentru aceasta este necesară memorarea tuturor studenților (a datelor de intrare, nume-nota). Cum memorăm însă aceste valori? Cu siguranță nu vom folosi n variabile pentru nume: nume1, nume2, etc. şi n variabile pentru nota: nota1, nota2, etc., mai ales că nici nu ştim exact cât este n - câți studenți vor fi!
Vom folosi în loc o singură variabilă de tip tablou, cu mai multe elemente pentru nume şi o singură variabilă de tip tablou cu mai multe elemente, pentru note.

Prin ***tablou*** se înțelege în programare o colecție finită, liniară, de date de acelaşi tip – numit tip de bază al tabloului – colecție care ocupă un spațiu continuu de memorie. În limba engleză se foloseşte cuvântul *array*.
În funcție de numărul de dimensiuni putem avea mai multe tipuri de tablouri, cele mai utilizate fiind cele unidimensionale, numite, de obicei, vectori, şi cele bidimensionale cunoscute sub numele de matrice.

IB.05.2 Tablouri unidimensionale: vectori

Un tablou unidimensional care conține valorile *Val1*, *Val2*, etc., poate fi reprezentat grafic astfel:



Elementele sale sunt memorate unele după altele – ocupă un spațiu continuu de memorie.
Modul de declarare al unui **vector** cu dimensiune constantă este următorul:

Sintaxa:
tip_de_bază nume_tablou [dimensiune] ***= { const0, const1, ...}***

Unde:

- *tip_de_bază* este tipul elementelor;
- *dimensiune* (lungimea vectorului) reprezintă numărul maxim de elemente ale tabloului şi este în general o constantă întreagă (de obicei o constantă simbolică);
- *const0, const1, etc.* reprezintă valori constante de inițializare, şi prezența lor este opțională.

Memoria ocupată de un vector este egală cu *dimensiune* * sizeof(*tip_de_bază*).
Exemple:

```
int tab[10];      //defineşte un tablou de 10 elemente întregi

float v[60];      //defineşte un tablou de 60 elemente reale

#define N 10
int tab[N];       // definitie echivalenta cu prima
```

Pentru a crea un vector, trebuie să cunoaştem dimensiunea sa (lungimea vectorului) în avans, deoarece odată creat, dimensiunea sa este fixă: este alocată la compilare şi nu mai poate fi modificată la execuţie!
Uneori nu putem cunoaşte în avans dimensiunea vectorului, aceasta fiind o dată de intrare a programului (de exemplu, câţi studenţi vom avea?). În astfel de cazuri vom estima o dimensiune maximă pentru vector, dimensiune care este o limită a acestuia şi vom declara vectorul ca având această dimensiune maximă acoperitoare (însă cât mai apropiată de cea reală). Acesta este probabil principalul dezavantaj al utilizării unui vector. De obicei se foloşte o constantă simbolică pentru a desemna dimensiunea maximă a unui vector şi se verifică dac[valoarea citită este cel mult egală cu valoarea constantei. De asemenea, vom mai avea o variabilă în care vom păstra numărul efectiv de elemente din vector – *numele variabilei este de obicei n!*
Exemplu:

```
#define M 100                       // dimensiune maxima vector
int main () {
      int a[M], n;
      scanf ("%d", &n); // citeste dimensiune efectiva
      if ( n > M) {
            printf ("Eroare: n > %d \n",M); return;
      }
      ...   // citire şi utilizare elemente vector
```

În acest fel, codul programului este independent de dimensiunea efectivă a vectorului, care poate fi diferită de la o execuţie la alta.
Este permisă iniţializarea unui vector la declarare, prin precizarea constantelor de iniţializare între acolade şi separate prin virgule. Dacă numărul acestora:

- este egal cu dimensiune - elementele tabloului se iniţializează cu constantele precizate
- < dimensiune - constantele neprecizate (lipsă) sunt considerate implicit 0
- > dimensiune - apare eroare la compilare.

Dacă este prezentă partea de iniţializare, dar lipseşte dimensiune, aceasta este implicit egală cu numărul constantelor din partea de iniţializare şi este **singurul caz în care este posibilă omiterea dimensiunii!**
Exemple:

```
// Declararea si initializarea unui vector de 3 numere intregi:
int azi[3] = { 01, 04, 2001 };      // zi,luna,an

/* Vectorul a va avea dimensiune 3,  aceasta fiind data de numarul constantelor
de initializare: */
double a[]={2,5.9};

//numarul constantelor < numarul elementelor:
#define NR_ELEM 5
float t[NR_ELEM]={1.2,5,3};
int prime[1000] = {1, 2, 3};
/*primele trei elemente se initializeaza cu constantele precizate, urmatoarele
cu 0. Poate fi confuz. Nu se recomandă! */

int a[1000] = { 0 };              //toate elementele sunt iniţializate cu zero
int numbers[2] = {11, 33, 44};  // EROARE: prea multe iniţializări
```

Putem afla dimensiunea cu care a fost declarat un vector folosind expresia:

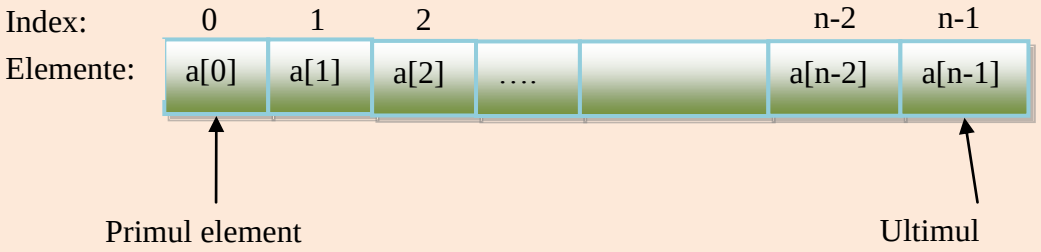
`sizeof(nume_tablou) / sizeof(tip_de_bază)`

- `sizeof(nume_tablou)` returnează numărul total de octeți ocupați de vector.

Fiecare element din vector este identificat printr-un indice întreg, pozitiv, care arată poziția sa în vector; pentru selectarea unui element se foloseşte operatorul de **indexare**: [] – paranteze drepte. Se spune că accesul este direct la orice element din vector.

Selectarea unui element dintr-un vector:
nume_tablou [indice]

unde *indice* este o expresie întreagă cu valori între 0 şi dimensiune -1.
Un element de tablou poate fi prelucrat ca orice variabilă având tipul de bază.
Numerotarea elementelor fiind de la zero, primul element din orice vector are indicele zero, iar ultimul element dintr-un vector are un indice mai mic cu 1 decât numărul elementelor din vector.



Exemple:

```
//dacă avem un vector de 5 note:
int note[5];
//atribuim valori elementelor astfel:
note[0] = 95;
note [1] = 85;
note [2] = 77;
note [3] = 69;
note [4] = 66;
//afisarea valorii unor elemente:
printf("prima nota este %d\n", note[0]);
printf("suma ultimelor doua note este  %d\n", note[3]+note[4]);
```

Utilizarea unui vector presupune, în general, repetarea unor operații asupra fiecărui element din vector deci folosirea unor structuri repetitive. De obicei, pentru a realiza o prelucrare asupra tuturor elementelor unui tablou se foloseşte instrucțiunea *for* cu o variabilă *contor* care să ia toate valorile indicilor (între 0 şi dimensiune -1).

Exemplu:

```
#define N 10
int tab[N], i;
for(i=0; i<N; i++)
  …… //prelucrare tab[i]
```

După cum spuneam, de obicei nu toate elementele tabloului (alocate in memorie) sunt folosite, ci doar primele n elem<=dimensiune (vezi programul următor, se pot citi doar primele n elem):

```
int a[100], n, i;
// vectorul a de max 100 de intregi, n numărul efectiv de elemente folosite

//citirea şi afişarea unui vector de întregi:
scanf ("%d",&n);              // citeşte nr efectiv de elemente din vector
for (i=0;i<n;i++)
      scanf ("%d", &a[i]);    // citire elemente vector
for (i=0;i<n;i++)
      printf ("a[%d]=%d\n", i, a[i]);     // scrie elemente vector

//suma elementelor 0..n-1 din vectorul a
int s;
for (i=0, s=0; i<n; i++)
      s = s + a[i];
```

Observaţii:

- Nici compilatorul, nici mediul de execuție nu verifică valorile indicilor. Cu alte cuvinte, nu sunt generate în mod normal avertizări/erori (warning/error) dacă indexul este în afara limitelor; programatorul trebuie să codifice astfel încât indicele să ia valori în intervalul *0 .. dimensiune-1*, deoarece pot apare erori imprevizibile, ca de exemplu modificarea nedorită a altor variabile.

Exemple:

```
#define                                N                                10
int tab[N];
tab[N]=5;    // se modifica zona de 2 octeti urmatoare tabloului
tab[-10]=6; /* se modifica o zona de 2 octeti situata la o adresa cu 20
de octeti inferioara tabloului */

// Program ce poate compila şi chiar rula dar cu posibile erori
colaterale:
const int dim = 5;
int numere[dim];        // vector cu index de la 0 la 4
numere[88] = 999;
printf("%d\n", numere[77]);
// Index in afara limitelor, nesemnalat!
```

Aceasta este un alt dezavantaj C/C++. Verificarea limitelor indexului ia timp şi putere de calcul micşorând performanțele. Totuşi, e mai bine să fie sigur decât rapid!

- Numele unui vector nu poate să apară în partea stângă a unei atribuiri deoarece are asociată o adresă constantă (de către compilator), care nu poate fi modificată în cursul executiei.

Exemplu de greşeală:

```
int a[30]={0}, b[30];
b=a;  // eroare !
```

- Pentru copierea datelor dintr-un vector într-un alt vector se va scrie un ciclu pentru copierea element cu element, sau se va folosi funcția *memcpy*.

Exemplu:

```
int a[30]={1,3,5,7,9}, b[30], i, n;
....
for (i=0;i<n;i++)
      b[i]=a[i];
```

- Dimensiunea unui vector poate fi şi valoarea unei variabile, dar **atenție**!, declararea vectorului se va face după inițializarea variabilei, nu înainte, deoarece tentativa de a citi valoarea variabilei după declararea vectorului va genera eroare!

Exemple:

```
int size;
printf("Introduceti dimensiunea vectorului:");
scanf("%d", &size);
float values[size];

// NU!
int size;
float values[size];
printf("Introduceti dimensiunea vectorului:");
scanf("%d", &size);
```

Totuşi, acesta nu este un mod recomandat de declarare a vectorilor datorită complicațiilor care pot apare.

IB.05.3 Tablouri multidimensionale

Modul de declarare al unui tablou multidimensional este următorul:

Sintaxa:
tip_de_baza nume_tablou [dim1] [dim2] ... [dimn] = { {const10,...}, {const20,...}, ..., {constn0,...} };

unde:

- *tip_de_bază* este tipul elementelor;
- *dim1, dim2, ..., dimn* - expresii întregi constante (de obicei constante simbolice).
- *const10, const20, etc.* reprezintă valori constante de inițializare, şi prezența lor este opțională.

dim1, dim2, ..., dimn reprezintă numărul maxim de elemente pentru prima dimensiune, a 2-a dimensiune, etc.

Observație: în practică se folosesc rar tablouri cu mai mult de 2 dimensiuni.

Memoria continuă ocupată de tablou este de dimensiune:

*dim1 * dim2 * ... * dimn * sizeof(tip_de_baza).*

Elementele tabloului multidimensional sunt memorate astfel încât ultimul indice variază cel mai rapid. Tabloul poate fi inițializat la definire - vezi partea opționala marcată - prin precizarea constantelor de inițializare.

Selectarea unui element dintr-un tablou multidimensional:
nume_tablou [ind1] [ind2] … [indn]

Unde *ind1, ind2, ..., indn* - expresii întregi cu valori între *0* şi *dimi-1* pentru *i=1..n*. Un element de tablou poate fi prelucrat ca orice variabila având tipul de bază.

IB.05.4 Tablouri bidimensionale: matrici

Un tablou bidimensional (caz particular de tablou multidimensional), numit şi *matrice*, are două dimensiuni, orizontală şi verticală, şi este definit prin dimensiune maximă pe orizontală - număr maxim de linii şi dimensiune maximă pe verticală - număr maxim de coloane.

În limbajul C matricele sunt liniarizate pe linii, deci în memorie linia 0 este urmată de linia 1, linia 1 este urmată de linia 2 ş.a.m.d., cu alte cuvinte, elementele unei matrici sunt memorate pe linii,

unele după altele – ocupă un spațiu continuu de memorie. O matrice bidimensională este privită în C ca un vector de vectori (de linii).

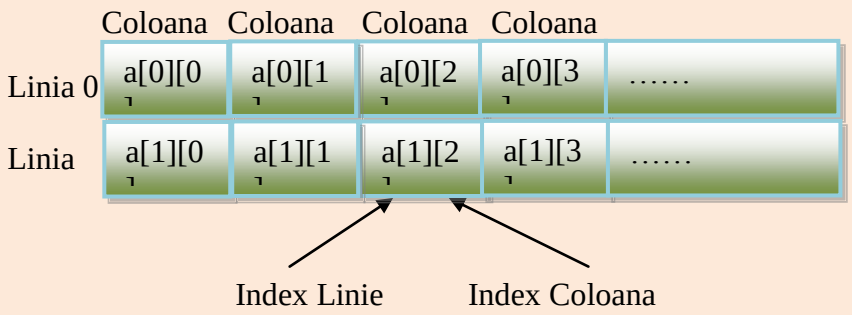
Modul de declarare al unei matrici este următorul:

Sintaxa:
tip_de_bază nume_tablou [dim1] [dim2] ***= {{const10,...},{const20,...},...,{constn0,...}};***

unde:

- *tip_de_bază* este tipul elementelor;
- *dim1* reprezintă numărul maxim de linii (prima dimensiune) iar *dim2* numărul maxim de coloane (a -2-a dimensiune) şi sunt constante întregi (de obicei constante simbolice);
- *const10, const20, etc.* reprezintă valori constante de inițializare, şi prezența lor este opțională.

Memoria continuă ocupată de tablou este *dim1 * dim2 * sizeof(tip_de_bază).*
O matrice poate fi reprezentată grafic astfel:



Exemple:

```
int m[10][5];     /* defineste un tablou dimensional de elemente intregi,
cu maxim 10 linii si 5 coloane*/

// definitie echivalenta cu cea de mai sus:
#define NL 10
#define NC 5
int m[NL][NC];
```

Rămân valabile toate observațiile făcute la vectori, legate de dimensiunile matricii. De obicei se folosesc constante simbolice pentru aceste dimensiuni şi se verifică încadrarea datelor citite în dimensiunile declarate. De cele mai multe ori, vom folosi două variabile în care vom păstra numărul efectiv (cel cu care se lucrează la execuția curentă) de linii, respectiv de coloane din matrice!
Este posibilă inițializarea unei matrice la definirea ei, iar elementele care nu sunt inițializate explicit primesc valoarea zero. Avem aceleaşi observații ca la vectori.
Exemple:

```
int b[2][3] = {1,2,3,4,5,6};          // echivalent cu:
int b[2][3] = {{ 1,2,3},{ 4,5,6}};   // echivalent cu:
int b[][ 3] = {{ 1,2,3},{ 4,5,6}}
double a[3][2]={{2},{5.9,1},{-9}};
//elementele pe linii sunt: 2  0 / 5.9  1 / -9  0

double a[3][2]={2,5.9,1,-9};
//elementele pe linii sunt:   2  5.9  / 1   -9  / 0    0
```

Selectarea unui element dintr-o matrice:
nume_tablou [ind1] [ind2]

De exemplu, notația *a[i][j]* desemnează elementul din linia *i* şi coloana *j* a unei matrice *a*, sau altfel spus elementul din poziția *j* din vectorul *a[i]*.

Prelucrarea elementelor unei matrice se face prin două cicluri; un ciclu pentru parcurgerea fiecărei linii şi un ciclu pentru parcurgerea fiecărei coloane dintr-o linie:
Exemplu:

```
// afişare matrice cu nl linii şi nc coloane pe linii
for (i=0;i<nl;i++) {
for (j=0;j<nc;j++)
      printf (“%6d”, a[i][j]);
printf(“\n”);
}
```

Observații:

- Exemplul de mai sus corespunde unei prelucrări *pe linii* a elementelor matricei; în cazul unei prelucrări *pe coloane*, ciclul exterior este cel cu contorul corespunzător indicelui de coloană.
- Numărul de cicluri incluse poate fi mai mare dacă la fiecare element de matrice se fac prelucrări repetate. De exemplu, la înmulțirea a două matrice *a* şi *b*, fiecare element al matricei rezultat *c* se obține ca o sumă:

Exemplu:

```
for (i=0;i<n;i++)
      for (j=0;j<m;j++) {
                  c[i][j]=0;  //initializare element matrice produs
                  for (k=0;k<p;k++)
                              c[i][j] += a[i][k]*b[k][j];
//calcul suma de produse
      }
```

- Numerotarea liniilor şi coloanelor de la 0 în C este diferită de numerotarea uzuală din matematică, care începe de la 1. Pentru numerotarea de la 1 putem să nu folosim linia zero şi coloana zero (ceea ce nu se recomandă) sau să ajustăm indicii matematici scăzând 1.

Exemplu de citire şi afişare matrice cu numerotare linii si coloane de la 1, în care linia zero şi coloana zero nu se folosesc:

```
int nl, nc, i, j;
float a[50][50];

//citire numar de linii, numar de coloane
printf("nr.linii: ");
scanf("%d",&nl);
printf("nr.coloane: ");
scanf("%d",&nc);

//citire matrice pe linii
for (i=1;i<=nl;i++)
        for (j=1;j<=nc;j++)
          scanf ("%f", &a[i][j]);

//afisare matrice pe linii
for (i=1;i<=nl;i++) {
        for (j=1;j<=nc;j++)
                  printf ("%f  ",a[i][j]);
        printf ("\n");
      }
```

IB.05.5 Probleme propuse

Programele din capitolul IB.01 rezolvate în C folosind tablouri

Problema 12. Se dă o secvență de n numere întregi pozitive. Să se afişeze cele mai mari numere de 2 cifre care nu se află în secvența respectivă.
Rezolvare: În acest caz vom folosi un vector ca variabilă auxiliară în care vom ține minte dacă un număr de două cifre a fost citit de la tastatură (***v[nr]*** este 0 dacă ***nr*** nu a fost citit şi ***v[nr]*** devine 1 dacă ***nr*** a fost citit de la tastatură). Inițial, toate valorile din vector sunt 0.
Pentru a afla cele mai mari numere de două cifre care nu sunt în secvența citită vom parcurge vectorul v de la coadă (99) la cap până întâlnim două valori zero.
Atenție! În cazul în care nu există una sau două valori de două cifre care să nu aparțină secvenței citite nu se va afişa nimic!

```
#include <stdio.h>
#define N 103

int main() {
    int n, v[N], i, nr;

    scanf("%d", &n);

    for(i = 10; i < 100; i++)
     v[i] = 0;

    for(i = 0; i < n; i++) {
     scanf("%d", &nr);
     if(nr > 9 && nr < 100)
      v[nr] = 1;
    }

     i = 99;
     while(v[i] != 0 && i > 0)
        i--;
     if(i > 9)
        printf("%d ", i);

     i--;
     while(v[i] != 0 && i > 0)
        i--;
     if(i > 9)
        printf("%d ", i);
     return 0;
}
```

Problema 13. Se dă o secvență de n numere întregi, ale căror valori sunt cuprinse în intervalul 0-100. Să se afişeze valorile care apar cel mai des.
Rezolvare: Vom utiliza de asemenea un vector în care vom ține minte de câte ori a apărut fiecare valoare de la 0 la 100 – ***v[nr]*** reprezintă de câte ori a fost citit ***nr***. Inițial toate valorile din vector sunt 0. Vom determina apoi valoarea maximă din acest vector, după care, pentru a afişa toate numerele care apar de cele mai multe ori mai parcurgem încă o dată vectorul şi afişăm indicii pentru care găsim valoarea maximă.

```
#include <stdio.h>
#define MaxN 103

int main() {
      int i, n, v[MaxN], nr, max;
  scanf("%d", &n);
```

```
   for(i = 0; i < 100; i++)
      v[i] = 0;

   for(i = 0; i < n; i++) {
      scanf("%d", &nr);
      v[nr]++;
   }
   max=0;
   for(i = 0; i < 100; i++)
      if(v[i] > max)
       max = v[i];
   for(i = 0; i < 100; i++)
      if(v[i] == max)
        printf("%d\n", i);
   return 0;
}
```

Alte exemple propuse

1. Se consideră un vector de N elemente întregi (N este constantă predefinită). Să se prelucreze tabloul astfel:

- să se citească cele N elemente de la tastatură (varianta: până la CTRL/Z - se decomenteaza linia comentată din partea de citire şi se comentează cea anterioară)
- să se afişeze elementele
- să se afişeze maximul şi media aritmetică pentru elementele vectorului
- să se caute în vector o valoare citită de la tastatură
- să se construiască un vector copie al celui dat
- să se afişeze elementele tabloului copie în ordinea inversă.

```
#include <stdio.h>
#define N 8

int main(){
    int tablou[N], copie[N], nr_elem;
    /* tablou va contine nr_elem de prelucrat */
    int i,max,suma, de_cautat;
    float meda;

    //  citire
    puts("Introduceti elementele tabloului:");
    for(i=0;i<N;i++){
       printf("elem[%d]=",i); //se afiseaza indicele elem ce se citeste
       scanf("%d",&tablou[i]);
        //if(scanf("%d",&tablou[i])==EOF)break;
    }
    nr_elem=i;

    //  tiparire
    puts("Elementele tabloului:");
    for(i=0;i<nr_elem;i++) printf("elem[%d]=%d\n",i,tablou[i]);

    //  info
    for(i=suma=0,max=tablou[0];i<nr_elem;i++){
         suma+=tablou[i];
         if(tablou[i]>max) max=tablou[i];
    }
    printf("Val maxima=%d, media aritm=%f\n", max, (float)suma/nr_elem);

    //  cautare
    printf("Se cauta:"); scanf("%d",&de_cautat);

    for(i=0;i<nr_elem;i++)
```

```
            if(de_cautat==tablou[i])break;
    //cele doua linii de mai sus se pot scrie echivalent:
    // for(i=0;i<nr_elem && de_cautat!=tablou[i];i++)

    if(i<nr_elem) printf("S-a gasit valoarea la indicele %d!\n",i);
    else puts("Nu s-a gasit valoarea cautata!");

    //  copiere
    for(i=0;i<nr_elem;i++) copie[i]=tablou[i];

    //  tiparire inversa
    puts("Elementele tabloului copie in ordine inversa:");
    for(i=nr_elem-1;i>=0;i--) printf("copie[%d]=%d\n",i,copie[i]);
    return 0;
}
```

2. Să se scrie un program care citeşte coeficienţii unui polinom de x, cu gradul maxim N (N este o constantă predefinită), calculând apoi valoarea sa în puncte x citite, până la introducerea pentru x a valorii 0. Să se afişeze şi valoarea obţinută prin apelarea funcţiei de biblioteca *poly*. Să se modifice programul astfel încât să citească ciclic polinoame, pe care să le evalueze.

```
#include <stdio.h>
#define N 10

int main(){
      double coeficient[N+1], x, val;
      //numarul de coeficienti este cu 1 mai mare decat gradul
      int grad, i; //grad variabil <=N

      // citire grad cu validare
      do {
            printf("grad maxim(0..%d)=",N);scanf("%d",&grad);
      } while ( grad<0 || grad>N );

      // citire coeficienti polinom
      puts("Introduceti coeficientii:");
      for(i=0;i<=grad;i++){
            printf("coef[%d]=",i);
            scanf("%lf",&coeficient[i]);
           // se afiseaza indicele elem ce se citeste
      }

      //afisare polinom
      printf("P(x)=");
      for(i=grad;i>0;i--)
            if(coeficient[i])
                  printf("%lf*x^%d+",coeficient[i],i);
      if(coeficient[0])
            printf("%lf\n",coeficient[0]); /* termenul liber */

      //citire repetata pana la introducerea valorii 0
      while(printf("x="),scanf("%lf",&x),x){
     /*while(printf("x="),scanf("%lf",&x)!=EOF) //daca oprire la CTRL/Z */

            //  calcul valoare polinom in x
            val=coeficient[grad];
            for(i=grad-1;i>=0;i--){
                  val*=x;
                  val+=coeficient[i];
            }

            //afisare valoare calculata
```

```
            printf("P(%lf)=%lf\n",x,val);
      }
      return 0;
}
```

3. Pentru o matrice de numere întregi cu NL linii si NC coloane (NL, NC constante simbolice, să se scrie următoarele programe:

a. citeşte elementele matricii pe coloane;
b. tipăreşte elementele matricii pe linii;
c. determină şi afişeaza valoarea şi poziția elementului maxim din matrice;
d. construieşte un tablou unidimensional, ale cărui elemente sunt sumele elementelor de pe câte o linie a matricii;
e. interschimbă elementele de pe două coloane ale matricii cu indicii citiți;
f. caută în matrice o valoare citită, afişându-i poziția;
g. calculează şi afişează matricea produs a două matrici de elemente întregi.

Capitolul IB.06. Funcții. Definire şi utilizare în limbajul C

Cuvinte cheie
Subrutină, top down, funcție apelată, funcție apelantă, parametri, argumente, definire funcții, funcții void, declarare funcții, domeniu de vizibilitate (scope), apel funcție, instrucțiunea return, transmiterea parametrilor, funcții cu argumente vectori, recursivitate

IB.06.1. Importanța funcțiilor în programare

În unele cazuri, o anumită porțiune de cod este necesară în mai multe locuri (puncte) ale programului. În loc să repetăm acel cod în mai multe locuri, este preferabil să-l reprezentăm într-o aşa numită ***subrutină*** şi să *chemăm (apelăm)* această subrutină în toate punctele programului în care este necesară execuția acelui cod. Acest lucru permite o mai bună înțelegere a programului, uşurează depanarea, testarea şi eventuala sa modificare ulterioară. În C/C++ subrutinele se numesc ***funcții***.

De ce discutam acum despre funcții? Deoarece programele prezentate în continuare vor creşte în complexitate, aşa încât, pentru dezvoltarea lor, vom aplica **tehnica de analiza şi proiectare *Top Down*** sau *Stepwise Refinement* ce cuprinde următorii paşi:

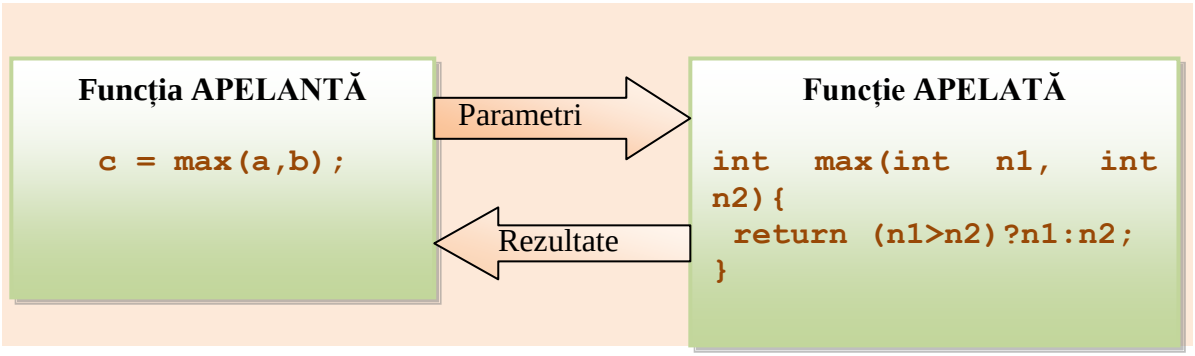
1. problema se descompune în subprobleme - în paşi de prelucrare
2. fiecare subproblemă poate fi descompusă la rândul său în alte subprobleme
3. fiecare subproblemă este implementată într-o funcție
4. aceste funcții sunt apelate în *main*, ceea ce va duce la execuția pe rând a paşilor necesari pentru rezolvarea problemei.

Funcția este un concept important în matematică şi programare. În limbajul C prelucrările sunt organizate ca o ierarhie de apeluri de funcții. Orice program trebuie să conțină cel puțin o funcție, funcția *main*.
Funcțiile încapsulează prelucrări bine precizate şi pot fi reutilizate în mai multe programe. Practic, nu există program care să nu apeleze atât funcții din bibliotecile existente cât şi funcții definite în cadrul aplicației respective. Ceea ce numim uzual *program* sau *aplicație* este de fapt o colecție de funcții (subprograme).

Motivele utilizării funcțiilor sunt multiple:

- Evită repetarea codului: este uşor să faci *copy* şi *paste*, dar este greu să menții şi să sincronizezi toate copiile;
- Utilizarea de funcții permite după cum spuneam dezvoltarea progresivă a unui program mare, fie de jos în sus (*bottom up*), fie de sus în jos (*top down*), fie combinat. Astfel, un program mare poate fi mai uşor de scris, de înțeles şi de modificat dacă este modular, adică format din module funcționale relativ mici;
- O funcție poate fi reutilizată în mai multe aplicații - prin adăugarea ei într-o bibliotecă de funcții - ceea ce reduce efortul de programare al unei noi aplicații;
- O funcție poate fi scrisă şi verificată separat de restul aplicației, ceea ce reduce timpul de punere la punct al unei aplicații mari (deoarece erorile pot apare numai la comunicarea între subprograme corecte);
- Întreținerea unei aplicații este simplificată, deoarece modificările se fac numai în anumite funcții şi nu afectează alte funcții (care nici nu mai trebuie recompilate);

Standardul limbajului C conține o serie de funcții care există în toate implementările limbajului. Declarațiile acestor funcții sunt grupate în *fișiere antet* cu același nume pentru toate implementările. În afara acestor funcții standard există și alte biblioteci de funcții: funcții specifice sistemului de operare, funcții utile pentru anumite aplicații (grafică, baze de date, aplicații de rețea ș.a.).
Două entități sunt implicate în utilizarea unei funcții: un apelant, care apelează (cheamă) funcția și **funcția apelată**. Apelantul, care este și el o funcție, transmite **parametri** (numiti și **argumente)** funcției apelate. Funcția primește acești parametri, efectuează operațiile din corpul funcției și returnează rezultatul/rezultatele înapoi **funcției apelante**.



Comunicarea de date între funcții se face de regulă prin argumente și numai în mod excepțional prin variabile externe funcțiilor – vezi domeniu de definiție.

Exemplu

Să presupunem că avem nevoie să evaluăm aria unui cerc de mai multe ori (pentru mai multe valori ale razei). Cel mai bine este să scriem o funcție numită **calculAria**(), și să o folosim când avem nevoie.

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

// prototipul funcției (declararea)
double calculAria (double);

int main() {
        double raza1 = 1.1, aria1, aria2;

        // apeleaza  functia calculAria:
        aria1 = calculAria (raza1);
        printf("Aria 1 este %lf\n ", area1);

        // apeleaza  functia calculAria:
        aria2 = calculAria (2.2);
        printf("Aria 2 este %lf\n ", area2);

        // apeleaza  functia calculAria:
        printf("Aria 3 este %lf\n ",  calculAria (3.3));
        return 0;
}
// definirea functiei
double calculAria (double raza) {
        return raza* raza*M_PI; // M_PI definit in biblioteca math

}
```

Rezultatul va fi:
```
Aria 1 este 3.80134
Aria 2 este 15.2053
Aria 3 este 34.212
```

În acest exemplu este definită o funcție numită *calculAria* care primeşte un parametru de tip double de la funcția apelantă – în acest caz *main-ul* – efectuează calculul şi returnează un rezultat, tot de tip *double* funcției care o apelează. În *main*, funcția *calculAria* este chemată de trei ori, de fiecare dată cu o altă valoare a parametrului.

IB.06.2. Definirea şi utilizarea funcțiilor

Pentru a putea fi utilizată într-un program, definiția unei funcții trebuie să preceadă utilizarea (apelarea) ei.

Forma generală a unei definiții de funcție, conform standardului, este:

Sintaxa:
```
tip_rezultat_returnat nume_functie (lista_parametri_formali) {
   /* corpul functiei: */
   definirea variabilelor locale          //declarații
   prelucrari                             // instructiuni
}
```

unde:

- *tip_rezultat_returnat* este tipul rezultatului returnat de funcție. În limbajul C o parte din funcții au o valoare ca rezultat iar altele nu au(sunt de tip *void*). Pentru o funcție cu rezultat diferit de *void* tipul funcției este tipul rezultatului funcției. Tipul unei funcții C poate fi orice tip numeric, orice tip pointer, orice tip structură (*struct*) sau *void*.
  Dacă *tip_rezultat_returnat* este:
  - int - funcția este întreagă
  - void - funcția este void
  - float - funcția este reală.
- Lista parametrilor formali cuprinde declarația parametrilor formali, separați prin virgulă:

  Sintaxa:
  ```
  lista_parametri_formali = tip_p1 nume_p1, tip_p2 nume_p2, ..., tip_pn nume_pn
  ```
  unde:
  - ***tip_p1, tip_p2,., tip_pn*** sunt tipurile parametrilor formali
  - ***nume_p1, nume_p2, ..., nume_pn*** sunt numele parametrilor formali

Exemple:

1. Să se calculeze şi să se afişeze valoarea expresiei $x^m+y^n+(xy)^{m\wedge n}$, x, y, m, n fiind citiți de la tastatură, astfel încât întregii m,n să fie pozitivi. Ridicarea la putere se va realiza printr-o funcție putere care primeşte baza şi exponentul ca parametri şi returnează rezultatul.
   Modul în care se va apela (folosi) aceasta funcție îl vom arăta la **Apelul unei funcții.**

```
// functia care calculeaza baza^exp
double putere (double baza, int exp){
   int i;                        //declarare variabila locale
   float rez;              //declarare variabila locale

   //instructiuni:
   for(i=rez=1;i<=exp;i++)
        rez*=baza;

   return rez;             // returnare valoare calculata in functia apelanta
}
```

2. Numărarea şi afişarea numerelor prime mai mici ca un întreg dat n. Pentru aceasta vom scrie o funcție care testează dacă un număr este prim. Modul în care se va apela (folosi) această funcție îl vom arăta la **Apelul unei funcții.**

```
//returneaza 0 daca numarul nu este prim, 1 daca este prim
int prim (int numar){
   int div;                      //declarare variabila locale
   //instructiuni:
   for (div=2; div<=sqrt(numar); div++)
        if (numar % div ==0) return 0;// returnare 0 daca am gasit divizor
divizor

   /*aici se ajunge doar daca nu s-a iesit pe return 0, adica daca nu am
gasit niciun divizor*/
   return 1;                  // returnare 1 daca nu am gasit niciun divizor
```

Parametrii formali sunt vizibili doar în corpul funcției care îi definește. Prin parametrii formali, funcția primește datele inițiale (de intrare) necesare şi poate transmite rezultate. Parametrii formali pot fi doar nume de variabile, adrese de variabile (pointeri) sau nume de vectori, deci nu pot fi expresii sau componente de vectori.

Observații:

- Definițiile funcțiilor nu pot fi incluse una în alta (ca în Pascal).
- Se recomandă ca o funcție să îndeplinească o singură sarcină şi să nu aibă mai mult de câteva zeci de linii sursă (preferabil sub 50 de linii).
- Este indicat ca numele unei funcții să fie cât mai sugestiv, poate chiar un verb (denotă o acțiune) sau o expresie ce conține mai multe cuvinte neseparate între ele. Primul cuvânt este scris cu literă mică, în timp ce restul cuvintelor sunt scrise cu prima literă mare.
  Exemple: calculAria(), setRaza(), mutaJos(), ePrim(), etc.

Funcții void

Să presupunem că avem nevoie de o funcție care să efectueze anumite acțiuni (de exmplu o tipărire), fără să fie nevoie să returneze o valoare apelantului. Putem declara această funcție ca fiind de tipul void.
Dacă funcția nu returnează nici un rezultat dar primeşte parametri, definiția va fi:

Sintaxa:

```
void nume_functie (lista_parametri_formali) {
   /* corpul functiei: */
   definirea variabilelor locale          //declarații
   prelucrari                             // instructiuni
}
```

Dacă funcția nu returnează nici un rezultat şi nu primeşte parametri, definiția va fi:

Sintaxa:

```
void nume_functie ( ) {
   /* corpul functiei: */
   definirea variabilelor locale          //declarații
   prelucrari                             // instructiuni
}
```

Observație: o funcție void poate întoarce rezultatele prelucrărilor efectuate prin parametri.

IB.06.3. Declararea unei funcții

O funcție trebuie să fie declarată înainte de utilizare. Putem realiza acest lucru în două moduri:
- amplasând în textul programului definiția funcției înaintea definiției funcției care o apelează - *main* sau alta funcție
- declarând prototipul (antetul) funcției înainte de definiția funcției care o apelează; în acest caz, definiția funcției poate fi amplasată oriunde în program.

Cu alte cuvinte, dacă funcția este definită după funcția în care este apelată, atunci este necesară o declarație anterioară a prototipului funcției.

Declarația unei funcții se face prin precizarea prototipului (antetului) funcției:

Sintaxa:

```
tip_rezultat_returnat nume_functie (lista_parametri_formali);
```

În prototip, numele parametrilor formali pot fi omişi, apărând doar tipul fiecăruia.
Exemple:

```
// prototip funcție, plasat înaintea locului în care va fi utilizată
double calculAria(double);  // fără numele parametrilor
int max(int, int);

/* Numele parametrilor din antet vor fi ignorate de compilator dar vor
servi la documentare: */
double calculAria(double raza);
int max(int numar1, int numar2);
```

Antetele funcțiilor sunt uzual grupate împreună şi plasate într-un aşa numit fişier antet (*header file).* Acest fişier antet poate fi inclus în mai multe programe.

Exemple:

O funcție max(int, int), care primeşte două numere întregi şi întoarce valoarea maximă. Funcția *max* este apelată din *main*.

```
int max(int, int); // prototip funcție (declarare)

int main() {
    printf("%d\n ", max(5, 8)); // apel al funcției max cu valori constante
    int a = 6, b = 9, c;
    c = max(a, b);              // apel max() cu variabile
    printf("%d\n ", c);

    printf("%d\n ", max(c, 99)); // apel max()

    return 0;
```

```
}

// Definire functie
int max(int num1, int num2) {
    return (num1 > num2) ? num1 : num2;
}
```

```
// functia lines definită după main, dar declarată înainte:
void lines ( int);      // declarație funcție

int main () {
   lines (3);           // utilizare funcție
}

void lines (int n) {
   for (int i=0; i<n; i++)
   printf("\n");
}
```

Prototipul implicit al unei funcții, este:
int nume_functie(void);

Cu alte cuvinte, în lipsa unei declarații de tip explicite se consideră că tipul implicit al funcției este *int*.
Şi argumentele formale fără un tip declarat explicit sunt considerate implicit de tipul *int*, dar nu trebuie abuzat de această posibilitate.
Exemplu:

```
rest (a,b) { //echivalent cu: int rest (int a, int b)
      return a%b;
}
```

În limbajul C se pot defini şi funcții cu număr variabil de argumente, care pot fi apelate cu număr diferit de argumente efective.

Mai jos apar două variante de scriere a programelor. Prima variantă, în care funcția *main* e prezentă la începutul programului, după prototipurile funcțiilor apelate, oferă o imagine a prelucrărilor realizate de program.

Varianta 1:	Varianta 2:
prototipuri definiția funcției main definiții funcții	definiții funcții definiția funcției main

IB.06.4. Domeniu de vizibilitate (scope)

În C/C++ există următoarea convenție :

Convenție C:
Un nume (variabilă, funcție) poate fi utilizat numai după ce a fost declarat, iar domeniul de vizibilitate (scope) al unui nume este mulțimea instrucțiunilor (liniilor de cod) în care poate fi utilizat acel nume (numele este vizibil).

Prin urmare, avem ca regulă de bază: identificatorii sunt accesibili doar în blocul de instrucțiuni în care au fost declarați; ei sunt necunoscuți în afara acestor blocuri.

Într-un program putem avea două categorii de variabile:

- **locale - variabilele declarate:**
  - **în funcții (corpul unei functii este un bloc de instructiuni)**
  - **în blocuri de instructiuni**
  - **ca parametri formali.**
- **externe (globale) - variabile definite în afara funcțiilor.**

Locul unde este definită o variabilă determină domeniul de vizibilitate al variabilei respective: o variabilă definită într-un bloc poate fi folosită numai în blocul respectiv, iar variabilele cu acelaşi nume din funcții (blocuri) diferite sunt alocate la adrese diferite şi nu nici o legătură între ele.
Numele unei variabile este unic (nu pot exista mai multe variabile cu acelaşi nume), dar o variabilă locală poate avea numele uneia globale, caz în care, în interiorul funcției, e valabilă noua semnificație a numelui.
Pentru variabilele locale memoria se alocă la activarea funcției/blocului (deci la execuție) şi este eliberată la terminarea executării funcției/blocului. Inițializarea variabilelor locale se face tot la execuție şi de aceea se pot folosi expresii pentru inițializarea lor (nu numai constante).

Exemple:

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

int fact=1;                    // declarare varriabila externa - globala

void factorial(int n)                // parametru formal
{ int i;                             // declarare variabila locala
  fact=1;
  for(i=2;i<=n;i++) fact=fact*i;
}
int main(void) {
  int v;                             // declarare variabila locala
  factorial(3);
  printf("3!=%d\n",fact);            // utilizare variabila externa
  printf("Introd o valoare:");
  // utilizare variabila locala:
  scanf("%d",&v);
  factorial(v);
  printf("%d!=%d\n",v,fact);
  return 0;
}
```

Atenție! Definiția unui identificator maschează pe cea a aceluiaşi identificator declarat într-un supranivel sau în fişier (global)! Apariția unui identificator face referință la declararea sa în cel mai mic bloc care conține această apariție!

```
#include<stdio.h>
#include<stdlib.h>
int fact=1;                    // declarare variabila externa - globala
void factorial(int n)
{
  int i;                             // declarare variabila locala
  int fact=1;                        /* declarare variabila locala, acesta
                                  va fi folosita in functie mai departe! */
  for(i=2;i<=n;i++) fact=fact*i;
}
```

```
int main(void)
{
  int v;                              // declarare variabila locala
  factorial(3);
  printf("3!=%d\n",fact);             // utilizare variabila externa fact!!!

   printf("Introd o valoare:");
  scanf("%d",&v);
  factorial(v);
  printf("%d!=%d\n",v,fact);         // utilizare variabila externa fact!!!

return 1;
}
```

Atenție! Presupunând ca valoarea introdusă pentru v este 3, se va afișa 3!=1 deoarece valoarea variabilei globale fact nu este modificată. Rezultatul este 1 oricare ar fi valoarea lui v !!!

Observații:

- Unul dintre motivele pentru care se vor evita variabile externe este acela că, din neatenție, putem defini variabile locale cu acelaşi nume ca variabila externă, ceea ce poate conduce la erori.
- Nu se recomandă utilizarea de variabile externe decât în cazuri rare, când mai multe funcții folosesc în comun mai multe variabile şi se doreşte simplificarea utilizării funcțiilor, sau în cadrul unor biblioteci de funcții.
- O funcție poate deci utiliza variabilele globale, cele locale, precum si parametrii formali. Pentru reutilizare şi pentru a nu modifica accidental variabilele globale, este indicat ca o funcție să nu acceseze variabile globale.

IB.06.5. Apelul unei funcții

Apelul unei funcții, adică utilizarea acesteia se poate face astfel:

Sintaxa:

```
nume_functie (lista_parametri_actuali)
/* poate apare ca operand intr-o expresie, dacă funcția returnează un rezultat */
```

Observații:

- **pentru funcție fără tip (void) apelul este:**
  nume_funcție (lista_parametrii_efectivi);
- **pentru funcție cu tip T, unde t este de tipul T, apelul poate fi:**
  t = nume_funcție (lista_parametri_efectivi);
  sau poate apare într-o expresie în care se foloseşte rezultatul ei:
  if (nume_funcție (lista_parametri_efectivi) == 1) …

Parametrii folosiți la apelul funcției se numesc parametri efectivi şi pot fi orice expresii (constante, funcții etc.). Parametrii efectivi trebuie să corespundă ca număr şi ca tipuri (eventual prin conversie implicită) cu parametrii formali (cu excepția unor funcții cu număr variabil de argumente).
Este posibil ca tipul unui parametru efectiv să difere de tipul parametrului formal corespunzător, cu condiția ca tipurile să fie "compatibile" la atribuire. Conversia de tip (între numere sau pointeri) se face automat, la fel ca la atribuire:

```
x = sqrt(2);            // param. formal double, param.efectiv int
y = pow(2,3);           //  arg. formale de tip double
```

În cazul funcțiilor *void* fără parametri, apelul se face prin:

Sintaxa:

```
nume_functie ( );       // instrucțiune expresie
```

La apelul unei funcții, pe stiva de apeluri (păstrată de Sistemul de Operare) se crează o *înregistrare de activare*, care cuprinde, de jos în sus:

- adresa de revenire din funcție
- valorile parametrilor formali
- valorile variabilelor locale.

În momentul apelării unei funcții, se execută corpul său, după care se revine în funcția apelantă, la instrucțiunea următoare apelului.
Revenirea dintr-o funcție se face fie la întâlnirea instrucțiunii *return*, fie la terminarea execuției corpului funcției (funcția poate să nu conțină instrucțiunea *return* doar în cazul funcțiilor *void*).

Exemple - *apelul funcțiilor* putere *şi* prim*:*

1. Să se calculeze şi să se afişeze valoarea expresiei $x^m+y^n+(xy)^{m\wedge n}$, x, y, m, n fiind citiți de la tastatură, astfel încât întregii m,n să fie pozitivi. Ridicarea la putere se va realiza printr-o funcție *putere* care primeşte baza şi exponentul ca parametri şi returnează rezultatul – vezi definirea unei funcții. Modul de apel al funcției *putere*:

```
int main(void){
  int m,n;
  double x,y;
     while( printf(" m(>=0):"), scanf("%d",&m), m<0);
     while( printf(" n(>=0):"), scanf("%d",&n), n<0);
  if ( m==0 && n==0 ) {      //semnalare eroare 0^0
          puts("0^0 imposibil");
          return 0;                    // din main
  }
  printf("valorile lui x,y:");
     scanf("%lf%lf",&x,&y);
     printf("%lf^%d+%lf^%d+(%lf*%lf)^%d^%d (cu putere ):%lf\n", x, m, y, n,
    x, y, m, n,  putere(x,m)+putere(y,n) +    putere(x*y,putere(m,n)));
    // apel functie putere scrisa de utilizator
  printf("Rezultat cu pow: %lf\n", pow(x,m)+pow(y,n)+pow(x*y,pow(m,n)));
    // apel functie pow din biblioteca math
     return 0;
}
```

2. Numărarea şi afişarea numerelor prime mai mici ca un întreg dat n. Pentru aceasta vom scrie o funcție care testează dacă un număr este prim – vezi definirea unei funcții. Modul în care se va apela (folosi) aceasta funcție:

```
int main () {
   int n,m,contor ;
      /* contor de numere prime*/

   printf ("n= "); scanf ("%d",&n);
   contor=0;
   for (m=2;m<=n;m++)                  // incearca numerele m
            if ( prim(m) == 1 ){       // dacă m este prim
```

```
                printf ("%d\n",m);
                contor++;
            }

  printf ("există  %d numere prime mai mici decat %d\n", contor,n);
  return 0;
}
```

IB.06.6. Instrucțiunea return

În corpul funcției se poate folosi instrucțiunea return pentru a returna o valoare funcției apelante (o valoare corespunzătoare antetului funcției).
Forme ale instrucțiunii return:

Sintaxa:

- **return;** **//dacă funcția e void**
- **return expresie;**

Expresia e de acelaşi tip cu tip_rezultat al funcției(eventual prin conversie implicită).

Corpul unei funcții poate conține una sau mai multe instrucțiuni `return`. În corpul unei funcții de tip *void*, se poate folosi instrucțiunea *return* - fără o valoare returnată - pentru a reda controlul apelantului, însă în acest caz, al funcțiilor *void*, utilizarea lui *return* este opțională; dacă lipseşte, se adaugă automat *return* ca ultimă instrucțiune.
În funcția *main* instrucțiunea *return* are ca efect terminarea întregului program.

Exemplu de program cu două funcții:

```
#include <stdio.h>
void clear () {                  // şterge ecran prin defilare
int i;                           // variabila locala funcției clear
for (i=0; i<24; i++)
  putchar('\n');
}
int main(){
  clear();                      // apel funcție
}
```

Observații:

- O funcție de un tip diferit de *void* trebuie să conțină cel puțin o instrucțiune *return* prin care se transmite rezultatul funcției:

```
// factorial de n
long fact (int n) {
long nf=1L;              // initializare rezultat
while (n)
   nf=nf * n--;          // echivalent cu: nf=nf * n; n=n-1;
return nf;               // rezultat funcție
}
```

- Compilatoarele C nu verifică şi nu semnalează dacă o funcție de un tip diferit de *void* conține sau nu instrucțiuni *return*, iar eroarea se manifestă la execuție.
Cuvântul *else* după o instrucțiune *return* poate lipsi, dar de multe ori este prezent pentru a face codul mai clar.
Exemplu fără else:

```
// transforma caracterul din variabila c in literă mare

char toupper (char c) {
  if (c>='a' && c<='z')            // daca c este litera mica
```

```
   return c+'A'-'a';                    // returnează cod litera mare
return c;                               // ramane neschimbat
}
```

- Instrucțiunea *return* poate fi folosită pentru ieşirea forțată dintr-un ciclu şi din funcție, cu reducerea lungimii codului sursă. Exemplu de funcție care verifică dacă un număr dat este prim:

```
int prim (int numar){
   int div;
   for(div=2; div<=sqrt(numar); div++)
      if (numar % div == 0) return 0;
   return 1;
}
```

IB.06.7. Transmiterea parametrilor

În C, transmiterea parametrilor se face prin valoare: parametrii actuali sunt transmişi prin valoare, la apelul funcției (valorile lor sunt depuse pe stiva program iar apoi copiate în parametrii formali. Modificarea valorii lor de către funcție nu este vizibilă în exterior!

Urmează un exemplu ce pune în evidență modul de transmitere a parametrilor prin valoare:

Observații:

- Dacă se dorește ca o funcție să modifice valoarea unei variabile, trebuie să i se transmită *pointerul* la variabila respectivă - vezi Pointeri!

- Dacă parametrul este un tablou, cum numele este echivalent cu pointerul la tablou, funcţia poate modifica valorile elementelor tabloului, primind adresa lui. A se observa că trebuie să se transmită ca parametri şi dimensiunea/ dimensiunile tabloului.
- Dacă parametrul este şir de caractere, dimensiunea tabloului de caractere nu trebuie să se transmită, sfârşitul şirului fiind indicat de caracterul terminator '\0'.

IB.06.8. Funcţii cu argumente vectori

Pentru argumentele formale de tip vector nu trebuie specificată dimensiunea vectorului între parantezele drepte, oricum aceasta va fi ignorată.
Exemplu:

```
// calcul valoare maxima dintr-un vector:

float maxim (float a[], int n ) {
      float max=a[0];
      for (int k=1;k<n;k++)
            if ( max < a[k]) max=a[k];
      return max;
}

// exemplu de utilizare - in main:

float xmax, x[]= {3,6,2,4,1,5,3};
xmax = maxim(x,7);
```

Un argument formal vector poate fi declarat şi ca pointer, deoarece are ca valoare adresa de început a zonei unde este memorat vectorul.
Exemplu:

```
float maxim (float *a, int n ) { ... }
```

De remarcat că pentru un *parametru efectiv* vector nu mai trebuie specificat explicit că este un vector, deoarece există undeva o declaraţie pentru variabila respectivă, care stabileşte tipul ei. Este chiar greşit sintactic să se scrie:

```
xmax = maxim ( x[ ], 7 ) ;          // NU !
```

O funcţie C nu poate avea ca rezultat direct un vector, dar poate modifica elementele unui vector primit ca parametru.
Exemplu de funcţie pentru ordonarea unui vector prin metoda bulelor – ***Bubble Sort***:

```
void sort (float a[ ], int n) {

      int n,i,j, gata;
      float aux;

      do {
            gata = 1;// presupunem initial ca nu sunt necesare schimbari de elemente
            for (i=0;i<n-1;i++) // compara n-1 perechi de elemente vecine
                        if ( a[i] > a[i+1] ) {// daca nu sunt in ordine crescatoare
                                    aux=a[i];
                                    a[i]=a[i+1];
                                    a[i+1]=aux;
                                    // am interschimbat a[i] cu a[i+1]
                                    gata =0;
                                    // seteaza ca s-a facut o schimbare
                  }
      } while (!gata); // repeta cat timp au mai fost schimbari de elemente
}
```

Probleme pot apare la parametrii efectivi de tip matrice din cauza interpretării diferite a zonei ce conține elementele matricei de către funcția apelată şi respectiv de funcția apelantă. Pentru a interpreta la fel matricea liniarizată este important ca cele două funcții să folosească acelaşi număr de coloane în formula de liniarizare. Din acest motiv nu este permisă absența numărului de coloane din declarația unui parametru formal matrice.

Exemple:

```
void printmat(int a[][10], int nl, int nc);     // corect, cu nc <= 10
void printmat(int a[][], int nl, int nc);       // greşit !
```

O altă soluție la problema parametrilor matrice este utilizarea de matrice alocate dinamic şi transmise sub forma unui pointer.
O sinteză a transmiterii parametrilor de tip tablou este dată în cele ce urmează:

Parametru formal	Parametru actual
tip_baza nume_vector[] **tip_baza nume_ vector[dim]**	**nume_vector**
tip_baza nume_ matrice[][dim2] //dim2 trebuie sa apara **tip_baza nume_ matrice[dim1][dim2]**	**nume_matrice**

IB.06.9. Funcții recursive

În continuare se va prezenta conceptul de recursivitate şi câteva exemple de algoritmi recursivi, pentru a putea înțelege mai bine această tehnică puternică de programare, ce permite scrierea unor soluții clare, concise şi rapide, care pot fi uşor înțelese şi verificate.

IB.06.9.1 Ce este recursivitatea ?

Un obiect sau un fenomen se defineşte în mod recursiv dacă în definiția sa există o referire la el însuşi.
Recursivitatea este folosită cu multa eficiență în matematică. Spre exemplu, definții matematice recursive sunt:

- **Definiția numerelor naturale:**

  0 € N
  dacă i € N, atunci succesorul lui i € N

- **Definiția funcției factorial**

  fact : N -> N

$$fact(n) = \begin{cases} 1, & \text{daca } n=0 \\ n * fact(n-1), & \text{daca } n>0 \end{cases}$$

- **Definiția funcției Ackermann**

  ac: N x N -> N

$$ac(m,n) = \begin{cases} n+1, & \text{dacă } m=0 \\ ac(m-1,1), & \text{dacă } n=0 \\ ac(m-1, ac(m,n-1)), & \text{dacă } m, n >0 \end{cases}$$

- **Definiția funcției Fibonacci**

fib : N -> N
fib(n) = { 1, dacă n=0 sau n=1
fib(n-2) + fib(n-1), dacă n>1

Utilitatea practică a recursivității rezultă din posibilitatea de a defini un set infinit de obiecte printr-o singură relație sau printr-un set finit de relații.

Recursivitatea s-a impus în programare odată cu apariția unor limbaje de nivel înalt, ce permit scrierea de module ce se autoapelează (PASCAL, LISP, ADA, ALGOL, C sunt limbaje recursive, spre deosebire de FORTRAN, BASIC, COBOL, nerecursive).
Recursivitatea este strâns legată de iterație. ***Iterația*** este execuția repetată a unei porțiuni de program, până la îndeplinirea unei condiții (exemple: *while, do-while, for* din C).
Recursivitatea presupune execuția repetată a unui modul, însă în cursul execuției lui (şi nu la sfârşit, ca în cazul iterației), se verifică o condiție, a cărei nesatisfacere implică reluarea execuției modulului de la început, fără ca execuția curentă să se fi terminat. În momentul satisfacerii condiției se revine în ordine inversă din lanțul de apeluri, reluându-se şi încheindu-se apelurile suspendate.
Un program recursiv poate fi exprimat: P = M (Si , P) , unde M este mulțimea ce conține instrucțiunile Si şi pe P însuşi.

Structurile de program necesare şi suficiente în exprimarea recursivității sunt funcțiile:

Definiție:
O funcție recursivă este o funcție care se apelează pe ea însăşi, direct sau indirect.

Recursivitatea poate fi directă - o funcție P conține o referință la ea însăşi, sau indirectă - o funcție P conține o referință la o funcție Q ce include o referință la P. Vom pune accentul mai ales pe funcțiile direct recursive.

Se pot deosebi două feluri de funcții recursive:

- Funcții cu un singur apel recursiv, ca ultimă instrucțiune, care se pot rescrie uşor sub forma nerecursivă (iterativă).
- Funcții cu unul sau mai multe apeluri recursive, a căror formă iterativă trebuie să folosească o stivă pentru memorarea unor rezultate intermediare.

Recursivitatea este posibilă în C datorită faptului că, la fiecare apel al funcției, adresa de revenire, variabilele locale şi argumentele formale sunt puse într-o stivă (gestionată de compilator), iar la ieşirea din funcție (prin *return*) se scot din stivă toate datele puse la intrarea în funcție (se "descarcă" stiva).

Exemplu generic:

```
void p(){ //functie recursiva
    p();  //apel infinit
}

//apelul trebuie conditionat in una din variantele:
   • if(cond) p();
   • while(cond) p();
   • do p() while(cond);
```

Exemplu de funcție recursivă de tip *void*:

```
void binar (int n) {                    // se afişeaza n in binar
  if (n>0) {
    binar(n/2);                         // scrie echiv. binar al lui n/2
    printf("%d",n%2);                   // şi restul impartirii n la 2
  }
}
```

Funcția de mai sus nu scrie nimic pentru *n=0*, dar poate fi uşor completată cu o ramură *else* la instrucțiunea *if*.

Apelul recursiv al unei funcții trebuie să fie condiționat de o decizie care să împiedice apelul în cascadă (la infinit); aceasta ar duce la o eroare de program - depăşirea stivei.

- **Orice funcție recursivă trebuie să conțină cel puțin o instrucțiune *if*, plasată de obicei chiar la început!**
- **Prin această instrucțiune se verifică dacă mai este necesar un apel recursiv sau se iese din funcție.**
- **Absența instrucțiunii *if* conduce la o recursivitate infinită (la un ciclu fără condiție de terminare)!**

Pentru funcțiile de tip diferit de *void* apelul recursiv se face printr-o instrucțiune *return*, prin care fiecare apel preia rezultatul apelului anterior!

Anumite funcții recursive corespund unor relații de recurență.

Exemple:

```
// Calculul a^n:
double putere (double a, int n) {
  if (n==0)       return 1.;            // a^0 = 1
  else return a * putere(a, n-1);       // a^n=a*a^(n-1)
}

// Algoritmul lui Euclid poate folosi o relație de recurență:
int cmmdc (int a,int b) {
  if ( a%b==0) return b;
  return cmmdc( b,a%b);                 // cmmdc(a,b)=cmmdc(b,a%b)
}
```

Observații:

- Funcțiile recursive nu conțin în general cicluri explicite (cu unele excepții), iar repetarea operațiilor este obținută prin apelul recursiv. O funcție care conține un singur apel recursiv ca ultimă instrucțiune poate fi transformată într-o funcție nerecursivă, înlocuind instrucțiunea *if* cu *while*.
- Fiecare apel recursiv are parametri diferiți, sau măcar o parte din parametri se modifică de la un apel la altul.
- Se pune întrebarea: *"Ce este mai indicat de utilizat: recursivitatea sau iterația?"* Algoritmii recursivi sunt potriviți pentru a descrie probleme care utilizează formule recursive sau pentru prelucrarea structurilor de date definite recursiv (liste, arbori), fiind mai eleganți şi mai simplu de înțeles şi verificat. Iterația este uneori preferată din cauza vitezei mai mari de execuție şi a memoriei necesare la execuție mai reduse. Spre exemplu, varianta recursivă a funcției Fibonacci duce la 15 apeluri pentru n=5, deci varianta iterativă este mult mai performantă.
- Apelul recursiv al unei funcții face ca pentru toți parametrii să se creeze copii locale apelului curent (în stivă) , acestea fiind referite şi asupra lor făcându-se modificările în timpul execuției curente a funcției. Când execuția funcției se termină, copiile sunt extrase din stivă, astfel încât

modificările operate asupra parametrilor nu afectează parametrii efectivi de apel, corespunzători. De asemenea, pentru toate variabilele locale se rezervă spațiu la fiecare apel recursiv.

- Pe parcursul unui apel, sunt accesibile doar variabilele locale şi parametrii pentru apelul respectiv, nu şi cele pentru apelurile anterioare, chiar dacă acestea poartă acelaşi nume!
- De reținut că, pentru fiecare apel recursiv al unei funcții se crează copii locale ale parametrilor valoare şi ale variabilelor locale, ceea ce poate duce la risipa de memorie.

IB.06.9.2 Verificarea şi simularea programelor recursive

Se face ca şi în cazul celor nerecursive, printr-o demonstrație formală, sau testând toate cazurile posibile:

- Se verifică întâi dacă toate cazurile particulare (ce se execută când se îndeplineşte condiția de terminare a apelului recursiv) funcționează corect.
- Se face apoi o verificare a funcției recursive, pentru restul cazurilor, presupunând că toate componentele din codul funcției funcționează corect. Verificarea e deci inductivă. Acesta e un avantaj al programelor recursive, ce permite demonstrarea corectitudinii lor simplu şi clar.

Exemplu: Funcția recursivă de calcul a factorialului

```
//varianta recursiva
int fact(int n){
     if(n==1)
          return 1;
     return n*fact(n-1); //apel recursiv
}
//varianta nerecursiva
int fact(int n){
     int i,f;
     for(i=f=1; i<=n; i++)
           f*=i;
```

Verificarea corectitudinii în acest caz cuprinde doi paşi:

1. pentru n=1 valoarea 1 ce se atribuie factorialului este corectă
2. pentru n>1, presupunând corectă valoarea calculată pentru predecesorul lui n de către *fact(n-1)*, prin înmulțirea acesteia cu *n* se obține valoarea corectă a factorialului lui n.

IB.06.9.3 Utilizarea recursivitatii în implementarea algoritmilor de tip *Divide et Impera*

Tehnica ***Divide et Impera***, fundamentală în elaborarea algoritmilor, constă în descompunerea unei probleme complexe în mai multe subprobleme a căror rezolvare e mai simplă şi din soluțiile cărora se poate determina soluția problemei inițiale (exemple: găsirea minimului si maximului valorilor elementelor unui tablou, căutarea binară, sortare Quicksort, turnurile din Hanoi).
Un algoritm de divizare general s-ar putea scrie:

```
void rezolva ( problema pentru x ) {
     dacă x e divizibil in subprobleme  {
        divide pe x in parti x1,...,xk
        rezolva(x1);
         ...
         rezolva(xk);
        combina solutiile partiale intr-o solutie pentru x
     }
     altfel
        rezolva pe x direct
}
```

da înjumătățirii intervalului de căutare. Se returnează indicele în vector sau *-1* dacă valoarea *v* nu se află în vector. Pentru o mai bună înțelegere, prima variantă este cea iterativă.

```
// varianta iterative, l- limita stanga, r - limita dreapta a intervalului
int bsearchIter(int v, int *a, int l, int r) {
   while(l<r) {
      int m = (l+r)/2;          // m - mijlocul intervalului
      if (v == a[m]) return m; // element gasit
      if (v > a[m]) l=m+1;      // reduc intervalul la jumatatea lui dreapta
      else r=m;                 // reduc intervalul la jumatatea lui stanga
      }
   return -1;                   // element negasit
}

// varianta recursiva
int bsearchRec (int v, int *a,int l, int r) {
      int m;
      if (l<r) {
         int m = (l+r)/2;
         if (v == a[l]) return l;
         else
            if ( v>a[m] )
                 return bsearchRec(v, a, m+1, r);
            else
                 return bsearchRec(v, a, l, m);
      }
      else return -1;
}

int main() {
    int v, a[N], n;
    …… // citire date intrare
    printf("Elem. %d se afla in poz %d\n ", v, bsearchRec(v,a,0,n));
    printf("Elem. %d se afla in poz %d\n ", v, bsearchIter(v,a,0,n));
    return 0;
}
```

IB.06.10. Anexă: Funcții în C++

În C++ toate funcțiile folosite trebuie declarate şi nu se mai consideră că o funcție nedeclarată este implicit de tipul *int*.

Absența unei declarații de funcții este eroare gravă în C++ şi nu doar avertisment ca în C (nu trece de compilare)!

În C++ se pot declara valori implicite pentru parametrii formali de la sfârşitul listei de parametri; aceste valori sunt folosite automat în absența parametrilor efectivi corespunzători la un apel de funcție. O astfel de funcție poate fi apelată deci cu un număr variabil de parametri.
Exemplu:

```
// afişare vector, precedat de un titlu (şirul primit sau şirul nul)
void printv ( int v[], int n, char * titlu="") {
   printf ("\n %s \n", titlu);
   for (int i=0; i<n; i++)
      printf ("%d ", v[i]);
}

// Exemple de apeluri:
printv ( x,nx );                          // cu 2 parametri
printv (a,na," multimea A este");         // cu 3 parametri
```

În C++ funcțiile scurte pot fi declarate inline, iar compilatorul înlocuieşte apelul unei funcții inline cu instrucțiunile din definiția funcției, eliminând secvențele de transmitere a parametrilor. Funcțiile inline sunt tratate ca şi macrourile definite cu *define*. Orice funcție poate fi declarată inline, dar compilatorul poate decide că anumite funcții nu pot fi tratate inline şi sunt tratate ca funcții obişnuite. De exemplu, funcțiile care conțin cicluri nu pot fi inline. Utilizarea unei funcții inline nu se deosebeşte de aceea a unei funcții normale. Exemplu de funcție *inline*:

```
inline int max (int a, int b) { return a > b ? a : b; }
```

În C++ pot exista mai multe funcții cu acelaşi nume dar cu parametri diferiți (ca tip sau ca număr). Se spune că un nume este *supraîncărcat* cu semnificații ("function overloading"). Compilatorul poate stabili care din funcțiile cu acelaşi nume a fost apelată într-un loc analizând lista de parametri şi tipul funcției:

```
float abs (float f) { return fabs(f); }
long abs (long x) { return labs(x); }
printf ("%6d%12ld  %f \n", abs(-2),abs(-2L),abs(-2.5) );
```

Capitolul IB.07. Pointeri. Pointeri şi tablouri. Pointeri şi funcţii

Cuvinte cheie
Pointer, referenţiere, dereferenţiere, indirectare, vectori şi pointeri, tablouri ca argumente ale funcţiilor, pointeri în funcţii, pointeri la funcţii, funcţii generice, tipul referinţă

IB.07.1. Pointeri

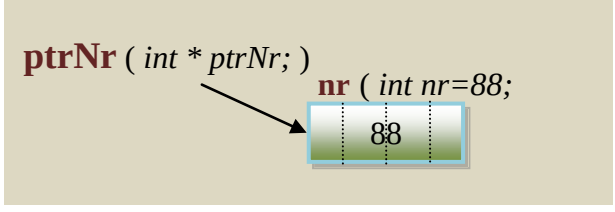
Pointerii reprezintă cea mai puternică caracteristică a limbajului C/C++, permiţând programatorului să acceseze direct conţinutul memoriei, pentru a eficientiza astfel gestiunea memoriei; programul şi datele sale sunt păstrate în memoria RAM (Random Access Memory) a calculatorului.
În acelaşi timp, pointerii reprezintă şi cel mai complex şi mai dificil subiect al limbajului C/C++, tocmai datorită libertăţilor oferite de limbaj în utilizarea lor.
Prin utilizarea corectă a pointerilor se paote îmbunătăţi drastic eficienţa şi performanţele programului. Pe de altă parte, utilizarea lor incorectă duce la apariţia multor probleme, de la cod greu de citit şi de întreţinut la greşeli penibile de genul pierderi de memorie sau depăşirea unei zone de date. Utilizarea incorectă a pointerilor poate expune programul atacurilor externe (hacking). Multe limbaje noi (Java, C#) au eliminat pointerii din sintaxa lor pentru a evita neplăcerile cauzate de aceştia.

O locaţie de memorie are o adresă şi un conţinut. Pe de altă parte, o variabilă este o locaţie de memorie care are asociat un nume şi care poate stoca o valoare de un tip particular. În mod normal, fiecare adresă poate păstra 8 biţi (1 octet) de date. Un întreg reprezentat pe 4 octeţi ocupă 4 locaţii de memorie. Un sistem ‘pe 32 de biţi’ foloseşte în mod normal adrese pe 32 de biţi. Pentru a stoca adrese pe 32 de biţi sunt necesare 4 locaţii de memorie.

Definiţie:
O variabilă pointer (pe scurt vom spune un pointer) este o variabilă care păstrează adresa unei date, nu valoarea datei.
Cu alte cuvinte, o variabilă pointer este o variabilă care are ca valori adrese de memorie. Aceste adrese pot fi:

- **Adresa unei valori de un anumit tip (pointer la date)**
- **Adresa unei funcţii (pointer la o funcţie)**
- **Adresa unei zone cu conţinut necunoscut (pointer la *void*).**

În figura următoare, s-a reprezentat grafic un pointer *ptrNr* care este un pointer la o variabilă *nr* (de tip *int*); cu alte cuvinte, *ptrNr* este o variabilă pointer ce stochează adresa lui *nr*. În general, vom reprezenta grafic pointerii prin săgeţi.



Un pointer poate fi utilizat pentru referirea diferitelor date şi structuri de date, cel mai frecvent folosindu-se pointerii la date. Schimbând adresa memorată în pointer, pot fi manipulate informaţii

situate la diferite locații de memorie, programul şi datele sale fiind păstrate în memoria RAM a calculatorului.
Deşi adresele de memorie sunt de multe ori numere întregi pozitive, tipurile pointer sunt diferite de tipurile întregi şi au utilizări diferite. Unei variabile pointer i se pot atribui constante întregi ce reprezintă adrese, după conversie .

În limbajul C tipurile pointer se folosesc în principal pentru:

- **declararea şi utilizarea de vectori, mai ales pentru vectori ce conțin şiruri de caractere;**
- **parametri de funcții prin care se transmit rezultate (adresele unor variabile din afara funcției);**
- **acces la zone de memorie alocate dinamic şi care nu pot fi adresate printr-un nume;**
- **parametri de funcții prin care se transmit adresele altor funcții.**

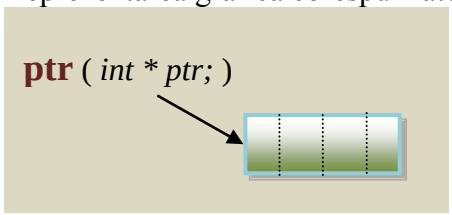
IB.07.2. Declararea pointerilor

Ca orice variabilă, pointerii trebuie declarați înainte de a putea fi utilizați.
În sintaxa declarării unui pointer se foloseşte caracterul * înaintea numelui pointerului. Declararea unei variabile (sau parametru formal) de un tip pointer include declararea tipului datelor (sau funcției) la care se referă acel pointer. Sintaxa declarării unui pointer la o valoare de tipul "tip" este:

Sintaxa:

```
tip * ptr;        // sau
tip* ptr;         //sau
tip  *ptr;
```

Reprezentarea grafică corespunzătoare acestei declarații este următoarea:



Exemple de variabile şi parametri pointer:

```
int * pi;                    // pi - adresa unui intreg sau vector de int
void * p;                    // p - adresa de memorie
int * * pp;                  // pp - adresa unui pointer la un intreg
char* str;  // str - adresa unui şir de caractere
```

Atunci când se declară mai multe variabile pointer de acelaşi tip, nu trebuie omis asteriscul care arată că este un pointer.
Exemple:

```
int *p, m;              // m de tip "int", p de tip "int *"
int *a, *b ;            // a şi b de tip pointer
```

Convenția de nume pentru pointeri sugerează să se pună un prefix sau sufix cu valoarea "`p`" sau "`ptr`". Exemplu: *`iPtr`*, *`numarPtr`*, *`pNumar`*, *`pStudent`*.

Dacă se declară un tip pointer cu *typedef* atunci se poate scrie astfel:

```
typedef int* intptr;          // intptr este nume de tip
intptr  p1, p2, p3;           // p1, p2, p3 sunt pointeri
```

Tipul unei variabile pointer este important pentru că determină câți octeți vor fi folosiți de la adresa conținută în variabila pointer şi cum vor fi interpretați. Un pointer la *void* nu poate fi utilizat pentru a obtine date de la adresa din pointer, deoarece nu se ştie câți octeți trebuie folosiți şi cum.

Există o singură constantă de tip pointer, cu numele *NULL* şi valoare zero, care este compatibilă la atribuire şi comparare cu orice tip pointer.

Observații:
Totuşi, se poate atribui o constantă întreagă convertită la un tip pointer unei variabile pointer:

```
char * p = (char*)10000;                  // o adresa de memorie
```

IB.07.3. Operații cu pointeri la date

IB.07.3.1 Inițializarea pointerilor, operația de referențiere (&)

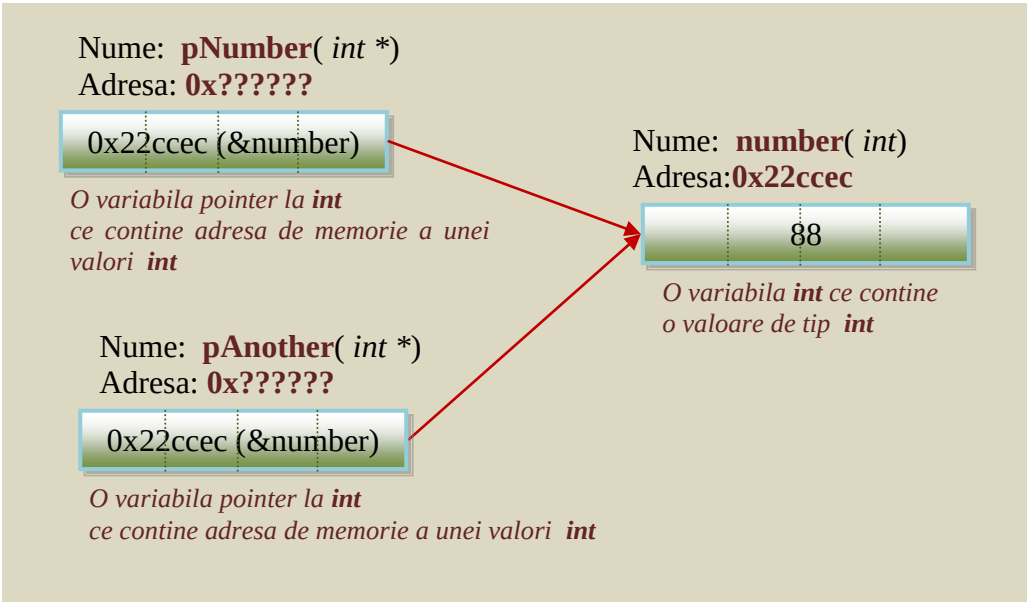
Atunci când declarăm un pointer, el nu este inițializat. Cu alte cuvinte, el are o valoare oarecare ce reprezintă o adresă a unei locații de memorie oarecare despre care bineînțeles că nu ştim dacă este validă (acest lucru este foarte periculos, pentru că poate fi de exemplu adresa unei alte variabile!). Trebuie să inițializăm pointerul atribuindu-i o adresă validă.
Aceasta se poate face în general folosind operatorul de referențiere - de luare a adresei - (`&`).

Sintaxa:
Operatorul unar *&* aplicat unei variabile are ca rezultat adresa variabilei respective (deci un pointer).

De exemplu, dacă ***nr*** este o variabilă de tip ***int*****,** ***&nr*** returnează adresa lui ***nr*****.** Această adresă o putem atribui unei variabile pointer:

```
int number = 88;        // o variabila int cu valoarea 88
int *pNumber;           // declaratia unui pointer la un intreg
pNumber = &number;      // atribuie pointerului adresa variabilei int
pNumber int *pAnother = &number;      /* declaratia unui pointer la un
                         intreg si initializare cu adresa variabilei int */
```

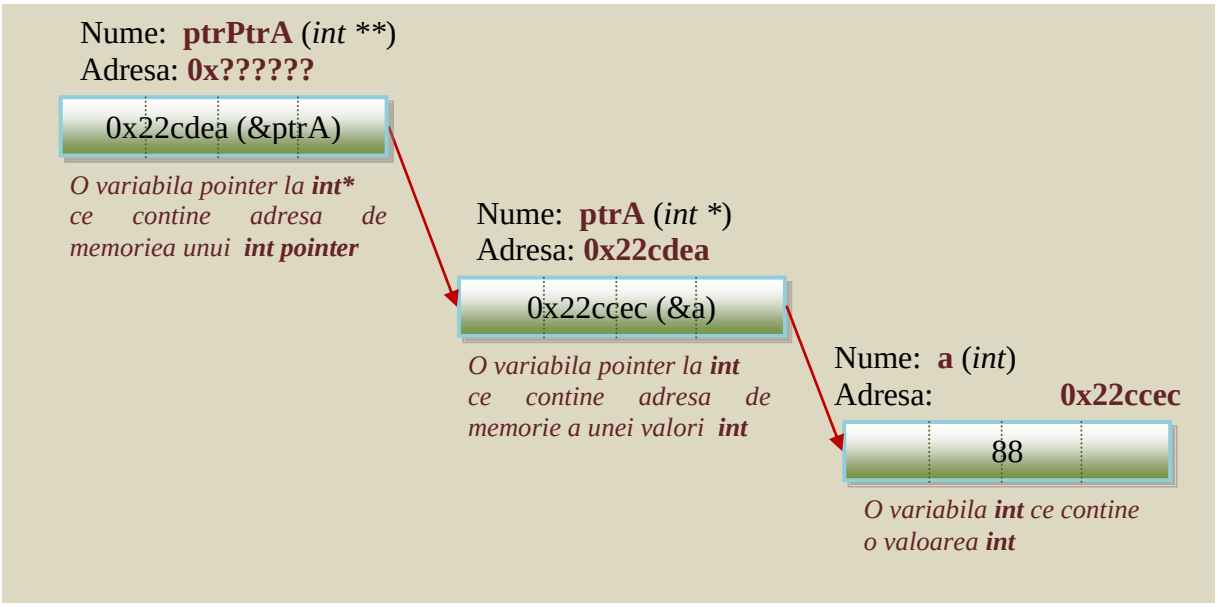
În figură, variabila *number*, memorată începând cu adresa `0x22ccec`, conține valoarea întreagă `88`. Expresia `&number` returnează adresa variabilei *number*, adresă care este `0x22ccec`. Această adresă este atribuită pointerului `pNumber`, ca valoare a sa inițială, dar şi pointerului *pAnother*, ca urmare cei doi pointeri vor indica aceeaşi celulă de memorie!



Exemplu:

```
int **ptrPtrA, *ptrA, a=1;
// ptrPtrA – pointer la un pointer la un intreg
// ptrA - pointer la un intreg
// a - variabila int cu valoarea 1
ptrA = &a;            // atribuie pointerului ptrA adresa variabilei int a
ptrPtrA = & ptrA;     /* atribuie pointerului ptrPtrA adresa variabilei
                                          pointer la int ptrA */
```

În figură, variabila *a*, memorată începând cu adresa `0x22ccec`, conține valoarea întreagă `88`. Expresia `&a` returnează adresa variabilei *a*, adresă care este `0x22ccec`. Această adresă este atribuită pointerului *ptrA*, ca valoare a sa inițială, iar pointerul *ptrPtrA* va avea ca valoare adresa pointerului *ptrA*!



IB.07.3. 2 Indirectarea sau operația de dereferențiere(*)

Indirectarea printr-un pointer (diferit de *void* *), pentru acces la datele adresate de acel pointer, se face prin utilizarea operatorul unar *, *operator de dereferențiere (indirectare)*.

Sintaxa:

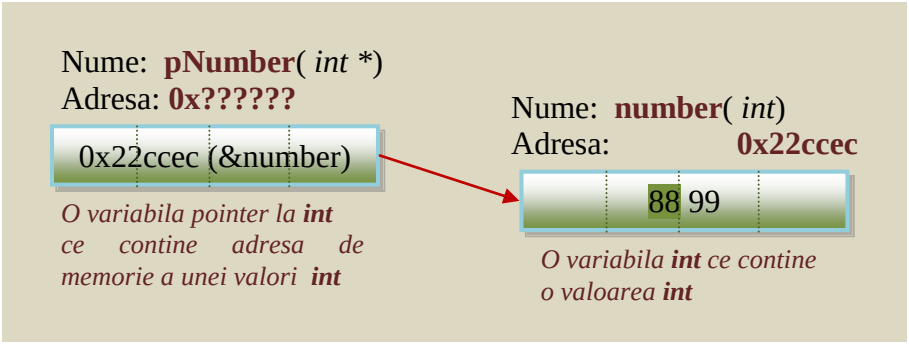
Operatorul unar * - operator de dereferențiere (indirectare) - returnează valoarea păstrată la adresa indicată de pointer.

Exemple:

```
int *p, m;  // p poinetr la int, m variabila int
m=*p;       // m ia valoarea indicata de pointerul p

int number = 88;
int *pNumber = &number;   /* Declara şi atribuie adresa variabilei number
             pointer-ului pNumber (acesta poate fi de exemplu 0x22ccec)*/
printf("%p\n", pNumber);   // Afiseaza pointerul (0x22ccec)
printf("%d\n", *pNumber);  /* Afiseaza valoarea indicata de pointer,
                              valoare care este de tip int (88)*/
*pNumber = 99;             /* Atribuie o valoare care va fi stocata la
            adresa indicata de pointer. Atentie! NU variabilei pointer!*/
printf("%d\n", *pNumber); /* Afiseaza noua valoare indicata de pointer,
                                                              99*/
printf("%d\n", number);    /* Valoarea variabilei number s-a schimbat de
                              asemenea (99)*/
```

`pNumber` stochează adresa unei locații de memorie; `*pNumber` se referă la valoarea păstrată la adresa indicată de pointer, sau altfel spus la valoarea indicată de pointer:



Putem spune că o variabilă face referire directă la o valoare, în timp ce un pointer face referire indirectă la o valoare, prin adresa de memorie pe care o stochează. Referirea unei valori în mod indirect printr-un pointer se numeşte **indirectare.**

Observație:

Simbolul `*` are înțelesuri diferite. Atunci când este folosit într-o declarație (`int *pNumber`), el denotă că numele care îi urmează este o variabilă de tip pointer. În timp ce, atunci când este folosit într-o expresie/instrucțiune (ex. `*pNumber = 99;` printf("%d\n", *pNumber);), se referă la valoarea indicată de variabila pointer.

IB.07.3.3 Operația de atribuire

În partea dreaptă poate fi un pointer de acelaşi tip (eventual cu conversie de tip), constanta *NULL* sau o expresie cu rezultat pointer.

Exemple:

```
int *p, *q=NULL;
float x=1.23;
p=q;
p=&x;
```

Unei variabile de tip *void** i se poate atribui orice alt tip de pointer fără conversie de tip explicită şi un argument formal de tip *void** poate fi înlocuit cu un argument efectiv de orice tip pointer. Atribuirea între alte tipuri pointer se poate face numai cu conversie de tip explicită (*cast*) şi permite interpretarea diferită a unor date din memorie. De exemplu, putem extrage cei doi octeți dintr-un întreg scurt astfel:

```
short n;
char * p = (char*) &n;
c1= *p;
c2 = *(p+1);
```

IB.07.3.4 Operații de comparație

Compararea a doi pointeri (operații relaționale cu pointeri) se poate face utilizând operatorii cunoscuți:

== **!=** **<** **>** **<=** **>=**

IB.07.3.5 Aritmetica pointerilor

Adunarea sau scăderea unui întreg la (din) un pointer, incrementarea şi decrementarea unui pointer se pot face astfel:

Sintaxa:

p++; ↔ **p=p+sizeof(tip);**
p--; ↔ **p=p-sizeof(tip);**
p=p+c; ↔ **p=p+c*sizeof(tip);**
p=p-c; ↔ **p=p-c*sizeof(tip);**

Exemplu:

```
void printVector( int a[], int n) {   // afişarea unui vector
  while (n--)
       printf ("%d ", *a++);
 }
```

Trebuie observat că incrementarea unui pointer şi adunarea unui întreg la un pointer nu adună întotdeauna întregul 1 la adresa conținută în pointer; valoarea adăugată (scăzută) depinde de tipul variabilei pointer şi este egală cu produsul dintre constantă şi numărul de octeți ocupat de tipul adresat de pointer.

Această convenție permite referirea simplă la elemente succesive dintr-un vector folosind indirectarea printr-o variabilă pointer.

O altă operație este cea de **scădere** a două variabile pointer de acelaşi tip (de obicei adrese de elemente dintr-un acelaşi vector), obținându-se astfel „distanța" dintre două adrese, atenție, nu în octeți ci în blocuri de octeți, în funcție de tipul pointerului.

Exemplu de funcție care întoarce indicele în şirul s1 a şirului s2 sau un număr negativ dacă s1 nu conține pe s2:

```
int pos ( char* s1, char * s2) {
  char * p =strstr(s1,s2); //p va fi adresa la care se găseşte s2 in s1
  if (p) return p-s1;
  else  return -1;
}
```

IB.07.3.6 Dimensiunea

Spațiul ocupat de o variabilă pointer se determină utilizând operatorul *sizeof*: Valoarea expresiei este 2 (în modelul small); oricare ar fi tip_referit, expresiile de mai jos conduc la aceeaşi valoare 2:

```
sizeof( &var )
sizeof( tip_referit * )
```

IB.07.3.7 Afişarea unui pointer

Tipărirea valorii unui pointer se face folosind funcția *printf* cu formatul *%p*, valoarea apărând sub forma unui număr în hexa.

Observații:

• Adresa unei variabile pointer este un pointer, la fel ca adresa unei variabile de orice alt tip: `&var_pointer`

Un pointer este asociat cu un tip şi poate conține doar o adresă de tipul specificat.

```
int i = 88;
double d = 55.66;
int *iPtr = &i;     // pointer int ce contine adresa unei variabile int
double *dPtr = &d; // pointer double ce indica spre o valoare double

iPtr = &d;      // EROARE, nu poate contine o adresa de alt tip
dPtr = &i;      // EROARE, nu poate contine o adresa de alt tip
iPtr = i;       /* EROARE, pointerul pastreaza adresa unui int,
                        NU o valoare int */

int j = 99;
iPtr = &j;      // putem schimba adresa continuta de un pointer
```

• O eroare frecventă este utilizarea unei variabile pointer care nu a primit o valoare (adică o adresă de memorie) prin atribuire sau prin inițializare la declarare. Inițializarea unui pointer se face prin atribuirea adresei unei variabile, prin alocare dinamică, sau ca rezultat al executării unei funcții.

Compilatorul nu generează eroare sau avertizare pentru astfel de greşeli.

Exemple incorecte:

```
int * a;          // declarata dar neinițializata     !!
scanf ("%d",a) ;  // citeşte la adresa conținuta in variabila a
int *iPtr;        // declarata dar neinițializata!!
*iPtr = 55;
print("%d\n",*iPtr);
```

Putem inițializa un pointer cu valoarea 0 sau `NULL`, aceasta însemnând că nu indică nicio adresă – pointer *null*. Dereferențierea unui pointer nul duce la o excepție de genul `STATUS_ACCESS_VIOLATION`, şi un mesaj de eroare 'segmentation fault', eroare foarte des întâlnită!

```
int *iPtr = 0;          // Declara un pointer int si-l initializeaza cu 0
print("%d\n",*iPtr);  // EROARE! STATUS_ACCESS_VIOLATION!
int *p = NULL;        // declara tot un pointer NULL
```

Inițializarea unui pointer cu NULL la declarare este o practică bună, deoarece elimină posibilitatea "uitării" inițializării cu o valoare validă!

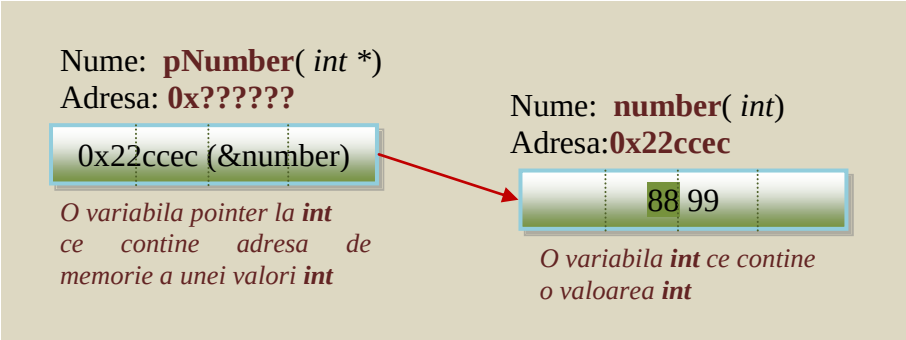
• void * înseamnă un pointer de tip neprecizat şi utilizarea acestui tip de pointeri ne permite păstrarea gradului de generalitate al unui program la maximum.

• Atenție însă: Nu putem face operații aritmetice asupra acestor pointeri sau asupra pointerului nul.

Exemplu:

```
/* Test pentru declarare si utilizare pointeri */
int number = 88;           // number - intreg cu valoarea initiala 88
int *pNumber = &number;    /* Declara şi atribuie adresa variabilei number
       pointer-ului pNumber (acesta poate fi de exemplu 0x22ccec)*/
printf("%p\n", pNumber);  // Afiseaza pointerul (0x22ccec)
```

```
printf("%p\n", &number);  // Afiseaza adresa lui number (0x22ccec)
printf("%d\n", *pNumber); /* Afiseaza valoarea indicata de pointer,
                             valoare care este de tip int (88)*/
*pNumber = 99;            /* Atribuie o valoare care va fi stocata la
        adresa indicata de pointer. Atentie! NU variabilei pointer!*/
printf("%p\n", pNumber);  // Afiseaza pointerul (0x22ccec)
printf("%p\n", &number);  // Afiseaza adresa lui number (0x22ccec)
printf("%d\n", *pNumber);  // Afiseaza noua valoare indicata de pointer 99
printf("%d\n", number);   /*Valoarea variabilei number s-a schimbat de
                                                    asemenea (99)*/
printf("%p\n", &pNumber);  //Afiseaza adresa pointerului pNumber 0x22ccf0
```



Notă: Valoarea pe care o veți obține pentru adresă este foarte puțin probabil să fie cea din acest exemplu!

Exemplu:

```
int *p, n=5, m;
p=&n;
m=*p;       // m este 5
m=*p+1;     // m este 6

int *p;
float x=1.23, y;
p=&x;
y=*p;             // valoare eronata pentru y!

int *a,**b, c=1, d;
a=&c;
b=&a;
d=**b;            // d este 1
```

IB.07.4 Vectori şi pointeri

Convenție!
***Numele unui tablou* este un pointer constant spre primul element (index 0) din tablou.**

Cu alte cuvinte, o variabilă de tip tablou conține adresa de început a acestuia (adresa primei componente) şi de aceea este echivalentă cu un pointer la tipul elementelor tabloului. Aceasta echivalență este utilizată de obicei în argumentele de tip tablou şi în lucrul cu tablouri alocate dinamic.
Expresiile de mai jos sunt deci echivalente:

nume_tablou ⇔ &nume_tablou ⇔&nume_tablou[0]
şi:
***nume_tablou ⇔ nume_tablou[0]**
***(nume_tablou +i) ⇔ nume_tablou [i]**

În concluzie, există următoarele echivalențe de notație pentru un vector *a*:

a[0]	***a**
&a[0]	a
a[1]	*(a+1)
&a[1]	a+1
a[k]	*(a+k)
&a[k]	a+k

Declarații echivalente pentru tablouri:
tip v [dim1] [dim2]…[dimn];
⇔
tip *…*v;

Exemple:

```
int v[10]; // vector cu dimensiune fixă
                                                                    int
*v=(int *)malloc(10*sizeof(int));        // vector alocat dinamic

// Referire elemente pentru ambele variante de declarare:
   v[i]            // sau:
   *(v+i)
```

```
int i;
double v[100], x, *p;
p=&v[0];          // corect, neelegant
p=v;
x=v[5];
x=*(v+5);
v++;              // incorect
p++;              //corect
Obs:
p[4]=2.5          //corect sintactic, dar nu e alocată memorie pentru p!!!
```

IB.07.5 Transmiterea tablourilor ca argumente ale funcțiilor

Un tablou este trimis ca parametru unei funcții folosind pointerul la primul element al tabloului. În declararea funcției putem folosi notația specifică tabloului (ex. `int[]`) sau notația specifică pointerilor (ex. `int*`). Compilatorul îl tratează întotdeauna ca pointer (ex. `int*`). De exemplu, pentru declararea unei funcții care primeşte un vector de întregi şi dimensiunea lui avem următoarele declarații echivalente:

```
void printVec (int a [ ], int n);
void printVec (int * a, int n);
void printVec (int a [50 ], int n);
```

Ele vor fi tratate ca `int*` de către compilator. Dimensiunea din parantezele drepte este ignorată. Numărul de elemente din tablou trebuie trimis separat, sub forma unui al doilea parametru de tip *int*. Compilatorul nu va lua în calcul acest parametru ca dimensiune a tabloului şi ca urmare nu va verifica dacă această dimensiune se încadrează în limitele specificate (>0 şi <dim_maximă_declarată).
Elementele unui tablou pot fi modificate în funcție, deoarece se transmite adresa acestuia – prin referință (vezi Transmiterea parametrilor – cap IB.06).

În interiorul funcției ne putem referi la elementele vectorului *a* fie prin indici (cu *a[i]*), fie prin indirectare (**(a+i)*), indiferent de felul cum a fost declarat vectorul *a*.

```
// prin indexare
void printVec (int a[ ], int n) {
   int i;
   for (i=0;i<n;i++)
     printf (%6d",a[i]);
}

// prin indirectare
void printVec (int *a, int n) {
   int i;
   for (i=0;i<n;i++)
     printf (%6d", *a++);
}
//Citirea elementelor unui vector se poate face asemănător:
for (i=0;i<n;i++)
   scanf ("%d", a+i);          // echivalent cu &a[i] şi cu a++
```

Apelul funcției se poate face astfel:

```
int main()
{
   int v[10], n;
   …
   printVec(v,n);
   …
}
      // SAU:
int main()
{
   int *v, n;     /* Atentie! Vectorul v nu are alocata memorie, va trebui
                       sa ii alocam dinamic!!*/
   …
   printVec(v,n);
   …
}
```

```
#include <stdio.h>
#include <stdlib.h>
#define N 5

int citire1(int tab[]){
   //citeste elementele lui tab prin accesarea indexata a elementelor
   int i=0;
      printf("Introduceti elementele tabloului:\n");
      while(scanf("%d",&tab[i]!=EOF) i++;
      return i;
}
```

```
void tiparire1(int *tab, int n){
   //tipareste elementele tabloului prin accesarea indexata a elementelor
       int i;
       printf("Elementele tabloului:\n");
       for(i=0;i<n;i++)
           printf("%d ",tab[i]);
       printf("\n");
}

int citire2(int tab[]){
   /* citeste elementele lui tab - accesarea fiecarui element se
   face printr-un pointer la el */
       int *pi;
       pi=tab;
       printf("Introduceti elementele tabloului:\n");
       while(scanf("%d",pi)!=EOF) pi++;
       return pi-tab;
}

void tiparire2 (int tab[], int n){
    // tipareste elementele lui tab prin accesare prin pointeri
        int *pi;
        printf("Elementele tabloului:\n");
        for (pi=tab; pi<tab+n; pi++)
          printf("%d ",*pi);
        printf("\n");
}

int main(){
   int tab1[N], tab2[N], n, m;
   n=citire1(tab1);
   tiparire1(tab1,n);
   m=citire2(tab2);
   tiparire2(tab2,m);
       return 0;
}
```

Program pentru citirea unui vector de întregi şi extragerea elementelor distincte într-un al doilea vector, care se va afisa. Se vor utiliza funcţii. Ce funcţii trebuie definite?

```
#define MAX 30
#include <stdio.h>
/* cauta pe x în vectorul a*/
int gasit(int *v, int n, int x){
   int m=0,i;
   for (i=0;i<n; i++)
     if (v[i]==x)  return i;
   return -1;
}

int main () {
   int a[MAX];           // un vector de intregi
   int b[MAX];           // aici se pun elementele distincte din a
   int n,m,i,j;          // n=dimensiune vector a, m=dimensiune vector b
   printf("Numar de elemente vector n="); scanf("%d",&n);
   printf ("Introducere %d numere intregi:\n",n);

   // citire vector:
   for (i=0;i<n;i++) scanf("%d",&a[i]);

   m=0;
   for (i=0;i<n;i++)
     if(gasit(b,m,a[i])==-1) b[m++]=a[i];
```

```
   // afiseaza elemente vector b:
   printf("Elementele distincte sunt:");
   for (j=0;j<m;j++)
     printf ("%5d",b[j]);

      return 0;
}
```

Pentru declararea unei funcții care primeşte o matrice ca parametru avem următoarele posibilități:

- **int min(int t[][NMAX], int m, int n);**
- **int min(int *t [NMAX], int m, int n);**
- **int min(int **t, int m, int n);**

Apelul funcției se va face astfel:

```
int a[NMAX][NMAX], m, n;
…..
int mimim = min( a, m, n);
```

Observații:

- În aplicațiile numerice se preferă argumentele de tip vector şi adresarea cu indici, iar în funcțiile cu şiruri de caractere se preferă argumente de tip pointer şi adresarea indirectă prin pointeri.
- O funcție poate avea ca rezultat un pointer dar nu şi rezultat vector.
- Diferența majoră dintre o variabilă pointer şi un nume de vector este aceea că un nume de vector este un pointer constant (adresa este alocată de compilatorul C şi nu mai poate fi modificată la execuție). Un nume de vector nu poate apare în stânga unei atribuiri, în timp ce o variabilă pointer are un conținut modificabil prin atribuire sau prin operații aritmetice.

Exemple:

```
int a[100], *p;
p=a; ++p;                          // corect
a=p; ++a;                          // ambele instrucțiuni produc erori
```

- Când un nume de vector este folosit ca argument, se transmite un pointer cu aceeaşi valoare ca numele vectorului, iar funcția poate folosi argumentul formal în stânga unei atribuiri.
- Declararea unui vector (alocat la compilare) nu este echivalentă cu declararea unui pointer, deoarece o declarație de vector alocă memorie şi inițializează pointerul ce reprezintă numele vectorului cu adresa zonei alocate (operații care nu au loc automat la declararea unui pointer).

```
int * a;  a[0]=1;       // greşit !
int *a={3,4,5};         // echivalent cu: int a[]={3,4,5}
```

- Operatorul *sizeof* aplicat unui nume de vector cu dimensiune fixă are ca rezultat numărul total de octeți ocupați de vector, dar aplicat unui argument formal de tip vector (sau unui pointer la un vector alocat dinamic) are ca rezultat mărimea unui pointer:

```
float x[10];
float * y=(float*)malloc (10*sizeof(float));   /*vector caruia i s-a
                    alocat dinamic memorie pentru 10 numere float */
printf ("%d,%d \n",sizeof(x), sizeof(y));      // scrie 40, 4
```

- Numărul de elemente dintr-un vector alocat la compilare sau inițializat cu un şir de valori se poate afla prin expresia: sizeof (x) / sizeof(x[0]).

```
   int x[10];
   printf ("%d\n",sizeof(x));              // scrie 40
   printf ("%d\n",sizeof(x[0]));           // scrie 4
   printf ("%d\n",sizeof(x)/sizeof(x[0]);  // scrie 10
```

IB.07.6 Pointeri în funcții

Reamintim că în C/C++, parametrii efectivi sunt transmişi prin valoare - valorile parametrilor actuali sunt depuse pe stivă, fiind apoi prelucrate ca parametri formali de către funcție. Ca urmare, modificarea valorii lor de către funcție nu este vizibilă în exterior! Un exemplu clasic este o funcție care *încearcă* să schimbe între ele valorile a două variabile, primite ca argumente:

```
void swap (int a, int b) {
   int aux;
   aux=a;
   a=b;
   b=aux;
}

int main () {
   int x=3, y=7;
   swap(x,y);
   printf ("%d,%d \n", x, y);       // scrie 3, 7  nu e ceea ce ne doream!
   return 0;
}
```

Pentru a înțelege mai bine mecanismul transmiterii parametrilor prin valoare oferim următoarea detaliere a paşilor efectuați în cazul funcției de interschimbarea a valorilor a două variabile:

În multe situații, se dorește modificarea parametrilor unei funcții. Acest lucru poate fi realizat prin transmiterea ca parametru al funcției a unui pointer la obiectul a cărui valoare vrem să o modificăm, modalitate cunoscută sub numele de transmitere prin referință.

Observație: Se pot modifica valorile de la adresele trimise ca parametri! Nu se pot modifica adresele trimise!

O funcție care:
- **trebuie să modifice mai multe valori primite prin argumente, sau**
- **care trebuie să transmită mai multe rezultate calculate de funcție**

trebuie să folosească argumente de tip pointer.

Versiunea corectă pentru funcția *swap* este următoarea:

```
void swap (int * pa, int * pb) {          // pointeri la intregi
   int aux;
   aux=*pa;
   *pa=*pb;
   *pb=aux;// Adresare indirecta pt a accesa valoarile de la adresele pa, pb
}

// apelul acestei funcții folosește argumente efective pointeri:

int main(void)
{
   int x=5, y=7;
   swap(&x, &y);
       //transmitere prin adresă
   printf("%d %d\n", x, y);
   /*valorile sunt inversate adică se va afișa 7 5*/
   return 0;
}
```

Exemple:

```
/* calcul nr²   */
#include <stdio.h>

void square(int *pNr) {
   *pNr *= *pNr; //Adresare indirecta pt a accesa valoarea de la adresa pNr
}

int main() {
   int nr = 8;
   printf("%d\n", nr); // 8
   square(&number);    // transmitere prin referinta explicita - pointer
   printf("%d\n", nr); // 64
   return 0;
}
```

Pentru a înțelege mai bine mecanismul transmiterii parametrilor prin pointeri oferim următoarea detaliere a pașilor efectuați în cazul funcției de interschimbarea a valorilor a două variabile:

Observaţii:

- O funcție care primeşte două sau mai multe numere pe care trebuie să le modifice va avea argumente de tip pointer sau un argument vector care reuneşte toate rezultatele (datele modificate).
- Dacă parametrul este un tablou, funcția poate modifica valorile elementelor tabloului, primind adresa tabloului. A se observa că trebuie să se transmită ca parametri şi dimensiunea/dimensiunile vectorului/matricii. Dacă parametrul este şir de caractere, dimensiunea vectorului de caractere nu trebuie să se transmită, sfârşitul şirului fiind indicat de caracterul terminator de şir, *\0*!
- O funcție poate avea ca rezultat un pointer, dar acest pointer nu trebuie să conțină adresa unei variabile locale, deoarece o variabilă locală are o existență temporară, garantată numai pe durata executării funcției în care este definită (cu excepția variabilelor locale statice) şi de aceea adresa unei astfel de variabile nu trebuie transmisă în afara funcției, pentru a fi folosită ulterior. Un rezultat pointer este egal cu unul din argumente, eventual modificat în funcție, fie o adresă obținută prin alocare dinamică (care rămâne valabilă si după terminarea funcției). Pentru detalii a se vedea **IB.10.**
  Exemplu corect:

```
int *plus_zece( int *a ) {
        *a=*a+10;
   return a;
}
```

Exemplu de programare greşită:

```
// Vector cu cifrele unui nr intreg
```

```
int *cifre (int n) {
      int k , c[5]; // Vector local
      for (k=4; k>=0; k--) {
            c[k]=n%10;
            n=n/10;
      }
      return c; // Gresit
}
```

- Pointerii permit:
  - să realizăm modificarea valorilor unor variabile transmise ca parametri unei funcții;
  - să accesăm mult mai eficient tablourile;
  - să lucrăm cu zone de memorie alocate dinamic;
  - să se acceseze indirect o valoare a unui tip de date.

Exemple:
Program pentru determinarea elementelor minim şi maxim dintr-un vector într-o aceeaşi funcție. Funcția nu are tip (void)!

```
#include<stdio.h>
void minmax ( float x[], int n, float* pmin, float* pmax) {
        float xmin, xmax;
        int i;
        xmin=xmax=x[0];
        for (i=1;i<n;i++) {
            if (xmin > x[i])  xmin=x[i];
            if (xmax < x[i]) xmax=x[i];
         }
         *pmin=xmin;
         *pmax=xmax;
      }

// utilizare funcție
int main () {
        float a[]={3,7,1,2,8,4};
        float a1,a2;
        minmax (a,6,&a1,&a2);
        printf("%f %f \n",a1,a2);
        getchar();
        return 0;
}

// NU!!!
int main () {
        float a[]={3,7,1,2,8,4};
        float *a1, *a2; // pointeri neinitializati !!!
        minmax (a,6,a1,a2);
        printf("%f %f \n",*a1,*a2);
        getchar();
        return 0;
}
```

Să se scrie o funcție care calculează valorile unghiurilor unui triunghi, în funcție de lungimile laturilor. Funcția va fi scrisă în două variante:

- cu 6 argumente: 3 date şi 3 rezultate
- cu 2 argumente de tip vector.

```
//varianta 1 cu 6 argumente: 3 date şi 3 rezultate
```

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

void unghiuri(float ab, float ac, float bc, float *a, float *b, float *c)
{
   *a=acos((ab*ab+ac*ac-bc*bc)/(2*ab*ac))*180/M_PI;
   *b=acos((ab*ab+bc*bc-ac*ac)/(2*ab*bc))*180/ M_PI;
   *c=acos((bc*bc+ac*ac-ab*ab)/(2*bc*ac))*180/ M_PI;
}

int main(){
   float a, b, c, ab, ac, bc;
   scanf("%f%f%f", &ab, &ac, &bc);
   unghiuri(ab, ac ,bc, &a ,&b, &c);
   printf("%f %f %f" , a, b, c);
   return 0;
}

//varianta 2 cu 2 argumente de tip vector

#include <stdio.h>
#include <math.h>
void unghiuri(float *L , float *U )
{
   U[0]=acos((L[0]*L[0]+L[1]*L[1]-L[2]*L[2])/(2*L[0]*L[1]))*180/ M_PI;
   U[1]=acos((L[0]*L[0]+L[2]*L[2]-L[1]*L[1])/(2*L[0]*L[2]))*180/ M_PI;
   U[2]=acos((L[2]*L[2]+L[1]*L[1]-L[0]*L[0])/(2*L[2]*L[1]))*180/ M_PI;
}

int main(){
   float L[2],U[2];
   int i;
   for (i=0;i<=2;i++)
       scanf("%f",&L[i]);

   unghiuri(L,U);

   for (i=0;i<=2;i++)
       printf("%f ",U[i]);
   return 0;
}
```

IB.07.7 Pointeri la funcții

Anumite aplicații numerice necesită scrierea unei funcții care să poată apela o funcție cu nume necunoscut, dar cu prototip şi efect cunoscut. De exemplu, o funcție care:

- să sorteze un vector ştiind funcția de comparare a două elemente ale unui vector
- să calculeze integrala definită a oricărei funcții cu un singur argument
- să determine o rădăcină reală a oricărei ecuații (neliniare).

Aici vom lua ca exemplu o funcție *listf* care poate afişa (lista) valorile unei alte funcții cu un singur argument, într-un interval dat şi cu un pas dat. Exemple de utilizare a funcției *listf* pentru afişarea valorilor unor funcții de bibliotecă:

```
int main () {
  listf (sin,0.,2.*M_PI, M_PI/10.);
  listf (exp,1.,20.,1.);
  return 0;
}
```

Problemele apar la definirea unei astfel de funcții, care primeşte ca argument numele (adresa) unei funcții.

Convenție C:
Numele unei funcții neînsoțit de o listă de argumente este adresa de început a codului funcției şi este interpretat ca un pointer către funcția respectivă.

Deci *sin* este adresa funcției *sin(x)* în apelul funcției *listf*.

Declararea unui parametru formal (sau unei variabile) de tip pointer la o funcție are forma următoare:

Sintaxa:

tip_returnat (*pf) (lista_param_formali);

unde:

- ***pf* este numele paramatrului (variabilei) pointer la funcție,**
- ***tip_returnat* este tipul rezultatului funcției.**
- ***lista_ param_ formali* include doar tipurile parametrilor.**

Observație:
Parantezele sunt importante, deoarece absența lor modifică interpretarea declarației. Astfel, declarația:

```
tip * f (lista_param_formali)
```

introduce o funcție cu rezultat pointer, nu un pointer la funcție!!

De aceea, o eroare de programare care trece de compilare şi se manifestă la execuție, este apelarea unei funcții fără paranteze; compilatorul nu apelează funcția şi consideră că programatorul vrea să folosească adresa funcției!

Exemplu:

```
if ( test ) break;      /* gresit, echiv. cu if (1) break; deoarece e luat
                        in calcul pointerul la functia test */

if ( test() ) break;    // aici se testeaza valoarea intoarsa de functie
```

În concluzie, definirea funcției *listf* este:

```
void listf (double (*fp)(double), double min, double max, double pas) {
   double x,y;
   for (x=min; x<=max; x=x+pas) {
      y=(*fp)(x);                                    // sau:  y=fp(x);
      printf ("\n%20.10lf %20.10lf", x,y);
   }
}
```

Pentru a face programele mai explicite se pot defini nume de tipuri pentru tipuri pointeri la funcții, folosind declarația ***typedef***.

```
typedef double (* ftype) (double);
void listf (ftype fp, double min, double max, double pas) {
  double x, y;
  for (x=min; x<=max; x=x+pas) {
            y = fp(x);
            printf ("\n%20.10lf %20.10lf", x,y);
   }
}
```

Exemple:

1. Program cu meniu de opțiuni; operatorul alege una dintre funcțiile realizate de programul respectiv.

```
#include<stdio.h>
#include<stdio.h>
typedef void (*funPtr) (); /* defineste tipul funPtr, care este pointer la
o functie de tip void fara argument */

// funcții pentru operatii realizate de program
void  unu () {
  printf ("unu\n");
}

void  doi () {
  printf ("doi\n");
}

void  trei () {
  printf ("trei\n");
}

// selectare şi apel funcție
int main () {
  funPtr tp[ ]= {unu,doi,trei};     // vector de pointeri la funcții
  short option=0;
  do{
     printf("Optiune (1/2/3):");
     scanf ("%hd", &option);
     if (option >=1 && option <=3)
                 tp[option-1]();         // apel funcție (unu/doi/trei)
            else break;
  }while (1);
  return 0;
}

//Secvența echivalentă, fara a folosi pointeri la functii, este:
do {
   printf("Optiune (1/2/3):");
   scanf ("%hd", &option);
   switch (option) {
     case 1: unu(); break;
     case 2: doi(); break;
     case 3: trei(); break;
   }
} while (1);
```

2. Program pentru operații aritmetice între numere întregi (doar adunare şi scădere, se poate completa):

```
/* Test pointeri la functii (TestFunctionPointer.cpp) */
#include <stdio.h>

int aritmetica(int, int, int (*)(int, int));
    /* int (*)(int, int) este un pointer la o functie,
      care primeste doi intregi si returneaza un intreg */
int add(int, int);
int sub(int, int);

int add(int n1, int n2) { return n1 + n2; }
int sub(int n1, int n2) { return n1 - n2; }

int aritmetica(int n1, int n2, int (*operation) (int, int)) {
```

```
   return (*operation)(n1, n2);
}

int main() {
   int number1 = 5, number2 = 6;

   // adunare
   printf("%d\n", aritmetica(nr1, nr2, add));
   // scadere
   printf("%d\n", aritmetica(nr1, nr2, sub));
   return 0;
}
```

IB.07.8 Funcții generice

În fişierul *stdlib.h* sunt declarate patru funcții generice pentru sortarea, căutarea liniară şi căutarea binară într-un vector cu componente de orice tip, care ilustrează o modalitate simplă de generalizare a tipului unui vector. Argumentul formal de tip vector al acestor funcții este declarat ca *void** şi este înlocuit cu un argument efectiv pointer la un tip precizat (nume de vector). Un alt argument al acestor funcții este adresa unei funcții de comparare a unor date de tipul celor memorate în vector, funcție furnizată de utilizator şi care depinde de datele folosite în aplicația sa.
Pentru exemplificare urmează declarațiile pentru trei din aceste funcții ("lfind" este la fel cu "lsearch"):

void *bsearch (const void *key, const void *base, size_t nelem, size_t width, int (*fcmp)(const void*, const void*));

void *lsearch (const void *key, void *base, size_t * pnelem, size_t width, int (*fcmp)(const void *, const void *));

void qsort (void *base, size_t nelem, size_t width, int (*fcmp)(const void *, const void *));

- *base* este adresa vectorului
- *key* este cheia (valoarea) căutată în vector (de acelaşi tip cu elementele din vector)
- *width* este dimensiunea unui element din vector (ca număr de octeți)
- *nelem* este numărul de elemente din vector
- *fcmp* este adresa funcției de comparare a două elemente din vector.

Exemplul următor arată cum se poate ordona un vector de numere întregi cu funcția *qsort* :

```
// functie pentru compararea a doua numere intregi
int intcmp (const void * a, const void * b) {
  return *(int*)a-*(int*)b;
}

int main () {
   int a[]= {5,2,9,7,1,6,3,8,4};
   int i, n=9;
            // n=dimensiune vector
   qsort ( a,9, sizeof(int),intcmp);            // ordonare vector
   for (i=0;i<n;i++)                            // afişare rezultat
     printf("%d ",a[i]);
}
```

IB.07.9 Anexă. Tipul referință în C++

În C++ s-a introdus tipul referință, folosit în primul rând pentru parametri modificabili sau de dimensiuni mari, dar şi funcțiile care au ca rezultat un obiect mare pot fi declarate de un tip referință, pentru a obține un cod mai performant.

Sintaxa:
Caracterul ampersand (&) folosit după tipul şi înaintea numelui unui parametru formal sau al unei funcții arată compilatorului că pentru acel parametru se primeşte adresa şi nu valoarea argumentului efectiv.

Spre deosebire de un parametru pointer, un parametru referință este folosit de utilizator în interiorul funcției la fel ca un parametru transmis prin valoare, dar compilatorul va genera automat indirectarea prin pointerul transmis (în programul sursă nu se foloseşte explicit operatorul de indirectare *).

Exemplu:

```
void schimb (int & x, int & y) {    // schimba intre ele doua valori
   int t = x;
   x = y;
   y = t;
}
void sort ( int a[], int n ) {                  // ordonare vector
   ...
   if ( a[i] > a[i+1])  schimb ( a[i], a[i+1]);
   ...
}
```

Sintaxa declarării unui tip referință este următoarea:

Sintaxa:
tip & nume
unde nume poate fi:
- **numele unui parametru formal**
- **numele unei funcții (urmat de lista argumentelor formale)**
- **numele unei variabile (mai rar).**

Efectul caracterului & în declarația anterioară este următorul: compilatorul creează o variabilă *nume* şi o variabilă pointer la variabila *nume*, inițializează variabila pointer cu adresa asociată lui *nume* şi reține că orice referire ulterioară la *nume* va fi tradusă într-o indirectare prin variabila pointer anonimă creată.

O funcție poate avea ca rezultat o referință la un vector dar nu poate avea ca rezultat un vector. O funcție nu poate avea ca rezultat o referință la o variabilă locală, aşa cum nu poate avea ca rezultat un pointer la o variabila locală.

Referințele simplifică utilizarea unor parametri modificabili de tip pointer, eliminând necesitatea unui pointer la pointer.
Exemplu de funcție care primeşte adresa unui şir şi are ca rezultat adresa primei litere din acel şir:

```
void skip (char * & p){
   while ( ! isalpha(*p))
      p++;
}
```

Parametrul efectiv transmis unei funcții pentru un parametru referință trebuie să fie un pointer modificabil şi deci nu poate fi numele unui vector alocat la compilare:

```
int main () {
    char s[]=" 2+ana -beta "; // un sir
    char *p=s;          // nu se poate scrie:  skip(s);
    skip(p);
    puts(p);
       return 0;
}
```

Există riscul modificării nedorite, din neatenție, a unor parametri referință, situație ce poate fi evitată prin copierea lor în variabile locale ale funcției.

Capitolul IB.08. Şiruri de caractere. Biblioteci standard

Cuvinte cheie

Şiruri de caractere, funcţii de intrare/ieşire şiruri de caractere, biblioteca string, extragere atomi lexicali, funcţii stdlib, funcţii ctype, erori, argumente în linia de comandă

IB.08.1. Şiruri de caractere în C

În limbajul C nu există un tip de date *şir de caractere*, deşi există constante şir. Reamintim că o constantă şir de caractere se reprezintă între ghilimele.

Definiţie:
Un şir de caractere este un vector de caractere terminat cu caracterul '\0'.

Există însă anumite particularităţi în lucrul cu şiruri faţă de lucrul cu alţi vectori.

Prin natura lor, şirurile pot avea o lungime variabilă în limite foarte largi, iar lungimea lor se poate modifica chiar în cursul execuţiei unui program ca urmare a unor operaţii cum ar fi concatenarea a două şiruri, ştergerea sau inserţia într-un şir .

Pentru simplificarea listei de parametri şi a utilizării funcţiilor pentru operaţii cu şiruri s-a decis ca fiecare şir memorat într-un vector să fie terminat cu un caracter numit *terminator de şir*, caracterul *\0* ce are codul ASCII egal cu zero, şi astfel să nu se mai transmită explicit lungimea şirului. Multe funcţii care produc un nou şir precum şi funcţiile standard de citire adaugă automat un octet terminator la şirul produs (citit), iar funcţiile care prelucrează sau afişează şiruri detectează sfârşitul şirului la primul octet zero.
Exemplu: Şirul de caractere *"Anul 2012"* ocupă 10 octeţi, ultimul fiind \0.

Există două posibilităţi de definire a şirurilor:

- **ca tablou de caractere;**

Exemple:

```
char sir1[30];
char sir2[10]="exemplu";

#define MAX_SIR 100
char s[MAX_SIR];
```

- **ca pointer la caractere (iniţializat cu adresa unui şir sau a unui spaţiu alocat dinamic). La definire se poate face şi iniţializare:**

Exemple:

```
char *sir3;  /* sir3 trebuie iniţializat cu adresa unui şir sau a unui
                spaţiu alocat pe heap */
sir3=sir1;                      // sir3 ia adresa unui şir static
sir3=sir1; ⇔ sir3=&sir1;  ⇔ sir3=&sir1[0];     // sunt echivalente
sir3=(char *)malloc(100);       // se aloca dinamic un spatiu pe heap
char *sir4="test";              / * sir2 este initializat cu adresa
                               sirului constant */
```

IB.08.2. Funcțiile de intrare/ieşire pentru şiruri de caractere sunt:

Funcție	Exemple utilizare	Descriere	Rezultat
char * gets(char * s);	char sir[10]; gets(sir);	Citeşte caractere până la întâlnirea lui *Enter*; acesta nu se adaugă la şirul s. Plasează */0* la sfârşitul lui s. Obs: codul lui *Enter* e scos din buffer-ul de intrare.	Returnează adresa primului caracter din şir. Dacă se tastează CTRL+Z returnează *NULL*.
int puts(const char * s);	char sir[10]; puts(sir);	Tipăreşte şirul primit ca parametru, apoi *NewLine*, cursorul trecând la începutul rândului următor	Returnează codul ultimului caracter din şir sau *EOF* la insucces
int scanf("%s", char* s);	char sir[10]; scanf("%s",sir);	Citeşte caractere până la întâlnirea primului blanc sau *Enter*; acestea nu se adaugă la şirul *s*. Plasează *\0* la sfârşitul lui *s*. Codul lui blanc sau *Enter* rămân în buffer-ul de intrare	Returnează numărul de valori citite sau EOF în cazul în care se tastează *CTRL/Z*
int printf("%s", char*s);	char sir[10]; printf("%s",sir);	Tipăreşte şirul *s*.	Returnează numărul de valori tipărite sau EOF în cazul unei erori.

Citirea unei linii care poate include spații albe se va face cu *gets*. Citirea unui cuvânt (şir delimitat prin spații albe) se va face cu *scanf*.
Exemplu de citire şi afişare linii de text, cu numerotare linii:

```
char lin[128];                                  // linii de maxim 128 car
int nl=0;
while ( gets (lin) != NULL) {                   // citire linie in lin
     printf ("%4d ",++nl);                      // mareste numar linie
     printf (%d: %s \n", nl, lin);              // scrie numar şi conținut linie
}
```

Nu se recomandă citirea caracter cu caracter a unui şir, cu descriptorul *%c* sau cu funcția *getchar*, decât după apelul funcției *fflush*, care goleşte zona tampon de citire. În caz contrar se citeşte caracterul *\n* (cod 10), care rămâne în zona tampon după citire cu *scanf* sau cu *getchar*.

Pentru a preveni erorile de depăşire a zonei alocate pentru citirea unui şir se poate specifica o lungime maximă a şirului citit în funcția *scanf*.
Exemplu:

```
char nume[30];
while(scanf("%30s",nume) != EOF) //numai primele 30 de caractere citite
   printf ("%s \n", nume);
```

Din familia funcțiilor de intrare-ieşire se consideră că fac parte şi funcțiile standard *sscanf* şi *sprintf*, care au ca prim argument un şir de caractere ce este analizat (*scanat*) de *sscanf* şi respectiv produs

de *sprintf* (litera *s* provine de la cuvântul *string*). Aceste funcții se folosesc fie pentru conversii interne în memorie, după citire sau înainte de scriere din/în fișiere text, fie pentru extragere de subșiruri dintr-un șir cu delimitatori diferiți de spații.
Exemplu:

```
char d[]="25-12-1989";
int z, l, a;
sscanf(d, "%d-%d-%d", &z, &l, &a);        //z=25, l=12, a=1989
printf ("\n %d, %d, %d \n", z, l, a);     //25, 12, 1989
```

IB.08.3. Funcții standard pentru operații cu șiruri

Funcțiile standard pentru șiruri de caractere sunt declarate în fișierul *string.h*. Urmează o descriere puțin simplificată a celor mai folosite funcții:

Funcție	Descriere
• **char* strcpy(char *s1, const char *s2)** • **char* strncpy(char *s1, const char *s2, unsigned int n)**	• Copiază la adresa s1 tot șirul de la adresa *s2* (inclusiv terminator șir) • Copiază primele *n* caractere de la s2 la s1 Returnează s1
• **int strcmp(const char *s1, const char *s2)** • **int strncmp(const char *s1, const char *s2)**	• Compară șirurile de la adresele *s1* și *s2* • Compară primele *n* caractere din șirurile *s1* și *s2* Au următorul rezultat: • 0 dacă șirurile comparate conțin aceleași caractere (sunt identice) • < 0 dacă primul șir (*s1*) este inferior celui de al doilea șir (*s2*) • > 0 dacă primul șir (*s1*) este superior celui de al doilea șir (*s2*)
unsigned int strlen(const char *s)	Returnează lungimea sirului s, excluzând '\0'. char *msg="Hello"; strlen(msg); //5
• **char* strcat(char *s1, const char *s2)** • **char* strncat(char *s1, const char *s2, unsigned int n)**	• Adaugă șirul s2 la sfârșitul șirului *s1* • Adaugă primele *n* caractere de la adresa *s2* la șirul *s1*
char* strtok(char *s1, const char *s2)	Împarte s1 în tokeni – atomi lexicali, s2 conține delimitatorii
• **char * strchr (char *s, char c);** • **char *strrchr (char *s, char c);** • **char * strstr (char *s1, char *s2);**	• Returnează adresa lui *c* în șirul s (prima apariție a lui *c*) • Returnează adresa ultimei apariții a lui *c* în șirul s • Returnează adresa în șirul s1 a șirului *s2*
char * strdup (char *s);	Crearea unei copii a unui șir (alocă memorie și copiază conținutul lui *s)*

Observații:

- Rezultatul funcției de comparare nu este doar -1, 0 sau 1 ci orice valoare întreagă cu semn, deoarece comparația de caractere se face prin scădere.

- Funcțiile standard *strcpy* și *strcat* adaugă automat terminatorul zero la sfârșitul șirului produs de funcție! Funcția *strncpy* nu adaugă subșirului copiat la adresa *d* terminatorul de șir dacă *n* < *strlen(s)* dar subșirul este terminat cu zero dacă *n* > *strlen(s).*

- Funcțiile pentru operații pe șiruri nu pot verifica depășirea memoriei alocate pentru șiruri, deoarece primesc numai adresele șirurilor; cade în sarcina programatorului să asigure memoria necesară rezultatului unor operații cu șiruri.

- Funcțiile de copiere și de concatenare întorc rezultatul în primul parametru (adresa șirului destinație) pentru a permite exprimarea mai compactă a unor operații succesive pe șiruri.
  Exemplu:

```
char fnume[20], *nume="test", *ext="cpp";
strcat( strcat( strcpy( fnume,nume),"."),ext);
```

- Utilizarea unor șiruri constante în operații de copiere sau de concatenare poate conduce la erori prin depășirea memoriei alocate (la compilare) șirului constant specificat ca șir destinație. Exemplu:

```
     strcat ("test",".cpp");       // efecte nedorite !
```

- Funcția *strdup* alocă memorie la o altă adresă, copiază în acea memorie șirul *s* și întoarce adresa noului șir. Se folosește pentru crearea de adrese distincte pentru mai multe șiruri citite într-o aceeași zona buffer.
  Exemplu:

```
int main () {
   char buf[40];
   int i=0, j;
   char* tp[100];        //tp este un tabel de pointeri la şiruri
   while (gets(buf))     //gets are rezultat zero la sfârşit de fişier
     tp[i++]= strdup(buf);    //copiaza şirul citit şi memoreaza adresa
   for (j=0; j<i; j++)   //afişarea şirurilor folosind tabelul de pointeri
     puts(tp[j]);
      return 0;
}
```

IB.08.4. Extragerea atomilor lexicali

Un cuvânt sau un atom lexical (*token*) se poate defini în două feluri:

- **un şir de caractere separat de alte şiruri prin unul sau câteva caractere cu rol de separator între cuvinte (de exemplu, spații albe);**
- **un şir care poate conține numai anumite caractere şi este separat de alți atomi prin oricare din caracterele interzise în şir.**

În primul caz sunt puțini separatori de cuvinte și aceştia pot fi enumerați.

Pentru **extragerea de şiruri separate prin spații albe** (‘ ‘, ’\n’, ’\t’, ’\r’) se poate folosi o funcție din familia “scanf”:

- “fscanf” pentru citire dintr-un fişier
- “sscanf” pentru extragere dintr-un şir aflat în memorie

Între şiruri pot fi oricâte spații albe, care sunt ignorate.

int sscanf(const char **str*, const char **format*, ...);

Exemplu de funcție care furnizează cuvântul k dintr-un şir dat s:

```
char* kword (const char* s, int k) {
   char word[256];
   while ( k >= 1) {
     sscanf(s,"%s",word);       // extrage un cuvant din linie in word
```

```
      s=strstr(s, word)+ strlen(word);     // trece peste cuvantul extras
      k--;
    }
    return strdup(word);              // returneaza o copie a cuvantului k
}
```

Pentru **extragere de cuvinte separate prin câteva caractere** se poate folosi funcția de biblioteca *strtok*:

char *strtok(char *_str1_, const char *_str2_);

Exemplu care afişează atomii lexicali dintr-o linie de text:

```
char linie[128], * cuv;       // adresa cuvant in linie
char *sep=".,;\t\n "                // şir de caractere separator
gets(linie);                        // citire linie
cuv=strtok (linie,sep);       // primul cuvant din linie
while ( cuv !=NULL) {
     puts (cuv);                    // scrie cuvant
     cuv=strtok(0,sep);             // urmatorul cuvant din linie
  }
```

Funcția ***strtok*** are ca rezultat un pointer la următorul cuvânt din linie şi adaugă un octet zero la sfârşitul acestui cuvânt, dar nu mută la altă adresă cuvintele din text. Acest pointer este o variabilă locală statică în funcția *strtok*, deci o variabilă care îşi păstrează valoarea între apeluri succesive. Extragerea de cuvinte este o operație frecventă, dar funcția *strtok* trebuie folosită cu atenție deoarece modifică şirul primit, iar primul apel este diferit de următoarele apeluri.

In *stdlib.h* sunt declarate câteva funcții folosite pentru extragerea unor numere dintr-un text (dintr-un sir de caractere) şi care ilustrează o altă soluție pentru funcții care primesc o adresă într-un şir, extrag un subşir şi modifică adresa în cadrul şirului (după subşirul extras). Este vorba de funcțiile:

- ***double strtod*** (char *s, char **p); - string to double
- ***long strtod*** (char *s, char **p); - string to long

care extrag, la fiecare apel, un subşir care reprezintă un număr şi au două rezultate: valoarea numărului în baza zece şi adresa imediat următoare numărului extras în şirul analizat.

Exemplu de utilizare:

```
int main () {
            char * str =" 1   2.2   3.333       44e-1 ";
            char * p = str;
            double d;
            do {
                  d = strtod (p,&p);
                  printf ("%lf\n", d);
            } while (d != 0);
     return 0;
}
```

IB.08.5. Alte funcții de lucru cu şiruri de caractere

Din biblioteca **stdlib:**

Funcție	Descriere
`int atoi(char* s)`	Transformă un şir într-un int
`double atof(char* s)`	Transformă un şir într-un double

Din biblioteca **ctype:**

Funcție	Descriere

`int isalpha(int ch)`
`int isdigit(int ch)`
`int isalnum(int ch)`
	Returnează non-zero (true) dacă ch este literă sau cifră;
	sau zero (false) altfel

`int isspace(int ch)`	Verifică dacă ch este spațiu alb (Space, CR, LF, Tab, FF, VTab)
`int isupper(int ch)` `int islower(int ch)`	Verifică dacă ch este literă mare/mică
`int toupper(int ch)` `int tolower(int ch)`	Convertește la literă mare/mică

IB.08.6. Erori uzuale la operații cu șiruri de caractere

O primă eroare posibilă este aceea că numele unui vector este un pointer şi nu mai trebuie aplicat operatorul & de obținere a adresei în funcția *scanf*, aşa cum este necesar pentru citirea în variabile simple.
O eroare de programare (şi care nu produce întotdeauna erori la execuție) este utilizarea unei variabile pointer neinițializate în funcția *scanf* (sau *gets*).
Exemplu greşit:

```
char * s;                 // corect este: char s[80];  80= lungime maxima
scanf("%s",s);            // citeşte la adresa conținuta in s
```

O altă eroare frecventă (nedetectată la compilare) este compararea adreselor a două şiruri în locul comparației celor două şiruri.
Exemplu:

```
char a[50], b[50];              // aici se memoreaza doua şiruri
scanf (%50s%50s", a,b); // citire şiruri a şi b
if (a==b) printf("egale\n");  //greşit,rezultat zero
```

Pentru comparare corectă de şiruri se va folosi funcția *strcmp*.
Exemplu:

```
if (strcmp(a,b)==0)  printf ("egale\n");
```

Aceeaşi eroare se poate face şi la compararea cu un şir constant.
Exemple:

```
if ( nume == "." ) break; ...}                  // greşit !
if ( strcmp(nume,".") == 0 ) break;... }        // corect
```

Din aceeaşi categorie de erori face parte atribuirea între pointeri cu intenția de copiere a unui şir la o altă adresă, deşi o parte din aceste erori pot fi semnalate la compilare. *Exemple*:

```
char a[100], b[100], *c ;
a = b;      // eroare semnalata la compilare
c = a;      // corect sintactic dar nu copiaza şir (doar modifica  c)
c=strdup(a); // copiaza şir de la adresa a la adresa c, aloca mem pt c
strcpy (a,b); // copiaza la adresa a şirul de la adresa b
```

IB.08.7. Definirea de noi funcții pe şiruri de caractere

Funcțiile standard pe şiruri de caractere din C lucrează numai cu adrese absolute (cu pointeri) şi nu folosesc ca rezultat sau ca argumente adrese relative în şir (indici întregi). De exemplu, funcția *strstr* care caută într-un şir un subşir are ca rezultat un pointer: adresa în şirul cercetat a subşirului găsit. Parametrii de funcții prin care se reprezintă şiruri se declară de obicei ca pointeri dar se pot declara şi ca vectori.

La definirea unor noi funcții pentru operații pe șiruri programatorul trebuie să fie sigur de adăugarea terminatorului de șir la rezultatul funcției, pentru respectarea convenției și evitarea unor erori. Pentru realizarea unor noi operații cu șiruri se vor folosi pe cât posibil funcțiile existente.

Deoarece nu există funcții care să elimine un caracter dintr-un șir, care să insereze un caracter într-un șir sau care să extragă un subșir dintr-o poziție dată a unui șir, le vom defini în continuare:

```
// şterge n caractere de la adresa "d"
char * strdel ( char *d, int n) {
   if ( n < strlen(d))
     strcpy(d,d+n);
   return d;
}

// inserează şirul s la adresa d
void strins (char *d, char *s) {
   char *aux=strdup(d);
   strcpy(d,s);
   strcat(d,aux);
}
```

În general, nu se recomandă funcții care au ca rezultat adresa unei variabile locale, deși erorile de utilizare a unor astfel de funcții apar numai la apeluri succesive (în cascadă).

Precizări la declararea funcțiilor standard sau nestandard pe șiruri :

- Parametrii sau rezultatele care reprezintă lungimea unui șir sunt de tip *size_t* (echivalent de obicei cu *unsigned int*) și nu *int*, pentru a permite șiruri de lungime mai mare.
- Parametrii prin care se reprezintă adrese de șiruri care nu sunt modificate de funcție se declară ***const* (*const char * str*)**, interpretat ca ***pointer la un şir constant (nemodificabil)***.

Exemplu:

```
size_t  strlen (const char * s);
```

Cuvântul cheie *const* în fata unei declarații de pointer cere compilatorului să verifice că funcția care are un astfel de argument nu modifică datele de la acea adresă. Toate funcțiile de bibliotecă cu pointeri la date nemodificabile folosesc declarația *const*, pentru a permite verificări în alte funcții scrise de utilizatori dar care folosesc funcții de bibliotecă.

Exemple

1. Variante de implementare a funcțiilor de bibliotecă *strlen, strcmp, strcpy* şi *strcat*.

```
int strlen( char *s){
   int lg=0;
      while (s[lg]!='\0')
            lg++;
   return lg;
}

int strcmp( char *s1, char *s2){
    int i;
    for(i=0; s1[i] || s2[i]; i++)
       if (s1[i] < s2[i])
           return -1;
       else
         if (s1[i] > s2[i])
           return 1;
   return 0;
}
```

```
char *strcpy( char *d, char *s){
   int i=0;
   while(s[i]){
      d[i]=s[i];
            i++;
   }
   d[i]='\0';   // sau d[i]=0;
   return d;
}

// secvența ce cuprinde liniile cu verde este echivalentă cu:
//    while(d[i]=s[i]) i++;


char *strcat(char *d, char *s){
  int i=0,j=0;

  while(d[i]) i++;
  /*  la  iesirea  din  while,  i  este  indicele  caracterului  terminator*/
  while(d[i++]=s[j++]);
  return d;
}
```

2. Program care:

- **citeşte cuvinte tastate fiecare pe câte un rând nou, până la CTRL/Z (varianta: până la introducerea unui cuvânt vid)**
- **afişează cuvântul cel mai lung**
- **afişează cuvintele ce încep cu o vocală**

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define LUNG 81    //lungime maxima cuvant
#define NR 15       // nr max de cuvinte citite

void citire_cuv ( char tab_cuv[][LUNG], int *nr_cuv ) {
      printf( "Se introduc maxim %d cuvinte, terminate cu CTRL/Z:\n", NR );
      while(*nr_cuv<NR && gets ( tab_cuv[*nr_cuv] ) )
                  (*nr_cuv)++;
  /* la CTRL/Z gets returneaza NULL (= 0) */

  /* citirea se poate face şi cu scanf:
        while(*nr_cuv<NR && scanf("%s",tab_cuv[*nr_cuv])!=EOF) (*nr_cuv)++; */
  /* dacă terminarea se face cu un cuvant vid:
        while(*nr_cuv<NR && strcmp("",gets(tab_cuv[*nr_cuv]))) (*nr_cuv)++; */
}

void cuv_max ( char tab_cuv[][LUNG], int nr_cuv ){
  int i, lung_crt, lung_max=0;
  char * p_max;
  /* pointerul spre cuvantul maxim */
  /* se poate memora indicele cuvantului maxim: int i_max;
  sau memora cuvantul maxim intr-un şir: char c_max[LUNG]; */

  for(i=0;i<nr_cuv;i++)
    if ( ( lung_crt = strlen(tab_cuv[i]) ) > lung_max){
      p_max = tab_cuv[i];
      lung_max = lung_crt;
    }
  printf ("Cuvantul de lungime maxima %d este: %s\n", lung_max, p_max);
}

void cuv_vocale ( char tab_cuv[][LUNG], int nr_cuv ){
```

```
  int i;
  puts("Cuvintele ce incep cu vocale:");
  for(i=0;i<nr_cuv;i++)
      switch(toupper(tab_cuv[i][0])){
      case 'A': case'E': case 'I': case 'O': case 'U':
                  puts(tab_cuv[i]);
      }
   /* în loc de switch se putea folosi:
  char c;
  if(c=toupper(tab_cuv[i][0]),c=='A' || c=='E' || ...) puts(tab_cuv[i]); */
}

int main(){
  char tab_cuv[NR][LUNG];     //vectorul de cuvinte
  int nr_cuv=0;               // numarul cuvintelor introduse
  citire_cuv(tab_cuv,&nr_cuv);
  cuv_max(tab_cuv,nr_cuv);
  cuv_vocale(tab_cuv,nr_cuv);
  return 1;
}
```

3. Se citesc trei şiruri: s1, s2 şi s3. Să se afişeze şirul obținut prin înlocuirea în s1 a tuturor aparițiilor lui s2 prin s3. (Observație: dacă s3 este şirul vid, din s1 se vor şterge toate subşirurile s2).

```
#include <stdio.h>
#include <string.h>
#define N 81

int main(void){
  char s1[N],s2[N],s3[N],rez[N];
  char *ps1=s1,*pos, *r=rez;

  puts("sirul s1:"); gets(s1);
  puts("subsirul s2:"); gets(s2);
  puts("s3:"); gets(s3);

  while ( pos=strstr(ps1,s2) ){
    while(ps1<pos) *r++=*ps1++;          //copiez în r din s1 pana la pos
    strcpy(r,s3);                         //copiez în r pe s3
    r+=strlen(s3);                        //sar peste s3 copiat în r
    ps1+=strlen(s2);                      //sar în s1 peste s2
  }
  strcpy(r,ps1);                          //adaug ce a mai ramas din s1
  puts("sirul rezultat:");
  puts(rez);
  return 1;
}
```

IB.08.8. Argumente în linia de comandă

Funcția *main* poate avea două argumente, prin care se pot primi date prin linia de comandă ce lansează programul în execuție. Sistemul de operare analizează linia de comandă, extrage cuvintele din linie (şiruri separate prin spații albe), alocă memorie pentru aceste cuvinte şi introduce adresele lor într-un vector de pointeri (alocat dinamic).

Primul argument al funcției *main* este dimensiunea vectorului de pointeri (de tip *int*), iar al doilea argument este adresa vectorului de pointeri (un pointer).

Primul cuvânt, cu adresa în *argv[0]*, este chiar numele programului executat (numele fişierului ce conţine programul executabil), iar celelalte cuvinte din linie sunt date iniţiale pentru program: nume de fişiere folosite de program, opţiuni de lucru diverse.
Programele care preiau date din linia de comandă se vor folosi în acelasi mod ca şi comenzile predefinite ale sistemului (*DIR, TYPE*, etc.), deci extind comenzile utilizabile în linie de comandă.
Exemplu de afişare a datelor primite în linia de comandă:

```
// fisierul se numeste listare.c
int main ( int argc, char * argv[]) {     // sau :  char** argv
   int i;
   for (i=0;i<n;i++)                  // nu se afişeaza şi argv[0]
     printf ("%s ", argv[i]);
}
```

O linie de commandă de forma:
listare iata patru argumente
va lansa în execuţie listare.exe, care va tipări pe ecran:
listare.exe
iata
patru
argumente

Capitolul IB.09. Structuri de date. Definire şi utilizare în limbajul C

Cuvinte cheie
Structura(struct), câmp, typedef, structuri predefinite, structuri cu conținut variabil (union), enumerări

IB.09.1. Definirea de tipuri şi variabile structură

O structură este o colecție de valori eterogene ca tip, stocate într-o zonă compactă de memorie.

Cu alte cuvinte, o structură este un tip de date care permite gruparea unor date de tipuri diferite sub un singur nume. Componentele unei structuri, denumite **câmpuri**, sunt identificate prin nume simbolice, denumite **selectori**. Câmpurile unei structuri pot fi de orice tip, simplu sau derivat, dar nu *void* sau funcție. Printr-o declarație *struct* se defineşte un nou tip de date de tip structură, de către programator.

IB.09.1.1 Declararea structurilor

Declararea structurilor se face folosind cuvântul cheie ***struct***; definirea unui tip structură are sintaxa următoare:

Sintaxă:
```
struct nume_structură {
        tip_câmp1 nume_câmp1;
        tip_câmp2 nume_câmp2;
        …
} [lista_variabile_structură];
```

unde:

- ***nume_structura*** este un nume de tip folosit numai precedat de cuvântul cheie *struct* (în C, dar în C++ se poate folosi singur ca nume de tip).
- ***tip_câmp1, tip_câmp2***,... este tipul componentei (câmpului) i
- ***nume_câmp1, nume_câmp2***,... este numele unei componente (câmp)

Exemple:

```
//structura numar complex
struct Complex {
  double real;
  double imag;
};
```

Ordinea enumerării câmpurilor unei structuri nu este importantă, deoarece ne referim la câmpuri prin numele lor. Se poate folosi o singura declarație de tip pentru mai multe câmpuri:

```
//structura moment de timp
struct time {
   int ora, min, sec;
};

//structura activitate
struct activ {
   char numeact[30];          // nume activitate
   struct time start;         // ora de incepere
   struct time stop;          // ora de terminare
};
```

Nume_structura **sau** lista_variabile_structura **pot lipsi, dar nu simultan. Dacă se precizează** nume_structura**, atunci înseamnă că se face definirea tipului** struct nume_structura**, care poate fi apoi folosit pentru declararea de variabile, ca tip de parametri formali sau ca tip de rezultat returnat de funcții.**
Declararea unor variabile de un tip structură se poate face fie după declararea tipului structură, fie simultan cu declararea tipului structură.
Nu există constante de tip structură, dar este posibilă inițializarea la declarare a unor variabile structură. Astfel de variabile inițializate şi cu atributul *const* ar putea fi folosite drept constante simbolice.

Exemple:

```
struct time t1,t2, t[100];    //t este vector de structuri

struct complex {
        float re,im;
} c1, c2, c3;                  //numere complexe

struct complex cv[200];  //un vector de numere complexe

struct coordonate{      //se declara tipul struct coordonate
    float x,y;
} punct, *ppunct, puncte[20]; //variabile
struct coordonate alt_punct = {1,4.5},
                  alte_puncte[10] = {{2,4},{8},{9,7}},
                  mai_multe_puncte[25] = {1,2,3,4};
//variabilele de tip structura se pot initializa la declarare;
//campurile neprecizate sunt implicit 0

struct persoana{             //se declara tipul struct persoana
     char nume[20];
     int varsta;
};                           //lipseste lista_declaratori
struct persoana pers={"Ion Ionescu",21}, *ppers, persoane[12];

struct{                     //lipseste nume_struct
     char titlu[20], autor[15], editura[12];
     int an_aparitie;
 }carte, biblioteca[1000];  /* nu se declara un tip, deci doar aici pot fi
                            facute declaratiile de variabile */
```

Un câmp al unei structuri poate fi de tip structură, dar nu aceeaşi cu cea definită - se poate însă să se declare un câmp pointer la structura definită (aspect care va fi utilizat la implementarea listelor):

```
struct persoana{                //se declara tipul struct persoana
     char nume[20];
     struct{
       int zi,an,luna

    }data_nasterii;            //camp de tip structura
 }p;
struct nod_lista{
    tip info info;
    struct nod_lista * urm;    //camp pointer la structura definita
};
```

Numele de structuri se află într-un spațiu de nume diferit de cel al numelor de variabile - se pot declara deci variabile şi tipuri structură cu acelaşi nume - de evitat însă. Se pot declara tipuri structuri care au nume de câmpuri identice.

De remarcat că orice declarație *struct* se termină obligatoriu cu caracterul ';' chiar dacă acest caracter urmează după o acoladă; aici acoladele nu delimitează un bloc de instrucțiuni ci fac parte din declarația *struct.*
În structuri diferite pot exista câmpuri cu acelaşi nume, dar într-o aceeaşi structură numele de câmpuri trebuie să fie diferite.

Accesul la câmpurile unei variabile de tip structură se face utilizând operatorul punct (.).
Exemplu:

```
struct complex c1;
...
c1.re

struct time t2;
...
t2.ora

struct time t[10];
...
t[0].min.
```

Atenție! Câmpurile unei variabile structură nu se pot folosi decât dacă numele câmpului este precedat de numele variabilei structură din care face parte, deoarece există un câmp cu acelaşi nume în toate variabilele de un acelaşi tip structură.
Exemplu:

```
int main () {
   complex c1,c2;
   scanf ("%f%f", &c1.re, &c1.im);              // citire c1
   c2.re= c1.re; c2.im= -c1.im;                 // complex conjugat
   printf ("(%f,%f) ", c2.re, c2.im);           // scrie c2
}
```

Dacă un câmp este la rândul lui o structură, atunci numele câmpului poate conține mai multe puncte ce separă numele variabilei şi câmpurilor de care aparține (în ordine ierarhică).
Exemplu:

```
//structura moment de timp
struct time {
   int ora, min, sec;
};

//structura activitate
struct activ {
   char numeact[30];          // nume activitate
   struct time start;         // ora de incepere
   struct time stop;          // ora de terminare
};
....
struct activ a;
printf ("%s începe la %d: %d şi se termina la %d: %d \n", a.numeact,
a.start.ora, a.start.min, a.stop.ora, a.stop.min);
```

IB.09.2. Asocierea de nume sinonime pentru tipuri structuri - typedef

Printr-o declarație *struct* se definește un nou tip de date de către programator. Utilizarea tipurilor structură pare diferită de utilizarea tipurilor predefinite, prin existența a două cuvinte care desemnează tipul (*struct* numestr). Declarația ***typedef*** din C permite atribuirea unui nume oricărui tip, nume care se poate folosi apoi la fel cu numele tipurilor predefinite ale limbajului. Sintaxa declarației *typedef* este la fel cu sintaxa unei declarații de variabilă, dar se declară un nume de tip şi nu un nume de variabilă.

În limbajul C declarația *typedef* se utilizează frecvent pentru atribuirea de nume unor tipuri structură.
Exemple:

```
// definire nume tip simultan cu definire tip structură
typedef struct {
   float re,im;
} complex;

// definire nume tip după definire tip structura
typedef struct activ act;
```

Deoarece un tip structură este folosit în mai multe funcții (inclusiv *main*), definirea tipului structură (cu sau fără *typedef*) se face la începutul fişierului sursă care conține funcțiile (înaintea primei funcții). Dacă un program este format din mai multe fişiere sursă atunci definiția structurii face parte dintr-un fişier antet (de tip ***.h***).

Se pot folosi ambele nume ale unui tip structură (cel precedat de *struct* şi cel dat prin *typedef*), care pot fi chiar identice.
Exemple:

```
typedef struct complex  {
   float re;
   float im;
} complex;

typedef struct point {
   double x,y;
} point;
...
struct point p[100];

// calcul arie triunghi dat prin coordonatele varfurilor
double arie ( point a, point b, point c) {
   return  a.x * (b.y-c.y) - b.x * (a.y-c.y) + c.x * (a.y-b.y);
}

typedef struct card
{
   int val;
   char cul[20];
} Carte;
....
Carte c1, c2;
c1. val = 10;
strcpy (c1. cul, "caro");
c2 = c1;
```

Cu *typedef* structura poate fi şi anonimă (poate lipsi cuvantul card din ultimul exemplu):

```
typedef struct
{
   int val;
   char cul[20];
} Carte;
```

Atunci când numele unui tip structură este folosit frecvent, inclusiv în parametri de funcții, este preferabil un nume introdus prin *typedef*, dar dacă vrem să punem în evidență că este vorba de tipuri structură vom folosi numele precedat de cuvântul cheie *struct*.
În C++ se admite folosirea numelui unui tip structură, fără a fi precedat de *struct* şi fără a mai fi necesar *typedef*.

IB.09.3. Utilizarea tipurilor structură

Un tip structură poate fi folosit în:

- declararea de variabile structuri sau pointeri la structuri
- declararea unor parametri formali de funcții (structuri sau pointeri la structuri)
- declararea unor funcții cu rezultat de un tip structură

Operațiile posibile cu variabile de un tip structură sunt:

- Selectarea unui câmp al unei variabile structură se realizează folosind operatorul de selecție **.** . Cîmpul selectat se comportă ca o variabilă de acelaşi tip cu câmpul, deci i se pot aplica aceleaşi prelucrări ca oricărei variabile de tipul respectiv.

```
variabila_structura.nume_camp
```

Exemplu:

```
struct persoana{              //se declara tipul struct persoana
    char nume[20];
    struct {
      int zi,an,luna
    } data_nasterii;          //camp de tip structura
} p;

//Selecția câmpurilor pentru variabila p de mai sus:
p.nume                                  //tablou de caractere
p.nume[0]       //primul caracter din nume
p.nume[strlen(p.nume)-1]    //ultimul caracter din nume
p.data_nasterii.an
p.data_nasterii.luna
p.data_nasterii.an
```

- O variabilă structură poate fi inițializată la declarare prin precizarea între {} a valorilor câmpurilor; cele neprecizate sunt implicit 0.

- O variabilă structură poate fi copiată în altă variabilă de acelaşi tip.
  Exemplu cu declarațiile de mai sus:

```
printf("%d %d\n", sizeof(pers), sizeof(struct persoana));
ppers = &pers;
persoane[0] = *ppers;      //atribuirea intre doua variabile structura
```

- O variabilă structură nu poate fi citită sau scrisă direct, ci prin intermediul câmpurilor!!
  Se pot aplica operatorii:
  & - referință
  sizeof - dimensiune

- **->**. Deoarece structurile se prelucrează frecvent prin intermediul pointerilor, a fost introdus operatorul `->` care combină operatorul de indirectare '`*`' cu cel de selectie'**.**'. Cele două expresii de mai jos sunt echivalente:

```
(*variabila_pointer).nume_camp ⇔ variabila_pointer->nume_camp
```

- Transmiterea ca argument efectiv la apelarea unei funcții;
- Transmiterea ca rezultat al unei funcții, într-o instrucțiune *return.*

Singurul operator al limbajului C care admite operanzi de tip structura este cel de atribuire. Pentru alte operații cu structuri trebuie definite funcții: comparații, operații aritmetice, operații de citire-scriere etc.

Exemplul următor arată cum se poate ordona un vector de structuri time:

```
struct time {                    //structura moment de timp
   int ora, min, sec;
};

void wrtime ( struct time t) {// scrie ora, min, sec
  printf ("%02d:%02d:%02d \n", t.ora,t.min,t.sec);
}

int cmptime (struct time t1, struct time t2) { // comparare
  int d;
  d=t1.ora - t2.ora;
  if (d) return d;
  d=t1.min - t2.min;               // <0 daca t1<t2 şi >0 daca t1>t2
  if (d) return d;                 // rezultat negativ sau pozitiv
  return t1.sec - t2.sec;          // rezultat <0 sau =0 sau > 0
}

// utilizare funcții
int main () {
  struct time tab[200], aux;
  int i, j, n;
  . . .    // citire date
  // ordonare vector
  for (j=1;j<n;j++)
    for (i=1;i<n;i++)
      if ( cmptime (tab[i-1],tab[i]) > 0) {
        aux=tab[i-1];
        tab[i-1]=tab[i];
        tab[i]=aux;
    }
  // afişare vector ordonat
  for (i=0;i<n;i++)
    wrtime(tab[i]);
}
```

Principalele avantaje ale utilizării unor tipuri structură sunt:

- Programele devin mai explicite dacă se folosesc structuri în locul unor variabile separate.
- Se pot defini tipuri de date specifice aplicației iar programul reflectă mai bine universul aplicației.
- Se poate reduce numărul de parametri al unor funcții prin gruparea lor în parametri de tipuri structură şi deci se simplifică utilizarea acelor funcții.
- Se pot utiliza structuri de date extensibile, formate din variabile structură alocate dinamic şi legate între ele prin pointeri (liste înlănțuite, arbori ş.a).

IB.09.4. Funcții cu parametri şi/sau rezultat structură

O funcție care produce un rezultat de tip structură poate fi scrisă în două moduri, care implică şi utilizări diferite ale funcției:

- funcția are rezultat de tip structură:

```
// citire numar complex (varianta 1)
complex readx () {
  complex c;
  scanf ("%f%f",&c.re, &c.im);
  return c;
}
```

```
// utilizare
complex a[100];
for (i=0;i<n;i++)
   a[i]=readx();
```

- funcția este de tip *void* şi depune rezultatul la adresa primită ca parametru (pointer la tip structură):

```
// citire numar complex (varianta 2)
void readx ( complex * px) {        // px pointer la o structură complex
   scanf ("%f%f", &px->re, &px->im);
}

// utilizare
complex a[100];
......
for (i=0;i<n;i++)
   readx (&a[i]);                   // adresa variabilei structură a[i]
```

Reamintim că notația px->re este echivalentă cu notația (*px).re şi se interpretează astfel: "câmpul *re* al structurii de la adresa *px*".

Uneori, mai multe variabile descriu împreună un anumit obiect şi trebuie transmise la funcțiile ce lucrează cu obiecte de tipul respectiv. Gruparea acestor variabile într-o structură va reduce numărul de parametri şi va simplifica apelarea funcțiilor. Exemple de obiecte definite prin mai multe variabile: obiecte geometrice (puncte, poligoane ş.a), date calendaristice şi momente de timp, structuri de date (stiva, coada, ş.a), vectori, matrice, etc.
Exemplu de grupare într-o structură a adresei şi dimensiunii unui vector:

```
typedef struct {
   int vec[1000];
   int dim;
} vector;

// afişare vector
void scrvec (vector v) {
   int i;
   for (i=0;i<v.dim;i++)
      printf ("%d  ",v.vec[i]);
   printf ("\n");
}

// elementele comune din 2 vectori
vector comun (vector a, vector b) {
   vector c;
   int i,j,k=0;
   for (i=0;i<a.dim;i++)
      for (j=0;j<b.dim;j++)
         if (a.vec[i]==b.vec[j])
            c.vec[k++]=a.vec[i];
   c.dim=k;
   return c;
}
```

Pentru structurile care ocupă un număr mare de octeți este mai eficient să se transmită ca parametru la funcții adresa structurii (un pointer) în loc să se copieze conținutul structurii la fiecare apel de funcție şi să se ocupe loc în stivă, chiar dacă funcția nu face nici o modificare în structura a cărei adresă o primeşte.
Funcțiile cu parametri pointeri la structuri pot produce efecte secundare (laterale) nedorite, prin modificarea involuntară a unor variabile din alte funcții.

În concluzie, avantajele utilizării tipurilor structură sunt următoarele:
Programele devin mai explicite dacă se folosesc structuri în locul unor variabile separate.

- Se pot defini tipuri de date specifice aplicației iar programul reflectă mai bine universul aplicației.
- Se poate reduce numărul de parametri al unor funcții prin gruparea lor în parametri de tipuri structură şi deci se simplifică utilizarea acelor funcții.
- Se pot utiliza structuri de date extensibile, formate din variabile structură alocate dinamic şi legate între ele prin pointeri (liste înlănțuite, arbori s.a).

Exemple

1.Să se definească o structură Point pentru un punct geometric 2D şi o structură Rectangle pentru un dreptunghi definit prin colțul stânga sus şi colțul dreapta jos. Să se inițializeze şi să se afişeze o variabilă de tip Rectangle.

```
#include <stdio.h>

typedef struct Point {
  int x, y;
} Point;

typedef struct Rectangle {
  Point topLeft;
  Point bottomRight;
} Rectangle;

int main() {
 Point p1, p2;
 p1.x = 0;  // p1 la (0, 3)
 p1.y = 3;
 p2.x = 4;  // p2 la (4, 0)
 p2.y = 0;
 printf( "p1 la ( %d, %d)\n", p1.x, p1.y);
 printf( "p2 la ( %d, %d)\n", p2.x, p2.y);

 Rectangle rect;
 rect.topLeft = p1;
 rect.bottomRight = p2;
 printf("Stanga sus la(%d,%d)\n", rect.topLeft.x,rect.topLeft.y);
 printf("Dreapta jos la (%d,%d)\n", rect.bottomRight.x,rect.bottomRight.y);
 return 0;
}
```

Rezultatul va fi:

```
p1 la (0,3)
p2 la (4,0)
Stanga sus la (0,3)
Dreapta jos la (4,0)
```

2. Să se defineasca o structura “time" care grupează 3 întregi ce reprezintă ora, minutul şi secunda pentru un moment de timp. Să se scrie funcții pentru:

- Verificare corectitudine ora
- Citire moment de timp
- Scriere moment de timp
- Comparare de structuri “time".

Program pentru citirea şi ordonarea cronologică a unor momente de timp şi afişarea listei ordonate, folosind funcțiile anterioare.

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

```
typedef struct time {
    int ora, min, sec;
}time;

void wrtime ( time t) {       // scrie ora, min, sec
      printf ("%02d:%02d:%02d \n", t.ora,t.min,t.sec);
}

int cmptime (time t1, time t2) {    // compara momente de timp
  int d;

  d=t1.ora - t2.ora;
  if (d) return d;
  d=t1.min - t2.min;                // <0 daca t1<t2 şi >0 daca t1>t2
  if (d) return d;                  // rezultat negativ sau pozitiv
  return t1.sec - t2.sec;           // rezultat <0 sau =0 sau > 0
}

int corect (time t) {               // verifica daca timp plauzibil
      if ( t.ora < 0 || t.ora > 23 ) return 0;
  if ( t.min < 0 || t.min > 59 ) return 0;
  if ( t.sec < 0 || t.sec > 59 ) return 0;
  return 1;                         // plauzibil corect
}

time rdtime () {                    // citire ora
  time t;

  do {
           scanf ("%d%d%d", &t.ora, &t.min,&t.sec);
           if ( ! corect (t))
             printf ("Date gresite, repetati introducerea: \n");
           else break;
  }while(1);

  return t;
}

void sort (time a[], int n) {       // ordonare vector de date
      int i,gata;
       time aux;

      do {
         gata=1;
         for (i=0;i<n-1;i++)
             if (cmptime(a[i],a[i+1]) > 0 ) {
                  aux=a[i];
          a[i]=a[i+1];
          a[i+1]=aux;
          gata=0;
                  }
       }while (!gata);

}

int main () {
        time t[30];
        int n,i;
        printf("introducere n: ");
    scanf("%d",&n);
        for(i=0;i<n;i++) t[i] = rdtime();
        sort (t, n);
        for ( i=0 ;i<n ;i++) wrtime(t[i]);
        return 0;
```

```
}
```

IB.09.5. Structuri predefinite

În bibliotecile C standard există unele structuri predefinite. Un astfel de exemplu este structura *struct tm* definită în biblioteca *time*; această structură conține componente ce definesc complet un moment de timp:

```
struct tm {
  int     tm_sec, tm_min, tm_hour;          // secunda, minut, ora
  int     tm_mday,  tm_mon,  tm_year;       // zi, luna, an
  int     tm_wday, tm_yday;                 // nr zi în saptamana şi în an
  int     tm_isdst;                         // 1 - se modifica ora (iarna/vara), 0 - nu
};
```

Există funcții care lucrează cu această structură: *asctime*, *localtime*, etc.

Exemplu: cum se poate afişa ora şi ziua curentă, folosind numai funcții standard

```
#include <stdio.h>
#include <time.h>
int main(void) {
  time_t t;                   // time_t  este alt nume pentru  long
  struct tm *area;            // pentru rezultat funcţie localtime
  t = time (NULL);            // obtine ora curenta
  area = localtime(&t);       // conversie din time_t în struct tm
  printf ("Local time is: %s", asctime(area));
}
```

Observații:

- *asctime(struct tm* t)* returnează un şir ce reprezintă ziua şi ora din strucura t. Şirul are următorul format: DDD MMM dd hh:mm:ss YYYY
- funcția *time* transmite rezultatul şi prin numele funcției şi prin parametru: *long time (long*)*, deci se putea apela şi: *time (&t);*

O altă structură predefinită este *struct stat* definită în fişierul *sys/stat.h. Structura* reuneşte date despre un fişier, cu excepția numelui:

```
struct stat {
  short unix [7];              // fără semnificatie în sisteme Windows
  long st_size;                 // dimensiune fisier (octeti)
  long st_atime, st_mtime;      // ultimul acces / ultima modificare
  long st_ctime;               // data de creare
};
```

Funcția *stat* completează o astfel de structură pentru un fişier cu nume dat:

```
int stat (char* filename,  struct stat * p);
```

Exemplu:
Pentru a afla dimensiunea unui fişier normal (care nu este fişier director) vom putea folosi funcția următoare:

```
long filesize (char * filename) {
   struct stat fileattr;
   if (stat (filename, &fileattr) < 0)          // daca fisier negasit
      return -1;
```

```
   else     // fisier gasit
      return (fileattr.st_size);     // campul st_size contine lungimea
}
```

IB.09.6. Structuri cu conținut variabil (uniuni)

Uniunea (reuniunea – union) defineşte un grup de variabile care nu se memorează simultan ci alternativ.
În felul acesta se pot memora diverse tipuri de date la o aceeaşi adresă de memorie.
Cuvântul cheie ***union*** se foloseşte la fel cu *struct*. Alocarea de memorie se face (de către compilator) în funcție de variabila ce necesită maxim de memorie.

Declararea uniunilor se face folosind cuvântul cheie ***union:***

Sintaxă:
union {
tip_comp1 nume_comp1;
tip_comp2 nume_comp2;
…
} [lista_variabile_structură];

unde:

- ***tip_comp1, tip_comp2***,... este tipul componentei i
- ***nume_comp1, nume_comp2***,... este numele unei componente

Exemplu:

```
union {
     int ival;
    long lval;
    float fval;
    double dval;
} val;
```

O uniune face parte de obicei dintr-o structură care mai conține şi un *câmp discriminant*, care specifică tipul datelor memorate (alternativa selectată la un moment dat).
Exemplul următor arată cum se poate lucra cu numere de diferite tipuri şi lungimi, reunite într-un tip generic:

```
typedef struct numar {
  char tipn;      // tip numar (caracter: i-int, l-long, f-float, d-double
  union {
      int ival;
     long lval;
     float fval;
     double dval;
  } val;                       // valoare numar
} Numar;                       // definire tip de date Numar

void write (Numar n) {         // in functie de tip afiseaza valoarea
  switch (n.tipn) {
   case 'i': printf ("%d ", n.val.ival); break;
   case 'l': printf ("%ld ", n.val.lval); break;
   case 'f': printf ("%f ", n.val.fval); break;
   case 'd': printf ("%.15lf ", n.val.dval);
  }
}
```

Observație:

În locul construcției *union* se poate folosi o variabilă de tip *void** care va conține adresa unui număr, indiferent de tipul lui. Memoria pentru număr se va aloca dinamic:

```
typedef struct number {
   char tipn;            // tip numar
   void * pv;            // adresa numar
}number;

// in functie de tip afiseaza valoarea
void write (number n) {
   switch (n.tipn) {
            case 'i': printf("%d",*(int*)n.pv);
                      //conversie la int* si indirectare
               break;
      ...
      case 'd': printf("%.15lf",*(double*)n.pv); /*conversie la double *
                                                   si indirectare*/
               break;
   }
}
```

IB.09.7. Enumerări

Tipul enumerare este un caz particular al tipurilor întregi. Se utilizează pentru a realiza o reprezentare comodă şi sugestivă a unor obiecte ale caror valori sunt identificate printr-un număr finit de nume simbolice.
Tipul enumerare declară **constante simbolice**, cărora li se **asociază coduri numerice de tip întreg**.
Compilatorul asociaza constantelor enumerate câte un cod întreg din succesiunea începând cu 0. Implicit, şirul valorilor e crescător cu pasul 1.
Un nume de constantă nu poate fi folosit în mai multe enumerări.

Sintaxa este următoarea:

Sintaxă:
enum [nume_tip] { lista_constante } [lista_variabile] ;

unde *nume_tip* şi *lista de variabile* pot lipsi!

Exemple:

```
enum zile_lucr { luni, marti, miercuri, joi, vineri };
//   luni   e   asociat   cu   0,   marti   cu   1,   ...,   vineri   cu   4

// se pot defini variabile de tipul zile_lucr, ca mai jos, variabila zi:
enum zile_lucr zi=marti;

enum Color {
   red, green, blue
} myColor;      // Defineste o enumerare şi declară o variabilă de tipul ei
......
myColor = red;          // Atribuie o valoare variabilei
Color yourColor;        // Declară o variabilă de tipul Color
yourColor = green;      // Atribuie o valoare variabilei
```

Dacă se dorește o altă codificare a constantelor din enumerare decât cea implicită, pot fi folosite în enumerare elemente de forma:

nume_constantă = valoare_întreagă;

Constantelor simbolice ce urmează unei astfel de inițializări li se asociază numerele întregi următoare:

```
enum transport { tren, autocar=5, autoturism, avion };
/* tren e asociat cu 0, autocar cu 5, autoturism cu 6, avion cu 7 */
```

```
enum luni_curs {ian=1, feb, mar, apr, mai, iun, oct=10, nov, dec};
```

După cum spuneam, tipurile enumerare sunt tipuri întregi, variabilele enumerare se pot folosi la fel ca variabilele întregi, asigurând un cod mai lizibil decât prin declararea separată de constante.

Putem folosi declarațiile de tip typedef pentru a introduce un tip enumerare:

```
enum {D, L, Ma, Mc, J, V, S} zi;
/* tip anonim; declară doar variabila zi, tipul nu are nume, deci nu mai
putem declara altundeva variabile */

// sau:

typedef enum {D, L, Ma, Mc, J, V, S} Zi;
// tip enumerare cu numele Zi; se pot declara variabile

int nr_ore_lucru[7];          // vector cu număr de ore pe zi
for (zi = L; zi <= V; ++zi) nr_ore_lucru[zi] = 8;


typedef    enum    {    inginer=1,    profesor,    avocat    }    profesie;
profesie profesia_mea=inginer; // definirea variabilei profesia_mea
```

IB.09.8. Exemple

1. Să se scrie un program care declară o strucură pentru un număr generic, folosind o uniune pentru valoarea numărului şi un câmp tip ce va spune ce fel de număr este (întreg – i, întreg lung – l, real – f, real dubla precizie – d).
Să se scrie o funcție ce citeşte o variabilă de tipul structurii definite.
Să se scrie o funcție ce afişează valoarea unei variabile de tipul structurii definite.

Rezolvare:

```
#include<stdio.h>

typedef struct numar {      // structura pentru un număr de orice tip
  char tipn;                // tip numar (caracter: i-int, l-long, f-float,
d-double
  union {
      int ival;
     long lval;
     float fval;
     double dval;
  } val;                     // valoare numar
} Numar;                     // definire tip de date Numar

// afisare număr
void write (Numar n) {
  switch (n.tipn) {                // in functie de tip afiseaza valoarea
   case 'i': printf ("Intreg %d ", n.val.ival); break;
   case 'l': printf ("Intreg lung %ld ", n.val.lval); break;
   case 'f': printf ("Real %f ", n.val.fval); break;
   case 'd': printf ("Real dublu %.15lf ", n.val.dval);
  }
}

// citire număr
Numar read (char tip) {
  Numar n;
  n.tipn=tip;
```

```
  switch (tip) {
   case 'i': scanf ("%d", &n.val.ival); break;
   case 'l': scanf ("%ld", &n.val.lval); break;
   case 'f': scanf ("%f", &n.val.fval); break;
   case 'd': scanf ("%lf", &n.val.dval);
   }
  return n;
}

int main () {
  Numar a, b, c, d;
  a = read('i');
   b = read('l');
   c = read('f');
   d = read('d');
  write(a);
   write(b);
   write(c);
   write(d);
       return 0;
}
```

2. Pentru câmpul discriminant se poate defini un tip enumerare, împreună cu valorile constante (simbolice) pe care le poate avea. Să se refacă programul!

Rezolvare:

```
#include<stdio.h>
enum tnum {I, L, F, D};       // definire tip "tnum"
typedef struct{
  tnum tipn;                  // tip număr (un caracter)
  union {
      int ival;
     long lval;
     float fval;
     double dval;
  } val;                      // valoare numar
} Numar;                      // definire tip de date Numar

void write (Numar n) {
  switch (n.tipn) {
   case I: printf ("%d", n.val.ival); break;          // int
   case L: printf ("%ld", n.val.lval); break;         // long
   case F: printf ("%f", n.val.fval);break;           // float
   case D: printf ("%.15lf", n.val.dval);             // double
  }
}

Numar read (tnum tip) {
  Numar n;
  n.tipn=tip;
  switch (tip) {
       case I: scanf ("%d", &n.val.ival); break;
        ...
  }
  return n;
}

int main () {
      Numar a,b,c,d;
      a = read(I);
      write(a);
      ...
}
```

Capitolul IB.10. Alocarea memoriei în limbajul C

Cuvinte cheie
Clase de memorare, alocare statică, alocare dinamică, variabile auto, variabile locale, variabile globale, variabile register, funcții standard, vectori de pointeri, srucuturi alocate dinamic

IB.10.1. Clase de memorare (alocare a memoriei) în C

Clasa de memorare arată când, cum şi unde se alocă memorie pentru o variabilă.
Orice variabilă are o clasă de memorare care rezultă fie din declarația ei, fie implicit din locul unde este definită variabila.

Zona de memorie utilizată de un program C cuprinde 4 subzone:
- Zona text: în care este păstrat codul programului
- Zona de date: în care sunt alocate (păstrate) variabilele globale
- Zona stivă: în care sunt alocate datele temporare (variabilele locale)
- Zona heap: în care se fac alocările dinamice de memorie

Moduri de alocare a memoriei:

- **Statică:** variabile implementate în zona de date - globale

Memoria este alocată la compilare în segmentul de date din cadrul programului şi nu se mai poate modifica în cursul execuției. Variabilele externe, definite în afara funcțiilor, sunt implicit statice, dar pot fi declarate *static* şi variabile locale, definite în cadrul funcțiilor.

- **Auto**: variabile implementate în stivă - locale

Memoria este alocată automat, la activarea unei funcții, în zona stivă alocată unui program şi este eliberată automat la terminarea funcției. Variabilele locale unui bloc (unei funcții) şi parametrii formali sunt implicit din clasa *auto*. Memoria se alocă în stiva ataşată programului.

- **Dinamică**: variabile implementate în heap

Memoria se alocă dinamic (la execuție) în zona *heap* ataşată programului, dar numai la cererea explicită a programatorului, prin apelarea unor funcții de bibliotecă (*malloc, calloc, realloc*). Memoria este eliberată numai la cerere, prin apelarea funcției *free*

- **Register**: variabile implementate într-un registru de memorie

IB.10.2. Clase de alocare a memoriei: Auto

Variabilele locale unui bloc (unei funcții) şi parametrii formali sunt implicit din clasa *auto*.
Durata de viață a acestor variabile este temporară: memoria este alocată automat, la activarea blocului/funcției, în zona stivă alocată programului şi este eliberată automat la ieşirea din bloc/terminarea funcției. Variabilele locale NU sunt inițializate! Trebuie să le atribuim o valoare inițială!
Exemplu:

```
int doi() {
      int x = 2;
      return x;
}
int main() {
      int a;
      {
            int b = 5;
            a = b*doi();
      }
      printf("a = %d\n", a);
      return 0;
}
```

Conținut stivă:

(x) 2

(b) 5

(a) 10

IB.10.3. Clase de alocare a memoriei: Static

Memoria este alocată la compilare în segmentul de date din cadrul programului şi nu se mai poate modifica în cursul execuției.
Variabilele globale sunt implicit *statice* (din clasa *static*).
Pot fi declarate *static* şi variabile locale, definite în cadrul funcțiilor, folosind cuvântul cheie static.
O variabilă sau o funcție declarată (sau implicit) *static* are durata de viață egală cu cea a programului. In consecință, o variabilă *statică* declarată într-o funcție îşi păstrează valoarea între apeluri succesive ale funcției, spre deosebire de variabilele *auto* care sunt realocate pe stivă la fiecare apel al funcției şi pornesc de fiecare dată cu valoarea primită la inițializarea lor (sau cu o valoare imprevizibilă, dacă nu sunt inițializate).
Exemple:

```
int f1() {
   int x = 1;           /*Variabilă locală, inițializată cu 1 la fiecare
                  apel al lui f1*/
   ......
}

int f2() {
   static int y = 99; /*Variabilă locală statică, inițializată cu 99
                  doar  la primul apel al lui f2; valoarea ei este
                  reținută pe parcursul apelurilor lui f2*/
   ......
}

int f() {
      static int nr_apeluri=0;
      nr_apeluri++;
      printf("funcția f() este apelata pentru a %d-a oara\n", nr_apeluri);
      return nr_apeluri;
}
int main() {
      int i;
      for (i=0; i<10; i++) f();   //f() apelata de 10 ori
      printf("functia f() a fost apelata de %d ori.", f()); // 11 ori!!
      return 0;
}
```

Observații:
Variabilele locale statice se folosesc foarte rar în practica programării (funcția de bibliotecă *strtok* este un exemplu de funcție cu o variabilă statică).

- Variabilele statice pot fi inițializate numai cu valori constante (pentru că inițializarea are loc la compilare), dar variabilele *auto* pot fi inițializate cu rezultatul unor expresii (pentru că inițializarea are loc la execuție).
  Exemplu de funcție care afişează un întreg pozitiv în cod binar, folosind câturile împărțirii cu puteri descrescătoare ale lui 10:

```
// afisare intreg in binar
void binar ( int x) {
```

```
    int n=digits(x);    //functie care intoarce nr-ul de cifre al lui x
    int d=pw10 (n-1);   //functie care calculeaza 10 la o putere intreaga
    while ( x >0) {
       printf("%d",x/d); //scrie catul impartirii lui x prin d
       x=x%d;
       d=d/10;            //continua cu x = x%d si d = d/10
    }
}
```

- Toate variabilele externe (şi statice) sunt automat inițializate cu valori zero (inclusiv vectorii).

Cuvântul cheie *static* face ca o variabilă globală sau o funcție să fie *privată(proprie)* unității unde a fost definită: ea devine inaccesibilă altei unități, chiar prin folosirea lui *extern*.

- Cantitatea de memorie alocată pentru variabilele cu nume rezultă din tipul variabilei şi din dimensiunea declarată pentru vectori. Memoria alocată dinamic este specificată explicit ca parametru al funcțiilor de alocare, în număr de octeți.

Memoria neocupată de datele statice şi de instrucțiunile unui program este împărțită între *stivă* şi *heap*.

Consumul de memorie *stack* (*stiva*) este mai mare în programele cu funcții recursive (număr mare de apeluri recursive).

Consumul de memorie *heap* este mare în programele cu vectori şi matrice alocate (şi realocate) dinamic.

De observat că nu orice vector cu dimensiune constantă este un vector *static*; un vector definit într-o funcție (alta decât *main*) nu este static deoarece nu ocupă memorie pe toată durata de execuție a programului, deşi dimensiunea sa este stabilită la scrierea programului. Un vector definit într-o funcție este alocat pe stivă, la activarea funcției, iar memoria ocupată de vector este eliberată automat la terminarea funcției.

Sinteză variabile locale / variabile globale

O sinteză legată de variabilele locale şi cele globale din punct de vedere al duratai de viață vs. domeniu de vizibilitate este dată în tabelul următor:

	Variabile globale	Variabile locale
Alocare	Statică; la compilare	Auto; la execuție bloc
Durata de viață	Cea a întregului program	Cea a blocului în care e declarată
Inițializare	Cu zero	Nu se face automat

IB.10.4. Clase de alocare a memoriei: Register

A treia clasă de memorare este clasa *register,* pentru variabile cărora li se alocă registre ale procesorului şi nu locații de memorie, pentru un timp de acces mai bun.

O variabilă declarată *register* solicită sistemului alocarea ei într-un registru maşină, dacă este posibil.

De obicei compilatorul ia automat decizia de alocare a registrelor maşinii pentru anumite variabile *auto* din funcții. Se utilizează pentru variabile "foarte solicitate", pentru mărirea vitezei de execuție.

Exemplu:

```
{
register int i;
for(i = 0; i < N; ++i){
/*… */
}
```

```
} /* se elibereaza registrul */
```

IB.10.5. Clase de alocare a memoriei: `extern`

O variabilă externă este o variabilă definită în alt fişier. Declaraţia extern îi spune compilatorului că identificatorul este definit în alt fişier sursă (extern). Ea este este alocată în funcţie de modul de declarare din fişierul sursă.
Exemplu:

```
// File1.cpp
extern int i;  // Declara aceasta variabila ca fiind definita in alt fisier

// File2.cpp
int i = 88;    // Definit aici
```

IB.10.6. Alocarea dinamică a memoriei

Reamintim că pentru variabilele alocate dinamic memoria se alocă dinamic (la execuţie) în zona *heap* ataşată programului, dar numai la cererea explicită a programatorului, prin apelarea unor funcţii de bibliotecă (*malloc, calloc, realloc*). Memoria este eliberată numai la cerere, prin apelarea funcţiei *free.*
Principalele diferenţe între alocarea statică şi cea dinamică sunt:

- La alocarea statică, compilatorul alocă şi eliberează memoria automat, ocupându-se astfel de gestiunea memoriei, în timp ce la alocarea dinamică programatorul este cel care gestionează memoria, având un control deplin asupra adreselor de memorie şi a conţinutului lor.
- Entităţile alocate static sau auto sunt manipulate prin intermediul unor variabile, în timp ce cele alocate **dinamic** sunt gestionate prin intermediul **pointerilor**!

IB.10.6. 1 Funcţii standard pentru alocarea dinamică a memoriei

Funcţiile standard pentru alocarea dinamica a memoriei sunt declarate în fişierele *stdlib.h* şi *alloc.h*.

Alocarea memoriei:

void *malloc(size_t size);

Alocă memorie de dimensiunea *size* octeţi

void *calloc(int nitems, size_t size);

Alocă memorie pentru *nitems* de dimensiune *size* octeţi şi iniţializează zona alocată cu zerouri

Cele două funcţii au ca rezultat adresa zonei de memorie alocate (de tip *void*.
Dacă cererea de alocare nu poate fi satisfăcută, pentru că nu mai exista un bloc continuu de dimensiunea solicitată, atunci funcţiile de alocare au rezultat *NULL*. Funcţiile de alocare au rezultat *void** deoarece funcţia nu ştie tipul datelor ce vor fi memorate la adresa respectivă.
La apelarea funcţiilor de alocare se folosesc:

Operatorul *sizeof* pentru a determina numărul de octeţi necesar unui tip de date (variabile);

Operatorul de conversie *cast* pentru adaptarea adresei primite de la funcţie la tipul datelor memorate la adresa respectivă (conversie necesară atribuirii între pointeri de tipuri diferite).

Exemple:

```
//aloca memorie pentru 30 de caractere:
```

```
char * str = (char*) malloc(30);

//aloca memorie ptr. n întregi:
int * a = (int *) malloc( n * sizeof(int));

//aloca memorie ptr. n întregi si initializeaza cu zerouri
int * a= (int*) calloc (n, sizeof(int) );
```

IB.10.6. 2 Realocarea memoriei

Realocarea unui vector care creşte (sau scade) faţă de dimensiunea estimată anterior se poate face cu funcţia *realloc*, care primeşte adresa veche şi noua dimensiune şi întoarce noua adresă:

```
void *realloc(void* adr, size_t size);
```

Funcţia *realloc* realizează următoarele operaţii:

- Alocă o zonă de dimensiunea specificată prin al doilea parametru.
- Copiază la noua adresă datele de la adresa veche (primul parametru).
- Eliberează memoria de la adresa veche.

Exemple:

```
// dublare dimensiune curenta a zonei de la adr. a
a = (int *)realloc (a, 2*n* sizeof(int));
```

Atenţie! Se va evita redimensionarea unui vector cu o valoare foarte mică de un număr mare de ori; o strategie de realocare folosită pentru vectori este dublarea capacităţii lor anterioare.

Exemplu de funcţie cu efectul funcţiei *realloc*, dar doar pentru caractere:

```
char * ralloc (char * p, int size) {      // p = adresa veche
   char *q;                               // q=adresa noua

   if (size==0) {                         // echivalent cu free
      free(p);
      return NULL;
   }
   q = (char*) malloc(size);              // aloca memorie
   if (q) {                               // daca alocare reusita
        memcpy(q,p,size);                 // copiere date de la p la q
        free(p);                          // elibereaza adresa p
   }
   return q;                              // q poate fi NULL
}
```

Observaţie: La mărirea blocului, conţinutul zonei alocate în plus nu este precizat, iar la micşorarea blocului se pierd datele din zona la care se renunţă.

IB.10.6. 3 Eliberarea memoriei

Funcţia *free* are ca argument o adresă (un pointer) şi eliberează zona de la adresa respectivă (alocată dinamic). Dimensiunea zonei nu mai trebuie specificată deoarece este memorată la începutul zonei alocate (de către funcţia de alocare):

```
void free(void* adr);
```

Eliberarea memoriei prin *free* este inutilă la terminarea unui program, deoarece înainte de încărcarea şi lansarea în execuţie a unui nou program se eliberează automat toată memoria *heap*.

Exemple:

```
char *str;

str=(char *)malloc(10*sizeof(char));
```

```
…
str=(char *)realloc(str,20*sizeof(char));
…
free(str);
```

Observație:

Atenție la definirea de şiruri în mod dinamic! Şirul respectiv trebuie inițializat cu adresa unui alt şir sau a unui spațiu alocat pe heap (adică alocat dinamic)!

Exemple:

```
char *sir3;
char şir1[30];

// Varianta 1: sir3 ia adresa unui şir static
sir3 = sir1;        // Echivalent cu: sir3=&sir1; ⇔ sir3=&sir1[0];
char *sir4="test";  //sir4 este inițializat cu adresa unui şir constant

// Varianta 2: se alocă dinamic un spațiu pe heap
sir3=(char *)malloc(100*sizeof(char));
```

Exemplu

Program care alocă spațiu pentru o variabilă întreagă dinamică, după citire şi tipărire, spațiul fiind eliberat. Modificați programul astfel încât variabila dinamică să fie de tip double.

Rezolvare

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

int main(){

int *pi;
pi=(int *)malloc(sizeof(int));
if(pi==NULL){
puts("*** Memorie insuficienta ***");
return 1;      // revenire din main
}

printf("valoare:");
```

```
   //citirea variabilei dinamice, de pe heap, de la adresa din pi!!!
   scanf("%d",pi);
   *pi=*pi*2;      // dublarea valorii
   printf("val=%d,pi(adresa pe heap)=%p,adr_pi=%p\n", *pi, pi, &pi);


   // sizeof aplicat unor expresii:
   printf("%d %d %d\n",sizeof(*pi), sizeof(pi), sizeof(&pi));


   free(pi);      //eliberare spatiu
   printf("pi(dupa elib):%p\n",pi); // nemodificat, dar invalid!
   return 0;
   }
```

IB.10.7. Vectori alocați dinamic

Structura de vector are avantajul simplității şi economiei de memorie față de alte structuri de date folosite pentru memorarea unei colecții de date.
Dezavantajul unui vector cu dimensiune fixă (stabilită la declararea vectorului şi care nu mai poate fi modificată la execuție) apare în aplicațiile cu vectori de dimensiuni foarte variabile, în care este dificil de estimat o dimensiune maximă, fără a face risipă de memorie.
De cele mai multe ori programele pot afla din datele citite dimensiunile vectorilor cu care lucrează şi deci pot face o alocare dinamică a memoriei pentru aceşti vectori. Aceasta este o soluție mai flexibilă, care foloseşte mai bine memoria disponibilă şi nu impune limitări arbitrare asupra utilizării unor programe.
În limbajul C nu există practic nici o diferență între utilizarea unui vector cu dimensiune fixă şi utilizarea unui vector alocat dinamic, ceea ce încurajează si mai mult utilizarea unor vectori cu dimensiune variabilă.
Un vector alocat dinamic se declară ca variabilă pointer care se inițializează cu rezultatul funcției de alocare. Tipul variabilei pointer este determinat de tipul componentelor vectorului.

Exemplu:

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

int main() {
   int n, i;
   int * a;
      // adresa vector alocat dinamic

   printf ("n=");
   scanf ("%d", &n);                        // dimensiune vector
   a=(int *) calloc (n,sizeof(int));        // aloca memorie pentru vector
          // sau: a=(int*) malloc (n*sizeof(int));

   // citire component vector:
   printf ("componente vector: \n");
   for (i=0;i<n;i++)
     scanf ("%d", &a[i]);                   // sau   scanf ("%d", a+i);
   // afisare vector:
   for (i=0;i<n;i++)
     printf ("%d ",a[i]);
   return 0;
}
```

Există şi cazuri în care datele memorate într-un vector rezultă din anumite prelucrări, iar numărul lor nu poate fi cunoscut de la începutul execuției. În acest caz se poate recurge la o realocare dinamică a memoriei. O strategie de realocare pentru vectori este dublarea capacității lor anterioare.

În exemplul următor se citeşte un număr necunoscut de valori întregi într-un vector extensibil:

Program care citeşte numere reale până la CTRL+Z, le memorează într-un vector alocat şi realocat dinamic în funcție de necesități şi le afişează.

Rezolvare:

```
#include <stdio.h>
#include <stdlib.h>
#define INCR 4

int main() {
int n,n_crt,i ;
float x, * v;
n = INCR;           // dimensiune memorie alocata
n_crt = 0;          // numar curent elemente în vector
v = (float *)malloc (n*sizeof(float));    //alocare initiala

while (scanf("%f",&x) !=EOF){
if (n_crt == n) {
n = n + INCR;
v = (float *) realloc (v, n*sizeof(float) );       //realocare
}
v[n_crt++] = x;
}
for (i=0; i<n_crt; i++)
printf ("%.2f ", v[i]);
return 0;
}
```

Din exemplele anterioare lipseşte eliberarea memoriei alocate pentru vectori, dar fiind vorba de un singur vector alocat în funcția *main* şi necesar pe toată durata de execuție, o eliberare finală este inutilă. Eliberarea explicită poate fi necesară pentru vectori de lucru, alocați dinamic în funcții.

IB.10.8. Matrice alocate dinamic

Alocarea dinamică pentru o matrice este importantă deoarece foloseşte economic memoria şi permite matrice cu linii de lungimi diferite. De asemenea reprezintă o soluție bună la problema parametrilor de funcții de tip matrice.

O matrice alocată dinamic este de fapt un *vector de pointeri către fiecare linie din matrice*, deci un *vector de pointeri la vectori alocați dinamic*. Dacă numărul de linii este cunoscut sau poate fi estimată valoarea lui maximă, atunci vectorul de pointeri are o dimensiune constantă. O astfel de matrice se poate folosi la fel ca o matrice declarată cu dimensiuni constante.
Exemplu de declarare matrice de întregi:

```
int * a[M];          // M este o constanta simbolica
```

Dacă nu se poate estima numărul de linii din matrice atunci şi vectorul de pointeri se alocă dinamic, iar declararea matricei se face ca pointer la pointer:

```
int** a;
```

În acest caz se va aloca mai întâi memorie pentru un vector de pointeri (funcție de numărul liniilor) şi apoi se va aloca memorie pentru fiecare linie cu memorarea adreselor liniilor în vectorul de pointeri.
Notația *a[i][j]* este interpretată astfel pentru o matrice alocată dinamic:

- *a[i]* conține un pointer (o adresă *b*)
- *b[j]* sau *b+j* conține întregul din poziția *j* a vectorului cu adresa *b*.

Exemplu

Să se scrie funcții de alocare a memoriei şi afişare a elementelor unei matrice de întregi alocată dinamic.

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

// rezultat adresa matrice sau NULL
int ** intmat ( int nl, int nc) {
int i;
int ** p=(int **) malloc (nl*sizeof (int*));
if ( p != NULL)
for (i=0; i<nl ;i++)
p[i] =(int*) calloc (nc,sizeof (int));
return p;
}

void printmat (int ** a, int nl, int nc) {
int i,j;

for (i=0;i<nl;i++) {
for (j=0;j<nc;j++)
printf ("%2d", a[i][j] );
printf("\n");
}
```

```
}

int  main () {
int  **a, nl, nc, i, j;

printf ("nr linii şi nr coloane: \n");
scanf ("%d%d", &nl, &nc);
a= intmat(nl,nc);

for (i=0;i<nl;i++)
for (j=0;j<nc;j++)
a[i][j]= nc*i+j+1;

printmat (a ,nl,nc);
return 0;
}
```

Funcția *printmat* dată anterior nu poate fi folosită pentru afișarea unei matrice cu dimensiuni constante. Explicația este interpretarea diferită a conținutului zonei de la adresa aflată în primul argument.
Astfel, chiar dacă exemplul următor este corect sintactic el nu se execută corect:

```
int x [2][2]={{1,2},{3,4}};          // 2 linii şi 2 coloane
printmat ( (int**)x, 2, 2);
```

IB.10.9. Funcții cu rezultat vector

O funcție **nu** poate avea ca rezultat un vector sub forma:

```
int [] funcție(…) {…}
```

O funcție poate avea ca rezultat doar un pointer !!

```
int *funcție(…) {…}
```

De obicei, rezultatul pointer este egal cu unul din argumente, eventual modificat în funcție.
Exemplu corect:

```
// incrementare pointer p
char * incptr ( char * p) {
   return ++p;
}
```

Atenție! Acest pointer nu trebuie să conțină adresa unei variabile locale, deoarece:

- O variabilă locală are o existență temporară, garantată numai pe durata executării funcției în care este definită (cu excepția variabilelor locale statice)
- Adresa unei astfel de variabile nu trebuie transmisă în afara funcției, pentru a fi folosită ulterior!!

Exemplu greşit:

```
// vector cu cifrele unui nr intreg de maxim cinci cifre
```

```
int * cifre (int n) {
  int k, c[5];          // vector local
  for (k=4;k>=0;k--) {
    c[k]=n%10; n=n/10;
  }
  return c;             // aici este eroarea !
}
//warning la compilare şi POSIBIL rezultate greşite în main!!
```

O funcție care trebuie să transmită ca rezultat un vector poate fi scrisă corect în în mai multe feluri:

1. **Primeşte ca parametru adresa vectorului (*definit şi alocat în altă funcție*)** şi depune rezultatele la adresa primită (este soluția recomandată!!)

```
void cifre (int n, int c[ ]) {
  int k;
  for (k=4;k>=0;k--) {
    c[k]=n%10; n=n/10;
  }
}
int main(){
      int a[10];
      ….
      cifre(n,a);
      ….
}
```

2. **Alocă dinamic memoria** pentru vector (cu "malloc")

- Această alocare (pe heap) se menține şi la ieşirea din funcție.
- Funcția are ca rezultat adresa vectorului alocat în cadrul funcției.
- Problema este unde şi când se eliberează memoria alocată.

```
int * cifre (int n) {
  int k, *c;            // vector local
  c = (int*) malloc (5*sizeof(int));
  for (k=4;k>=0;k--) {
    c[k]=n%10; n=n/10;
  }
  return c;             // corect
}
```

3. O soluție oarecum echivalentă este **utilizarea unui vector local static**, care continuă să existe şi după terminarea funcției.

IB.10.10. Vectori de pointeri la date alocate dinamic

Ideea folosită la matrice alocate dinamic este aplicabilă şi pentru alte date alocate dinamic: adresele acestor date sunt reunite într-un vector de pointeri. Situațiile cele mai frecvente sunt:

- vectori de pointeri la şiruri de caractere alocate dinamic
- vectori de pointeri la structuri alocate dinamic.

Exemplu de utilizare a unui vector de pointeri la structuri alocate dinamic:

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

typedef  struct {
   int zi, luna, an;
} date;
```

```
// afisare date reunite în vector de pointeri
void print_vp ( date * vp[], int n) {
   int i;
   for(i=0;i<n;i++)
      printf ("%4d %4d %4d \n", vp[i]->zi, vp[i]->luna, vp[i]->an);
   printf ("\n");
}

int main () {
   date d, *dp;
   date *vp[100];
   int n=0;

   while (scanf ("%d%d%d", &d.zi, &d.luna, &d.an) {
      dp=(date*)malloc (sizeof(date));     // alocare dinamica ptr structură
      *dp=d;
             // copiaza datele citite la dp
      vp[n++]=dp;
       // memoreaza adresa in vector
   }
   print_vp (vp,n);
}
```

De reținut că trebuie create adrese distincte pentru fiecare variabilă structură şi că ar fi greşit să punem adresa variabilei ***d*** în toate pozițiile din vector!
Este posibilă şi varianta următoare pentru ciclul principal din *main* dacă cunoaştem numărul de elemente din structură:

```
....
scanf (“%d”,&n);                    // numar de structuri ce vor fi citite
for (k=0; k<n; k++) {
   dp = (date*) malloc (sizeof(date));// alocare dinamica ptr structură
   scanf ("%d%d%d", &dp->zi, &dp->luna, &dp->an)
   vp[n++]=dp;                      // memoreaza adresa in vector
}
....
```

Exemplu
Program pentru citirea unor nume, alocare dinamică a memoriei pentru fiecare şir (în funcție de lungimea şirului citit) şi memorarea adreselor şirurilor într-un vector de pointeri. În final se vor afişa numele citite, pe baza vectorului de pointeri.
Să se adauge programului anterior o funcție de ordonare a vectorului de pointeri la şiruri, pe baza conținutului fiecărui şir. Programul va afişa lista de nume în ordine alfabetică.
a. Vectorului de pointeri i se va aloca o dimenisune fixă.
b. Vectorul de pointeri se va aloca dinamic, funcție de numărul de şiruri.

Rezolvare a:

```
void printstr ( char * vp[], int n) { //afisare
int i;
for(i=0;i<n;i++)
printf ("%s\n",vp[i]);
}

int readstr (char * vp[]) {  // citire siruri şi creare vector de pointeri
```

```
int n=0; char * p, sir[80];
while ( scanf ("%s", sir) == 1) {
vp[n]= (char*) malloc (strlen(sir)+1);
strcpy( vp[n],sir);
//sau:  vp[n]=strdup(sir);
++n;
}
return n;
}

/* ordonare vector de pointeri la şiruri prin Bubble Sort (metoda bulelor)*/
void sort ( char * vp[],int n) {
int i,j,schimb=1;
char * tmp;

while(schimb){
schimb=0;
for (i=0;i<n-1;i++)
if ( strcmp (vp[i],vp[i+1])>0) {
tmp = vp[i];
vp[i] = vp[i+1];
vp[i+1] = tmp;
schimb = 1;
}
}
}

int main () {
int n;
char * vp[1000];   // vector de pointeri, cu dimens. fixa

n=readstr(vp);    // citire siruri şi creare vector
sort(vp,n);       // ordonare vector
printstr(vp,n);   // afişare şiruri
return 0;
}
```

IB.10.11. Anexa A: Structuri alocate dinamic

În cazul variabilelor structură alocate dinamic şi care nu au nume se va face **o indirectare printr-un pointer** pentru a ajunge la variabila structură.

Avem de ales între următoarele două notații echivalente:
pt->camp (*pt).camp
unde *pt* este o variabilă care conține un pointer la o structură cu câmpul **camp**.

O colecție de variabile structură alocate dinamic se poate memora în două moduri:
- Ca un vector de pointeri la variabilele structură alocate dinamic;
- Ca o listă înlăntuită de variabile structură, în care fiecare element al listei conține şi un câmp de legătură către elementul următor (ca pointer la structură).

Pentru primul mod de memorare a se vedea ***Vectori de pointeri la date alocate dinamic.***
Liste înlănțuite
O listă înlănțuită ("linked list") este o colecție de variabile alocate dinamic (de acelaşi tip), dispersate în memorie, dar legate între ele prin pointeri, ca într-un lanț. Într-o listă liniară simplu înlănțuită fiecare element al listei conține adresa elementului următor din listă. Ultimul element poate conține ca adresă de legatură fie constanta NULL, fie adresa primului element din listă (lista circulară).
Adresa primului element din listă este memorată într-o variabilă cu nume şi numită cap de lista ("list head"). Pentru o listă vidă variabila cap de listă este NULL.
Structura de listă este recomandată atunci când colecția de elemente are un conținut foarte variabil (pe parcursul execuției) sau când trebuie păstrate mai multe liste cu conținut foarte variabil.
Un element din listă (un nod de listă) este de un tip structură şi are (cel puțin) două câmpuri:
- un câmp de date (sau mai multe)
- un câmp de legătură

Definiția unui nod de listă este o definitie recursivă, deoarece în definirea câmpului de legătură se foloseşte tipul în curs de definire.
Exemplu pentru o listă de întregi:

```
typedef struct snod {
     int val ;                //  camp de date
     struct snod * leg ;      // camp de legatura
} nod;
```

Programul următor arată cum se poate crea şi afisa o listă cu adăugare la început (o stivă realizată ca listă înlănțuită):

```
int main ( ) {
  nod *lst=NULL, *nou, * p;              // lst = adresa cap de lista
  int x;
  // creare lista cu numere citite
  while (scanf("%d",&x) > 0) {     // citire numar intreg x
    nou=(nod*)malloc(sizeof(nod));// creare nod nou
    nou->val=x;                    // completare camp de date din nod
    nou->leg=lst; lst=nou;         // legare nod nou la lista
  }
  // afisare listă (fara modificare cap de lista)
  p=lst;
  while ( p != NULL) {             // cat mai sunt noduri
    printf("%d  ", p->val);        // afisare numar de la adr p
```

```
     p=p->leg;                          // avans la nodul urmator
   }
}
```

Câmpul de date poate fi la rândul lui o structură specifică aplicației sau poate fi un pointer la date alocate dinamic (un şir de caractere, de exemplu).
De obicei se definesc funcții pentru operațiile uzuale cu liste.
Exemple:

```
typedef struct nod {                        // un nod de lista inlantuita
     int val;                               // date din fiecare nod
     struct snod *leg;                      // legatura la nodul urmator
} nod;

// insertie la inceput lista
nod* insL( nod* lst, int x) {
     nod* nou ;                             // adresa nod nou
     if ((nou=(nod*)malloc(sizeof(nod)))==NULL)
     return NULL;
     nou->val=x;
     nou->leg=lst;
     return nou;                            // lista incepe cu nou
}

// afisare continut lista
void printL ( nod* lst) {
     while (lst != NULL) {
       printf("%d ",lst->val);              // scrie informatiile din nod
       lst=lst->leg;                        // si se trece la nodul următor
     }
}

int main () {                               // creare si afisare lista stiva
     nod* lst; int x;
     lst=NULL;                              // initial lista e vida
     while (scanf("%d",&x) > 0)
       lst=insL(lst,x);                 // introduce pe x in lista lst
     printL (lst);                          // afisare lista
}
```

Alte structuri dinamice folosesc câte doi pointeri sau chiar un vector de pointeri; într-un arbore binar fiecare nod conține adresa succesorului la stânga şi adresa succesorului la dreapta, într-un arbore multicăi fiecare nod conține un vector de pointeri către succesorii acelui nod.

IB.10.12. Anexa B: Operatori pentru alocare dinamică in C++

În C++ s-au introdus doi operatori noi:

- pentru alocarea dinamică a memoriei ***new***
- pentru eliberarea memoriei dinamice ***delete***

destinați să înlocuiască funcțiile de alocare şi eliberare.
Operatorul ***new*** are ca operand un nume de tip, urmat în general de o valoare inițială pentru variabila creată (între paranteze rotunde); rezultatul lui *new* este o adresă (un pointer de tipul specificat) sau *NULL* daca nu există suficientă memorie liberă.
Exemple:

```
nod * pnod;             // pointer la nod de lista
pnod = new nod;         // alocare fara inițializare
int * p = new int(3);   // alocare cu inițializare
```

Operatorul *new* are o formă puțin modificată la alocarea de memorie pentru vectori, pentru a specifica numărul de componente.
Exemplu:

```
int * v = new int [n];        // vector de n intregi
```

Operatorul ***delete*** are ca operand o variabilă pointer şi are ca efect eliberarea blocului de memorie adresat de pointer, a cărui mărime rezultă din tipul variabilei pointer sau este indicată explicit.
Exemple:

```
int * v;
delete v;         // elibereaza sizeof(int) octeți
delete [ ] v;
delete [n] v;     // elibereaza n*sizeof(int) octeți
```

Exemplu de utilizare new şi delete pentru un vector de întregi alocat dinamic:

```
#include <iostream>
#include <cstdlib>

int main() {
   const int SIZE = 5;
   int *pArray;

   pArray = new int[SIZE];  // alocare memorie

   // atribuie numere aleatoare intre 0 and 99
   for (int i = 0; i < SIZE; i++) {
      *(pArray + i) = rand() % 100;
   }

   // afisare
   for (int i = 0; i < SIZE; i++) {
      cout << *(pArray + i) << " ";
   }
   cout << endl;

   delete[] pArray;  // eliberare memorie
   return 0;
}
```

După alocarea de memorie cu *new* se pot folosi funcțiile *realloc* şi *free* pentru realocare sau eliberare de memorie.

Capitolul IB.11. Operații cu fișiere în limbajul C

Cuvinte cheie
Fişiere text, fişiere binare,
deschidere/ închidere fişiere, citire-scriere fişiere,
acces direct, fişiere predefinite, funcția fflush, redirectarea fişierelor
standard, fişiere în C++

IB.11.1. Noțiunea de fişier

Un fişier este o colecție de date memorate pe un suport extern şi care este identificată printr-un nume.

Conținutul fişierelor poate fi foarte variat:
- texte, inclusiv programe sursă
- numere
- alte informații binare: programe executabile, numere în format binar, imagini sau sunete codificate numeric ş.a.

Numărul de elemente ale unui fişier este variabil (poate fi nul).
Fişierele de date se folosesc pentru:
- date inițiale mai numeroase
- rezultate mai numeroase
- păstrarea permanentă a unor date de interes pentru anumite aplicații.

Fişierele sunt entități ale sistemului de operare şi ca atare ele au nume care respectă convențiile sistemului, fără legătură cu un anume limbaj de programare. Operațiile cu fişiere sunt realizate de către sistemul de operare, iar compilatorul unui limbaj traduce funcțiile de acces la fişiere în apeluri ale funcțiilor de sistem.

De obicei prin *fişier* se subînțelege un fişier disc (pe suport magnetic sau optic), dar noțiunea de fişier este mai generală şi include orice flux de date din exterior spre memorie sau dinspre memoria internă spre exterior. De aceea s-a introdus cuvântul *stream*, tradus prin flux de date si sinonim cu fişier logic, deci orice sursă sau destinație externă a datelor.
Stream (flux de date, canal) este în acest context sinonim cu *file* (fişier): pune accent pe aspectul dinamic al transferului de date.
Programatorul se referă la un fişier printr-o variabilă; tipul acestei variabile depinde de limbajul folosit şi chiar de funcțiile utilizate (în C). Asocierea dintre numele extern (un şir de caractere) şi variabila din program se face la deschiderea unui fişier, printr-o funcție standard.

IB.11.2. Tipuri de fişiere în C

Fişiere text
- **conțin o succesiune de linii, separate prin NewLine**
- **fiecare linie are 0 sau mai multe caractere tipăribile şi/sau tab**

Fişiere binare
- **conțin o succesiune de octeți**

Un fişier text conține numai caractere ASCII, grupate în linii de lungimi diferite, fiecare linie terminată cu unul sau două caractere terminator de linie.

Caracter terminator de linie:

- **fişierele Unix/Linux: un singur caracter terminator de linie '\n'**
- **fişierele Windows şi MS-DOS: caracterele '\r' şi '\n' (CR,LF) ca terminator de linie**

Un fişier text poate fi terminat printr-un *caracter terminator de fişier* (*Ctrl-Z* = *EOF*)
Valoarea efectivă este dependentă de sistem, dar în general este -1.

Acest terminator nu este însă obligatoriu. Sfârşitul unui fişier disc poate fi detectat şi pe baza lungimii fişierului (număr de octeți), memorată pe disc.

Funcțiile de citire sau de scriere cu format din/în fişiere text realizează conversia automată din format extern (şir de caractere) în format intern (binar virgulă fixă sau virgulă mobilă) la citire şi conversia din format intern în format extern, la scriere pentru numere întregi sau reale.

Fişierele binare pot conține:
- numere în reprezentare internă (binară)
- articole (structuri de date)
- fişiere cu imagini grafice, în diverse formate, etc

Citirea şi scrierea se fac fără conversie de format.
Pentru fiecare tip de fişier binar este necesar un program care să cunoască şi să interpreteze corect datele din fişier (structura articolelor). Este posibil ca un fişier binar să conțină numai caractere, dar funcțiile de citire şi de scriere pentru aceste fişiere nu cunosc noțiunea de linie; ele specifică numărul de octeți care se citesc sau se scriu.

Consola şi imprimanta sunt considerate fişiere text.
Fişierele disc trebuie deschise şi închise, dar fişierele consolă şi imprimanta nu trebuie deschise şi închise.

IB.11.3. Operarea cu fişiere

Pentru operarea cu un fişier (text sau binar) se defineşte o variabilă de tip ***FILE **** pentru accesarea fişierului:

FILE * - tip structură definită în stdio.h

Conține informații referitoare la fişier şi la tamponul de transfer de date între memoria centrală şi fişier:
- adresa
- lungimea tamponului
- modul de utilizare a fişierului
- indicator de sfârşit de fişier
- indicator de poziție în fişier

Etapele pentru operarea cu un fişier în limbajul C sunt:
- se deschide fişierul pentru un anumit mod de acces, folosind funcția de biblioteca ***fopen***,
  - realizează şi asocierea între variabila fişier şi numele extern al fişierului
- se prelucrează fişierul
  - operații citire/scriere
- se închide fişierul folosind funcția de biblioteca ***fclose***.

IB.11.4. Funcţii pentru deschidere şi închidere fişiere

Funcțiile standard pentru acces la fişiere sunt declarate în *stdio.h*. După cum spuneam mai devreme, funcțiile de citire/scriere/pozitionare în fişier folosesc pentru identificarea unui fişier o variabilă pointer la o structură predefinită *FILE*.

IB.11.4. 1 Deschiderea unui fişier

Pentru a citi sau scrie dintr-un/într-un fişier disc, acesta trebuie mai întâi deschis folosind funcția *fopen*.

FILE *fopen (const char *numefisier, const char *mod);

- Deschide fişierul cu numele dat pentru acces de tip mod
- Returnează pointer la fişier sau NULL dacă fişierul nu poate fi deschis
- Valoarea returnată este memorată în variabila fişier, care a fost declarată (FILE *) pentru accesarea lui.

unde:

- ***numefisier***: numele fişierului
- ***mod***: şir de caractere (între 1 şi 3 caractere):
  - ***r*** - readonly , este permisă doar citirea dintr-un fişier existent
  - ***w*** - write, crează un nou fişier, sau dacă există deja, distruge vechiul conținut
  - ***a*** - append, deschide pentru scriere un fişier existent (scrierea se va face în continuarea informației deja existente în fişier, deci pointerul de acces se plasează la sfârşitul fişierului)
  - **+** - permite scrierea şi citirea din acelasi fişier - actualizare (ex: "r+", "w+", "a+").
  - ***t*** sau ***b*** - tip fişier ("text", "binary"), implicit este ***t***

Primul argument al funcției *fopen* este numele extern al fişierului scris cu respectarea convențiilor limbajului C.

Numele fişier extern poate include următoarele:

- **Numele unității de disc sau partiției disc (** ex: A:, C:, D:, E:)
- **Calea spre fişier**: succesiune de nume de fişiere catalog (director), separate printr-un caracter ('\' în MS-DOS şi MS-Windows, sau '/' în Unix şi Linux)
- **Numele propriu-zis al fişierului**
- **Extensia,** care indică tipul fişierului şi care poate avea între 0 şi 3 caractere în MS-DOS.

Sistemele MS-DOS şi MS-Windows nu fac deosebire între litere mari şi litere mici, în cadrul numelor de fişiere.

Atenție! pentru separarea numelor de cataloage dintr-o cale se vor folosi:

- \\ - pentru a nu se considera o secvență de caractere e*scape* sau:
- caracterul /

Exemple:

```
char *numef = "C:\\WORK\\T.TXT";
char *numef = "c:/work/t.txt";
```

La deschiderea unui fişier se inițializează variabila pointer asociată, iar celelalte funcții se referă la fişier numai prin intermediul variabilei pointer.
Funcția *fopen* are rezultat NULL (0) dacă fişierul specificat nu este găsit după căutare în directorul curent sau pe calea specificată.

Exemplu:

```
//exemplu 1
char *numef = "C:\\WORK\\T.TXT";    // sau c:/work/t.txt
FILE * f;                           // pentru referire la fişier
if ( (f=fopen(numef,"r")) == NULL) {
   printf("Eroare la deschidere fişier %s \n", numef);
   return;
}

//exemplu 2
#include <stdio.h>
int main ( ) {
   FILE * f; // pentru referire la fişier

   // deschide un fişier binar ptr citire
   f = fopen ( "c:\\t.txt", "rb" );

   printf ( f == NULL ? "Fişier negasit" : " Fişier gasit");
   ...
   if (f)          // dacă fişier existent
      fclose(f);   // închide fişier
   return 0;
}
```

Diferența dintre *b* şi *t* este aceea că la citirea dintr-un fişier binar toți octeții sunt considerați ca date şi sunt transferați în memorie, iar la citirea dintr-un fişier text anumiți octeți sunt interpretați ca terminator de linie (\0x0a) sau ca terminator de fişier (\0x1a). Nu este obligatoriu ca orice fişier text să se termine cu un caracter special cu semnificația *sfârşit de fişier* (*CTRL-Z* , de exemplu) .

Pentru fişierele text sunt folosite modurile:

- *w* pentru crearea unui nou fişier
- *r* pentru citirea dintr-un fişier
- *a* pentru adăugare la sfârşitul unui fişier existent
- Modul *w+* poate fi folosit pentru citire după creare fişier.

Deschiderea în modul *w* şterge orice fişier existent cu acelaşi nume, fără avertizare, dar programatorul poate verifica existența unui fişier în acelaşi director înainte de a crea unul nou.
Pentru fişierele binare se practică actualizarea pe loc a fişierelor, fără inserarea de date între cele existente, deci modurile *r+*, *a+*, *w+*. (literele *r* şi *w* nu pot fi folosite simultan).

Fişierele standard de intrare-ieşire (tastatura şi ecranul consolei) au asociate variabile de tip pointer cu nume predefinit (*stdin* şi *stdout*); care pot fi folosite în funcțiile destinate tuturor fisierelor, cum ar fi *fflush*.

Pentru închiderea unui fişier disc se foloseşte funcția ***fclose***:

int fclose(FILE *fp);

- închide fişierul şi eliberează zona tampon
- în caz de succes întoarce 0, altfel, întoarce **EOF**.

Închiderea este absolut necesară pentru fişierele în care s-a scris ceva, dar poate lipsi dacă s-au făcut doar citiri din fişier.

IB.11.5. Operații uzuale cu fişiere text

Accesul la fişiere text se poate face
- **fie la nivel de linie**
- **fie la nivel de caracter**

dar numai secvenţial.

Deci nu se pot citi/scrie linii sau caractere decât în ordinea memorării lor în fişier şi nu pe sărite (aleator)!
Nu se pot face modificări într-un fişier text fără a crea un alt fişier, deoarece nu sunt de conceput deplasări de text în fişier!

Pentru citire/scriere din/în fişierele standard stdin/stdout se folosesc funcţii cu nume puţin diferit şi cu mai puţine argumente, dar se pot folosi şi funcţiile generale destinate fişierelor disc. Urmează câteva perechi de funcţii:

Sintaxa	Descriere
int fgetc (FILE * f); **// sau getc (FILE*)**	Citeşte un caracter din ***f*** şi îl întoarce ca un unsigned char convertit la int, Returnează ***EOF*** dacă s-a întâlnit sfârşitul de fişier sau în caz de eroare.
int fputc (int c, FILE * f); **// sau putc (int, FILE*)**	Scrie caracterul cu codul ascii ***c*** în fişier
char * fgets(char * line, int max, FILE *f);	Citeşte maxim n-1 caractere sau până la ***\n*** inclusiv, şi le depune în ***s***, adaugă la sfârşit ***\0*** dar nu elimină terminatorul de linie *\n*. Returnează adresa şirului. La eroare întoarce valoarea ***NULL***.
int fputs (char * line, FILE *f);	Scrie şirul ***line*** în fişier, fără caracterul '\0'. La eroare întoarce ***EOF***.

Detectarea sfârşitului de fişier se poate face şi cu ajutorul funcţiei ***feof*** (*Find End of File*):

int feof (FILE *fp);
- testează dacă s-a ajuns la *end-of-file* al fişierului referit de fp
- returnează 0 dacă nu s-a detectat sfârşit de fişier la ultima operaţie de citire, respectiv o valoare nenulă (adevarată) pentru sfârşit de fişier.

Atenţie! Rezultatul lui *feof* se modifică după încercarea de a citi după sfârşitul fişierului!
Se va scrie în fişierul de ieşire şi –1, rezultatul ultimului apel al funcţiei *fgetc*:

```
while ( ! feof(f1))
    fputc(fgetc(f1),f2);
```

Soluţia preferabilă pentru ciclul de citire-scriere caractere este următoarea:

```
while ( (ch=fgetc(f1)) != EOF)
            fputc ( ch, f2);
```

Exemplu:

```
// citire şi afişare linii dintr-un fişier
#include<stdio.h>
#include<stdlib.h>

int main()
{
   FILE *fp;
   char s[80];
   if ( (fp=fopen("c:\\test.c","r")) == NULL ) {
```

```
     printf ( "Nu se poate deschide la citire fişierul!\n" );
     exit (1);
   }
   while ( fgets(s,80,fp) != NULL )
     printf ( "%s", s);
   fclose (fp);
   return 0;
}

/*
Scriere sub formă de litere mici caracterele dintr-un fişier în alt fişier
numele sursei şi destinaţiei transmise în linia de comandă.

Lansarea în execuţie a programului:
copiere fişier_sursa.dat fişier_dest.dat
*/
#include<stdio.h>
#include<ctype.h>

int main(int argc, char** argv){
   FILE * f1, * f2;  int ch;
   f1= fopen (argv[1], "r");
   f2= fopen (argv[2], "w");
   if ( f1==0 || f2==0) {
     puts (" Eroare la deschidere fişiere \n");
     return 1;
   }
   while ( (ch=fgetc(f1)) != EOF)        // citeste  din f1
     fputc ( tolower(ch),f2);           // scrie în f2
   fclose(f1);
   fclose(f2);
   return 0;
}
```

În principiu se poate citi integral un fişier text în memorie, dar în practică se citeşte o singură linie sau un număr de linii succesive, într-un ciclu repetat până se termină fişierul (pentru a se putea prelucra fişiere oricât de mari).

Pentru actualizarea unui fişier text prin modificarea lungimii unor linii, ştergerea sau inserţia de linii se va scrie un alt fişier şi nu se vor opera modificările direct pe fişierul existent.

IB.11.6. Intrări/ieşiri cu conversie de format

Datele numerice pot fi scrise în fişiere disc fie în format intern (mai compact), fie transformate în şiruri de caractere (cifre zecimale, semn ş.a).
Un fişier text ocupă mai mult spaţiu deoarece formatul şir de caractere necesită şi caractere separator între numere. Avantajul este că un fişier text poate fi citit cu programe scrise în orice limbaj sau cu orice editor de texte sau cu alt program utilitar de vizualizare fişiere!

Funcţiile de citire-scriere cu conversie de format şi editare sunt:

int fscanf (FILE * f, char * fmt, ...)
realizează citirea cu format dintr-un fişier; analog scanf

int fprintf (FILE * f, char * fmt, ...)
identică cu printf cu deosebirea că scrie într-un fişier

Pentru aceste funcţii se aplică toate regulile de la funcţiile *scanf* şi *printf*.

Un fişier text prelucrat cu funcţiile *fprintf* şi *fscanf* conţine mai multe câmpuri de date separate între ele prin unul sau mai multe spaţii albe (*blanc, tab, linie nouă*). Conţinutul câmpului de date este scris şi interpretat la citire conform specificatorului de format pentru acel câmp.
Exemplu de creare şi citire fişier de numere:

```
FILE * f;                          // f = pointer la fişier
int x;

// creare fişier de date:
f = fopen("num.txt", "w");         // deschide fişier
for (x=1;x<=100;x++)
   fprintf(f,"%4d", x);            // scrie un numar

fclose(f);                         // inchidere fişier

// citire şi afişare fişier creat:
f=fopen("num.txt", "r");
if ( (f == NULL ) {
     printf ( "Nu se poate deschide la citire fişierul!\n" );
     exit (1);
}
while (fscanf(f,"%d", &x) == 1)          // pana la sfirsit  fişier
   printf("%4d", x);                     // afişare numar citit
```

Exemplu:
Într-un fişier de tip text sunt păstrate valorile reale ale unei măsuratori sub forma:

nr_măsuratori
val1
val2
val3 ...

Să se scrie programul care afişează numărul de măsurători şi valorile respective.
Se vor adăuga la fişier noi măsuratori până la introducerea valorii 0.
Rezolvare:

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#define MAX  100

void afiseaza(double *mas,int nrm){
   int i;
   for (i=0; i<nrm; i++)
     printf ("Masuratoarea %d = %6.2e\n", i+1, mas[i]);
}

void loadmas (FILE *fp, int *nrm, double *mas){
   int i=0;
   fscanf (fp,"%d", nrm);
   if (*nrm>MAX) *nrm = MAX;
   for ( ; i<*nrm; i++)
     fscanf (fp, "%lf", &mas[i]);
}

int main(){
   FILE *fp;
   double masur[MAX], mas_noua=1;
   char nume_fis[12];
   int nr_mas;
```

```
   printf ("nume fisier:");
   gets (nume_fis);
   if ( (fp = fopen(nume_fis, "r+") ) == 0 ){
      printf ("nu exista fisierul\n");
      exit (1);
   }
   loadmas (fp,&nr_mas,masur);
   afiseaza (masur,nr_mas);
   fclose(fp);

   if ( (fp = fopen(nume_fis, "a") ) == 0 ){
      printf ("nu exista fisierul\n");
      exit (1);
   }
   printf ("\nmasuratori noi:\n");
   while( nr_mas++<MAX-1){
      scanf("%lf",&mas_noua);
      if(mas_noua) fprintf (fp, "%lf\n", mas_noua);
      else break;
   }
   fclose(fp);

   if ( (fp = fopen(nume_fis, "r+") ) == 0 ){
      printf ("nu exista fisierul\n");
      exit (1);
   }
   fprintf (fp, "%d", --nr_mas);
   fclose (fp);
   return 0;
}
```

IB.11.7. Funcții de citire-scriere pentru fişiere binare

Un fişier binar este format în general din articole de lungime fixă, fără separatori între articole. Un articol poate conține:

- un singur octet
- un număr binar (pe 2, 4 sau 8 octeți)
- structură cu date de diferite tipuri

Funcțiile de acces pentru fişiere binare *fread* şi *fwrite* pot citi sau scrie unul sau mai multe articole, la fiecare apelare. Transferul între memorie şi suportul extern se face fără conversie sau editare (adăugare de caractere la scriere sau eliminare de caractere la citire). Prototipuri funcții intrare/iesire (fişiere binare – b):

size_t fread (void *ptr, size_t size, size_t nmemb, FILE *fp)
Citeşte la adresa **ptr** cel mult **nmemb** elemente de dimensiune **size** din fişierul referit de **fp**

size_t fwrite (void *ptr, size_t size, size_t nmemb, FILE *fp)
Scrie în fişierul referit de **fp** cel mult **nmemb** elemente de dimensiune **size** de la adresa **ptr**

Exemple:

```
int a[10];
fread (a, sizeof(int), 10, fp);
fwrite(a, sizeof(int),10,fp);
```

De remarcat că primul argument al funcțiilor *fread* şi *fwrite* este o adresă de memorie (un pointer): adresa unde se citesc date din fişier sau de unde se iau datele scrise în fişier.
Al doilea argument este numărul de octeți pentru un articol, iar al treilea argument este numărul de articole citite sau scrise. Numărul de octeți citiți sau scrişi este egal cu produsul dintre lungimea unui articol şi numărul de articole.

Rezultatul funcțiilor este numărul de articole efectiv citite sau scrise şi este diferit de argumentul 3 numai la sfârşit de fişier (la citire) sau în caz de eroare de citire/scriere!

Dacă ştim lungimea unui fişier şi dacă este loc în memoria RAM atunci putem citi un întreg fişier printr-un singur apel al funcției *fread* sau putem scrie integral un fişier cu un singur apel al funcției *fwrite*. Citirea mai multor date dintr-un fişier disc poate conduce la un timp mai bun față de repetarea unor citiri urmate de prelucrări, deoarece se pot elimina timpii de aşteptare pentru poziționarea capetelor de citire – scriere pe sectorul ce trebuie citit (rotație disc plus comandă capete).

Programul următor scrie mai multe numere întregi într-un fişier disc (unul câte unul) şi apoi citeşte conținutul fişierului şi afişează pe ecran numerele citite.

```
int main () {
   FILE * f; int x;  char * numef =“num.bin”;
   // creare fişier
   f=fopen(numef,"wb"));                  // fisier in directorul curent
   for (x=1; x<=100; x++)
     fwrite (&x,sizeof(float),1,f);
   fclose(f);

    // citire fişier pentru verificare
   if ( (f=fopen(numef,"rb")) == NULL ) { // fisier in directorul curent
     printf ( "Nu se poate deschide la citire fişierul!\n“ );
     exit (1);
   }
   printf("\n");
   while (fread (&x,sizeof(float),1,f)==1)
     printf ("%4d ",x);
   fclose(f);
   return 0;
}
```

Lungimea fişierului *num.bin* este de 200 de octeți, câte 2 octeți pentru fiecare număr întreg, în timp ce lungimea fişierului *num.txt* creat anterior cu funcția *fprintf* este de 400 de octeți (câte 4 caractere ptr fiecare număr). Pentru alte tipuri de numere diferența poate fi mult mai mare.

De obicei articolele unui fişier au o anumită structură, în sensul că fiecare articol conține mai multe câmpuri de lungimi şi tipuri diferite. Pentru citirea sau scrierea unor astfel de articole în program trebuie să existe (cel puțin) o variabilă structură care să reflecte structura articolelor.
Exemplu de definire a structurii articolelor unui fişier simplu cu date despre elevi şi a funcțiilor ce scriu sau citesc articole ce corespund unor variabile structură:

```
typedef struct {
  char nume[25];
  float medie;
} Elev;

// creare fişier cu nume dat
void creare(char * numef) {
   FILE * f;  Elev s;
```

```
   f=fopen(numef,"wb");
   printf ("Nume şi medie ptr. fiecare student: \n");
   while (scanf ("%s %f ", s.nume, &s.medie) != EOF)
      fwrite(&s,sizeof(s),1,f);
   fclose (f);
}

// afişare conţinut fişier pe ecran
void listare (char* numef) {
   FILE * f;  Elev e;
   if ( (f=fopen(numef,"rb")) == NULL ) { // fisier in directorul curent
      printf ( "Nu se poate deschide la citire fişierul!\n" );
      exit (1);
   }
   while (fread (&e,sizeof(e),1,f)==1)
      printf ("%-25s %6.2f \n",e.nume, e.medie);
   fclose (f);
}

// adaugare articole la sfârşitul unui fişier existent
void adaugare (char * numef) {
   FILE * f;  Elev e;
   if ( (f=fopen(numef,"ab")) == NULL ) { // fisier in directorul curent
      printf ( "Nu se poate deschide la citire fişierul!\n" );
      exit (1);
   }
   printf ("Adaugare nume şi medie:\n");
   while (scanf ("%s%f", e.nume, &e.medie) != EOF)
      fwrite (&e, sizeof(e), 1, f);
   fclose (f);
}
```

IB.11.8. Funcţii pentru acces direct la datele dintr-un fişier

Accesul direct la date dintr-un fişier este posibil numai pentru un fişier cu articole de lungime fixă şi înseamnă posibilitatea de a citi sau scrie oriunde într-un fişier, printr-o poziţionare prealabilă înainte de citire sau scriere. Fişierele mari care necesită regăsirea rapidă şi actualizarea frecventă de articole vor conţine numai articole de aceeaşi lungime.

În C poziţionarea se face pe un anumit octet din fişier, iar funcţiile standard permit accesul direct la o anumită adresă de octet din fişier. Funcţiile pentru acces direct din *stdio.h* permit operaţiile următoare:

- Poziţionarea pe un anumit octet din fişier (*fseek*).
- Citirea poziţiei curente din fişier (*ftell*).
- Memorarea poziţiei curente şi poziţionare (*fgetpos, fsetpos*).

Poziţia curentă în fişier este un număr de tip *long*, pentru a permite operaţii cu fişiere foarte lungi. Poziţia se obţine printr-un apel al funcţiei *ftell*:

long int ftell (FILE *fp)
Întoarce valoarea indicatorului de poziţie

- pentru fişier binar: numărul de octeţi de la începutul fişierului
- pentru fişier text: o valoare ce poate fi utilizată de *fseek* pentru a seta indicatorul de poziţie în fişier la această poziţie.

Funcţia *fseek* are prototipul următor :

int fseek (FILE *fp, long int offset, int poziţie)
poziţionează indicatorul de poziţie la valoarea dată de *offset* faţă de:

- SEEK_SET sau 0 ⇔ **începutul fişierului**

- SEEK_CUR sau 1⇔ **poziția curentă**
- SEEK_END sau 2 ⇔ **sfârşitul fişierului**

Offset reprezintă numărul de octeți față de punctul de referință.

Exemple:

- poziționarea la sfârşitul fişierului: **fseek (fp, 0, SEEK_END)**
- poziționarea la caracterul precedent: **fseek (fp, -1, SEEK_CUR)**
- poziționarea la inceputul fişierului: **fseek (fp, 0, SEEK_SET)**

Atenție! Poziționarea relativă la sfârşitul unui fişier nu este garantată nici chiar pentru fişiere binare, astfel că ar trebui evitată!
Ar trebui evitată şi poziționarea față de poziția curentă cu o valoare negativă, care nu funcționează în toate implementările!
Funcția *fseek* este utilă în următoarele situații:

- Pentru repoziționare pe început de fişier după o căutare şi înainte de o altă căutare secvențială în fişier (fără a închide şi a redeschide fişierul)
- Pentru poziționare pe începutul ultimului articol citit, în vederea scrierii noului conținut (modificat) al acestui articol, deoarece orice operație de citire sau scriere avansează automat poziția curentă în fişier, pe următorul articol.
- Pentru acces direct după conținutul unui articol (după un câmp cheie), după ce s-a calculat sau s-a găsit adresa unui articol cu cheie dată.

Într-un fişier text poziționarea este posibilă numai față de începutul fişierului, iar poziția se obține printr-un apel al funcției *ftell*.

Modificarea conținutului unui articol (fără modificarea lungimii sale) se face în mai mulți paşi:

- Se caută articolul ce trebuie modificat şi se reține adresa lui în fişier (înainte sau după citirea sa);
- Se modifică în memorie articolul citit;
- Se readuce poziția curentă pe începutul ultimului articol citit;
- Se scrie articolul modificat, peste conținutul său anterior.

Exemplu de secvență pentru modificarea unui articol:

```
pos=ftell (f);
fread (&e,sizeof(e),1,f );          // poziția inainte de citire
. . .
      // modifica ceva in variabila e
fseek (f,pos,0);                    // repoziționare pe articolul citit
fwrite (&e,sizeof(e),1,f);          // rescrie ultimul articol citit
```

Memorarea poziției curente sau poziționarea se pot realiza utilizând următoarele funcții:

int fgetpos (FILE *fp, fpos_t *poziție)

- **memorează starea curentă a indicatorului de poziție al fluxului referit de *fp* în poziție;**
- **întoarce 0 dacă operația s-a realizat cu succes!**

int fsetpos (FILE *fp, const fpos_t *poziție)

- **setează indicatorul de poziție al fluxului referit de *fp* la valoarea data de poziție**

void rewind (FILE *fp)

- **setează indicatorul de poziție al fluxului referit de *fp* la începutul fişierului**

Exemplu:
Funcție care modifică conținutul mai multor articole din fișierul de elevi creat anterior.

```
// modificare conținut articole, dupa cautarea lor
void modificare (char * numef) {
   FILE * f;
   Elev e;
   char nume[25];
   long pos;
   int  ef;

   if ( (f=fopen(numef,"rb+")) == NULL ) {// fisier in directorul curent
     printf ( "Nu se poate deschide la citire fișierul!\n" );
     exit (1);
   }

   do {
     printf ("Nume cautat: ");
     scanf ("%s",nume);
     if (strcmp(nume, ".") == 0)  break; // datele se termină cu un punct

     // cauta "nume" în fișier
     fseek (f, 0, 0);                    // readucere pe inceput de fișier
     while ( (ef=fread (&e, sizeof(e), 1, f)) ==1 )
       if (strcmp (e.nume, nume)==0) {
            pos= ftell(f) - sizeof(e);
            break;
       }
     if ( ef < 1) break;

     printf ("noua medie: ");
     scanf ("%f", &e.medie);

     fseek (f, pos, 0);             // pozit. pe inceput articol gasit
     fwrite (&e, sizeof(e), 1, f);  // rescrie articol modificat
   } while (1);

   fclose (f);
}

int main(){
   char name[]="c:elev.txt";
   creare (name);
   listare (name);
   adaugare (name);
   modificare (name);
   listare (name);
   return 0;
}
```

IB.11.9. Fişiere predefinite

Există trei ***fluxuri predefinite***, care se deschid automat la lansarea unui program:

- ***stdin*** - fişier de intrare, text, este intrarea standard - tastatura
- ***stdout*** - fişier de ieşire, text, este ieşirea standard - ecranul monitorului.
- ***stderr*** - fişier de iesire, text, este ieşirea standard de erori - ecranul monitorului.

Ele pot fi folosite în diferite funcții, un exemplu practic este funcția *fflush* care goleşte zona tampon (*buffer*) asociată unui fişier.

Observații

- Nu orice apel al unei funcţii de citire sau de scriere are ca efect imediat un transfer de date între exterior şi variabilele din program!
- Citirea efectivă de pe suportul extern se face într-o zonă tampon asociată fişierului, iar numărul de octeţi care se citesc depind de suport: o linie de la tastatură, unul sau câteva sectoare disc dintr-un fişier disc, etc.
- Cele mai multe apeluri de funcţii de I/E au ca efect un transfer între zona tampon (anonimă) şi variabilele din program.
- Este posibil ca să existe diferenţe în detaliile de lucru ale funcţiilor standard de citire-scriere din diferite implementări (biblioteci), deoarece standardul C nu precizează toate aceste detalii!

Funcţia *fflush*

Are rolul de a goli zona tampon folosită de funcţiile de I/E, zonă altfel inaccesibilă programatorului C. Are ca argument variabila pointer asociată unui fişier la deschidere, sau variabilele predefinite *stdin* şi *stdout*.

```
fflush (FILE* f);
```

Exemple de situaţii în care este necesară folosirea funcţiei *fflush*:

- Citirea unui caracter după citirea unui câmp sau unei linii cu *scanf* :

```
int main () {
   int n;
   char s[30];
   char c;

   scanf ("%d",&n);                  // sau  scanf("%s",s);
   // fflush(stdin);                 // pentru corectare

   c = getchar();                    // sau  scanf ("%c", &c);
   printf ("%d \n",c);               // afiseaza codul lui c
   return 0;
}

/*  va afişa 10 care este codul numeric al caracterului terminator de
linie \n, în loc să  afişeze codul caracterului c, deoarece după o
citire cu scanf în zona tampon rămân unul sau câteva caractere
separator de câmpuri ('\n', '\t', ' '), care trebuie scoase de acolo
prin fflush(stdin) sau prin alte apeluri scanf.  */
```

- Funcţia *scanf* opreşte citirea unei valori din zona tampon ce conţine o linie la primul caracter separator de câmpuri sau la un caracter ilegal în câmp (de ex. literă într-un câmp numeric)!
  In cazul repetării unei operatii de citire (cu *scanf*) după o eroare de introducere în linia anterioară (caracter ilegal pentru un anumit format de citire) în zona tampon rămân caracterele din linie care urmau după cel care a produs eroarea!

```
do {
   printf ("x, y = ");
   err = scanf ("%d%d", &x, &y);
   if ( err == 2 ) break;
   fflush (stdin);
} while (err != 2);
```

  Observaţie: După citirea unei linii cu funcţiile "gets" sau "fgets" nu rămâne nici un caracter în zona tampon şi nu este necesar apelul lui "fflush"!

- Se va folosi periodic *fflush* în cazul actualizării unui fişier mare, pentru a evita pierderi de date la producerea unor incidente (toate datele din zona tampon vor fi scrise efectiv pe disc):

```
int main () {
```

```
      FILE * f;
      int c;
      char numef[]="TEST.DAT";
      char x[ ] = "0123456789";

      f=fopen (numef,"w");
      for (c=0;c<10;c++)
        fputc (x[c], f);
      fflush (f);      // sau fclose(f);

      f=fopen (numef,"r");
      while ( (c=fgetc(f)) != EOF)
        printf ("%c", c);
      return 0;
}
```

IB.11.10. Redirectarea fişierelor standard

Trebuie observat că programele C care folosesc funcţii standard de I/E cu consola pot fi folosite, fără modificări, pentru preluarea de date din orice fişier şi pentru trimiterea rezultatelor în orice fişier, prin operaţia numită redirectare (redirecţionare) a fişierelor standard. Prin redirectare, fişierele standard se pot asocia cu alte fişiere.
Redirectarea se face prin adăugarea unor argumente în linia de comandă la apelarea programului.
Exemplu:

```
 fişier_exe < fişier_1  > fişier_2
```

În acest caz, preluarea informaţiilor se face din fişier_1, iar afişarea informaţiilor de ieşire se face în fişier_2.
Exemple:

```
/*
* Copierea conţinutului unui fişier în alt fişier utilizand redirectarea
* Folosind redirectarea fişierelor standard, se va lansa printr-o linie de
* comandă de forma:
*    copiere1 <fişier_sursa.dat >fişier_dest.dat
* Şi atunci următorul program va avea acelaşi rezultat ca si când s-ar citi
* din fisier_sursa.dat   * şi s-ar scrie în fişier_dest.dat
*/

#include <stdio.h>
int main(void){
   char c;
   while ( (c=getchar()) != EOF )
     putchar(c);
   return 0;
}
```

```
/*
* Exemplu de program "filter":
* filter este numele unui program (fişier executabil) care aplică un filtru
* oarecare pe un text pentru a produce un alt text
* Folosind redirectarea fişierelor standard, se va lansa printr-o linie de
* comandă de forma:
* filter <input  -  citire din input şi scriere pe ecran
* filter >output  - citire de la consola şi scriere în output
* filter <input  >output -  citire din input şi scriere în output
*/

#include <stdio.h>
// pentru funcţiile gets,puts
int main () {
   char line[256];                    // aici se citeşte o linie
   while ( gets(line) != NULL)        // repeta citire linie
```

```
   if ( line[0]=='/' && line[1]=='/')      // daca linie comentariu
                  puts (line);             // atunci se scrie linia
   return 0;
}
```

Utilizarea comenzii *filter* fără argumente citeşte şi afişează la consolă; utilizarea unui argument de forma <*input* redirectează intrările către fişierul *input*, iar un argument de forma >*output* redirectează ieşirile către fişierul *output*.

Redirectarea se poate aplica numai programelor care lucrează cu fişiere text.

IB.11.11. Anexa. Fişiere în C++

Fişierele sunt în C++ variabile de tipurile ***ifstream*** (*input file stream*), ***ofstream*** (*output file stream*) sau ***fstream*** (care permit atât citire cât şi scriere din fişier).
Operaţiile de citire/scriere se pot realiza fie prin funcţii specifice, fie prin operatorii de inserţie în flux (<<) sau extragere din flux (>>). Pentru a putea fi folosite, fişierele disc trebuie deschise, utilizând funcţia *open* şi închise după folosire utilizând funcţia *close*.

Exemplu de scriere într-un fişier text:

```
ofstream ofile;
char nr[4];
ofile.open ("numere.txt");
for (int i=1;i<100;i++)
   ofile << itoa (i,nr,10)<< endl;
ofile.close();
```

La citirea dintr-un fişier text există două diferenţe faţă de scriere:

- Trebuie detectat sfârşitul de fişier cu una din funcţiile ***eof(), good()*** sau ***bad().***
- Citirea cu operatorul >> repetă ultima linie citită din fişier şi de aceea se preferă funcţia ***getline*** (cu argument de tip *string* şi nu vector de caractere).

Exemplu de citire din fişierul text creat anterior:

```
ifstream ifile; char nr[4];
ifile.open ("numere.txt");
while (ifile.good()) {              // while ( ! ifile.eof()) {
   ifile >> nr;
   cout << nr << endl;
}
ifile.close();                      // poate lipsi
```

Se poate verifica dacă deschiderea fişierului a reuşit cu funcţia ***is_open()*** :

```
if (! ifile.is_open()) cout << "eroare la deschidere\n";
```

Capitolul IB.12. Convenţii şi stil de programare

Cuvinte cheie
Stil de programare, convenţii de scriere,
identificatori, indentare, spaţiere, directive preprocesor, macrouri

IB.12.1 Stil de programare – coding practices

Comparând programele scrise de diverşi autori în limbajul C se pot constata diferenţe importante în:

- modul de redactare al textului sursă (utilizarea de acolade, utilizarea de litere mici şi mari, etc.). Acest mod de redactare poate fi supus utilizării anumitor convenţii de programare
- modul de utilizare a elementelor limbajului (instrucţiuni, declaraţii, funcţii, etc.), aşa numitul stil de programare, propriu fiecărui programator, dar care poate fi supus totuşi unor reguli de bază.

O primă diferenţă de abordare este alegerea între a folosi cât mai mult facilităţile specifice oferite de limbajul C sau de a folosi construcţii comune şi altor limbaje (Pascal de ex.).
Exemple de construcţii specifice limbajului C:

- Expresii complexe, incluzând prelucrări, atribuiri şi comparaţii.
- Utilizarea de operatori specifici: atribuiri combinate cu alte operaţii, operatorul condiţional, etc.
- Utilizarea instrucţiunilor *break* şi *continue*.
- Utilizarea de pointeri în locul unor vectori sau matrice.
- Utilizarea unor declaraţii complexe de tipuri, în loc de a defini tipuri intermediare, mai simple.

Exemplu:

```
// definire vector de pointeri la functii void f(int,int)
void (*tp[M])(int,int);             // greu de citit!

// definire cu tip intermediar pointer la functie
typedef void (*funPtr) (int,int);   // pointer la o functie cu 2 argumente int
funPtr tp[M];                       // vector cu M elemente de tip funPtr
```

O alegere oarecum echivalentă este între programe sursă cât mai compacte (cu cât mai puţine instrucţiuni şi declaraţii) şi programe cât mai explicite şi mai uşor de înţeles. În general este preferabilă calitatea programelor de a fi uşor de citit şi de modificat şi mai puţin lungimea codului sursă şi, eventual, lungimea codului obiect generat de compilator.
Deci se recomandă programe cât mai clare şi nu programe cât mai scurte.

Exemplu de secvenţă pentru afişarea a n întregi câte m pe o linie:

```
for (i=1; i<=n; i++)
    printf ( "%5d%c", i, ( i%m==0 || i==n)? '\n':' ');
```

O variantă mai explicită dar mai lungă pentru secvenţa anterioară:

```
for (i=1; i<=n; i++){
  printf ("%6d ", i);
     if (i%m==0)
        printf("\n");
}
printf("\n");
```

Alte recomandări

- Se recomandă utilizarea standardului ANSI C pentru portabilitate

- Programele nu trebuie să depindă de caracteristicile compilatorului (ordinea de evaluare a expresiilor
  Exemple:

```
k = ++i + i;      /* gresit */
y = f(x) + z_glb; /* gresit daca f() schimba valoarea lui z_glb*/
k = ++i + j++;    /* OK */
a[i++] = j++;     /* OK */
```

- Orice definire (de structură, enumerare, tip, etc) utilizată în mai multe fişiere va fi inclusă într-un fişier antet (***.h***) care va fi apoi inclus în fişierele care folosesc acea definiție
- Toate conversiile de tip vor fi făcute explicit
- Variabilele structură se vor transmite prin adresa
- Pentru constantele utilizate pentru activarea / dezactivarea unor instrucțiuni se va verifica definirea acestora utilizând compilările condiționate

```
// Nerecomandat:
#define DEBUG 4 /* folosit pentru a indica nivelul dorit de debug */
for (i = 0; i < 5 && DEBUG; i++)
{
      printf("i = %d\n", i);
}

// Recomandat:
#define DEBUG
#ifdef DEBUG
      for (i = 0; i < 5; i++)
      {
            printf("i = %d\n", i);
      }
#endif
```

- Pentru un simbol testat cu *#ifdef, #ifndef sau #if defined* nu se va defini o valoare

```
// Nerecomandat
#define DEBUG 0
#ifdef DEBUG
for (i = 0; i < 5; i++)
{
      printf("i = %d\n", i);
}
#endif

//Recomandat
#define DEBUG
#ifdef DEBUG
      for (i = 0; i < 5; i++)
      {
            printf("i = %d\n", i);
      }
#endif
```

A se vedea anexa ***Directive preprocesor utile în programele mari. Macrouri***

- Elementele unui vector vor fi accesate utilizând [] şi nu operatorul de dereferențiere *.

```
// Nerecomandat
int array[11];
*(array + 10) = 0;
// Recomandat
int array[11];
array[10] = 0;
```

- Transmiterea parametrilor prin pointeri va fi evitată ori de cate ori este posibil

```
x = f(a, b, c);
// şi
x = f(a, b, c, x);
//sunt mai uşor de înţeles decât:
f(a, b, c, &x);
```

- Toate instrucţiunile *switch* vor avea clauza *default* care întotdeauna va fi ultima

```
switch (variabila_int)
{
      case:..
            break;
      case:..
            break;
      default:..
}
```

- Operatorul virgulă (‘**,**’) va fi utilizat doar în instrucţiunea *for* şi la declararea variabilelor
- Modificarea unui cod existent se va face conform standardului existent deja în acel cod
- Modulele unui program nu trebuie să depăşească un anumit grad de complexitate şi un anumit număr de linii (maxim o jumătate de pagină)
- Se va utiliza evaluarea condiţiei afirmative mai degrabă decât a celei negative (!)
- Condiţiile logice vor fi scrise explicit:

```
//Nerecomandat
if (is_available)
if (sys_cfg__is_radio_retry_allowed())
if (intermediate_result)
//Recomandat
if (is_available == FALSE)
if (sys_cfg__is_radio_retry_allowed() == TRUE)
if (intermediate_result != 0)
```

- Nu se recomandă utilizarea variabilelor globale; daca vor fi utilizate trebuie indeplinite următoarele cerinţe:
  - Toate variabilele globale pentru un proiect vor fi definite într-un singur fişier
  - Variabilele globale vor fi iniţializate înainte de utilizare
  - O variabilă globală va fi definită o singură dată pentru un program executabil

Funcţii

- Se vor folosi pe cât posibil funcţii standard în loc de funcţii specifice sistemului de operare
  - System Dependent: *open(), close(), read(), write(), lseek(),* etc.
  - ANSI Functions: *fopen(), fclose(), fread(), fwrite(), fseek(),* etc.
- Toate funcţiile vor avea tip definit explicit
- Funcţiile care întorc pointeri vor returna NULL în cazul neîndeplinirii unei condiţii
- Numărul parametrilor unei funcţii ar trebui limitat la cinci sau mai puţin
- Toate funcţiile definite trebuie însoţite de antete ce vor conţine lista tipurilor parametrilor

Constante

- Toate constantele folosite într-un fişier vor fi definite înainte de prima funcţie din fişier
- Dacă se definesc constantele TRUE şi FALSE, acestea trebuie să aibă valoare 1, respectiv 0:

```
#ifndef TRUE
      #define TRUE 1
#endif
#ifndef FALSE
      #define FALSE 0
```

```
#endif
```

- Definirea de constante simbolice (*#define*) va fi preferată utilizării directe a valorilor:

```
//Recomandat:
#define NMAX 100
int a[NMAX];

//Nerecomandat:
int a[100];
```

Variabile

- Toate variabilele se definesc înainte de partea de cod propriu-zis (instrucțiuni)
- Variabilele locale se definesc câte una pe linie; excepție fac indecşii, variabilele temporare şi variabilele inițializate cu aceeaşi valoare

```
int   Zona;
int   i, j, contor;
int   Mode = k = 0;
```

- Variabilele locale ar trebui inițializate înainte de utilizare
- Se va evita utilizarea variabilelor globale pe cât posibil

Dimensiune vectori

- Nu se transmite dimensiunea maximă a unui vector când acesta este parametru al unei funcții; aceasta se transmite separat într-o variabilă!

```
// Nerecomandat
char *substring(char string[80], int start_pos, int length)
{…
}
// Recomandat
char *substring(char string[], int start_pos, int length)
{…
}
```

Includerea fişierelor

- Fişierele antet vor defini o constantă simbolică pentru a permite includerea multiplă. Dacă fişierul se numeşte *file.h* constanta se poate numi *FILE_H*

```
// fisierul example.h
#ifndef EXAMPLE_H
#define EXAMPLE_H
…
#endif
```

- Se recomandă ca fişierele antet să nu includă alte fişiere antet
- Pentru includerea unui fişier antet definit de utilizator se vor utiliza “ ”, iar pentru o bibiloteca standard < >

Macrouri

- În macrourile tip funcție parametrii vor fi scrişi între paranteze

```
// Nerecomandat
#define prt_debug(a, b) printf("ERROR: %s:%d, %s\n", a,__LINE__, b);

// Recomandat
#define prt_debug(a, b) printf("ERROR: %s:%d, %s\n", (a),__LINE__,(b));
```

- Macrourile complexe vor fi comentate
- Un macrou nu va depăşi 10 linii

IB.12.2. Convenţii de scriere a programelor

Programele sunt destinate calculatorului şi sunt analizate de către un program compilator. Acest compilator ignoră spaţiile albe nesemnificative şi trecerea de la o linie la alta.
Programele sunt citite şi de către oameni, fie pentru a fi modificate sau extinse, fie pentru comunicarea unor noi algoritmi sub formă de programe. Pentru a fi mai uşor de înţeles de către oameni se recomandă folosirea unor convenţii de trecere de pe o linie pe alta, de aliniere în cadrul fiecărei linii, de utilizare a spaţiilor albe şi a comentariilor.
Respectarea unor convenţii de scriere în majoritatea programelor poate contribui la reducerea diversităţii programelor scrise de diverşi autori şi deci la facilitarea înţelegerii şi modificării lor de către alţi programatori.
O serie de convenţii au fost stabilite de autorii limbajului C şi ai primului manual de C.

Beneficiile utilizării unor convenţii de programare:

- Uniformizează modul de scriere a codului
- Facilitează citirea şi înţelegerea unui program
- Facilitează întreţinerea aplicaţiilor
- Facilitează comunicarea între membrii unei echipe ceea ce duce la un randament sporit al lucrului în echipă

Observaţie:
Chiar dacă unele persoane pot resimţi ca o "îngrădire" aceste convenţii, ele permit totuşi manifestarea creativităţii programatorului deoarece orice convenţie poate fi imbunătăţită şi adoptată ca standard al echipei respective de programatori.

IB.12.2.1 Identificatori

- Identificatorii trebuie să îndeplinească standardul ANSI C (lungimea<31, caractere permise: litere, cifre, _)
- Simbolul '_' nu va fi folosit ca prim caracter al unui identificator
- Nu se vor folosi nume utilizate de sistem decât dacă se doreşte înlocuirea acestora (constante, fişiere, funcţii)
- Toate fişierele header vor avea extensia ***.h***
- Toate constantele simbolice definite cu *#define* vor fi scrise cu litere mari
- Numele de variabile şi de funcţii încep cu o literă mică şi conţin mai mult litere mici (litere mari numai în nume compuse din mai multe cuvinte alăturate, cum sunt nume de funcţii din *MS-Windows*)
- În ceea ce priveşte numele unor noi tipuri de date părerile sunt împărţite
- Numele unui fişier nu trebuie să depşească 14 caractere în lungime (cu extensie).
- Variabilele locale, definiţiile de tip (*typedef*), numele de fişiere cât şi membrii unei structuri vor fi scrişi cu litere mici
- Numele variabilelor şi funcţiilor trebuie să fie semnificative şi în concordanţă cu ceea ce reprezintă.

Exemple:

Tip	Exemplu nume
Contor, index	i, j, k, l, g, h,
Pointeri	ptr_var, var_ptr, var_p, p_var, p
Variabile temporare	tmp_var, var_tmp, t_va
Valori returnate	status, return_value

Funcţii	readValues, reset_status, print_array
Fişiere	initialData.txt, entry.txt, setFuncţions.c, set.h

- Variabilele globale trebuie să se diferenţieze de cele locale (fie primul caracter literă mare, fie un sufix de genul *global*): **Vec_initial, vec_global, set_glb**

IB.12.2.2 Funcţii

- Se va scrie o singură instrucţiune pe linie
- Tipul unei funcţii şi numele acesteia vor fi pe aceeaşi linie

```
// Nerecomandat
int
err_msg(int error_code)
{
…
}

// Recomandat
int err_msg(int error_code)
{
…
}
```

IB.12.2.3 Spaţierea

Nu vor fi spaţii albe:

- După un cast explicit de tip

```
// Nerecomandat
int x = 1;
double y = 3.0;
y = (double) x + 16.7;

// Recomandat
int x = 1;
double y = 3.0;
y = (double)x + 16.7;
```

- Între operatorii unari (&, *, -, ~, ++, --, !, cast, sizeof) şi operanzii lor
- Înainte sau după operatorii primari (“()”,”[]”,”.”,”->”)
- Între caracterul # şi directiva de preprocesare

```
// Nerecomandat
#
define TEST 0
// sau
# define TEST 0

// Recomandat
#define TEST 0
```

- Între numele unei funcţii şi paranteza care îi urmează
- între primul argument al funcţiei şi paranteza deschisă
- între ultimul argument al funcţiei şi paranteza închisă

```
// Nerecomandat
just_return ( arg1, arg2 );

// Recomandat
just_return(arg1, arg2);
if (x == y)…
```

- Între parantezele deschisă, respectiv închisă şi expresia unei instrucţiuni condiţionale

```
if (x == y)…
```

Un singur spaţiu

- va exista între expresia condiţională a unei instrucţiuni şi numele ***if***

```
// Nerecomandat
if(x == y)

// Recomandat
if (x == y)
```

- precede şi urmează operatorii de atribuire, operatorii relaţionali, operatorii logici, operatorii aritmetici (excepţie cei unari) operatorii pe biţi şi operatorul condiţional

```
a = 3;
```

- va urma unei virgule

```
int a, b, c;
```

Alte recomandări:

- Va exista cel puţin o linie goală care să separe definirea variabilelor locale de instrucţiuni

```
int just_return(int first_arg, int second_arg)
{
      /*-------------- LOCAL VARIABLES -------------*/
      int i = 0;  /* Loop counter. */
      int j = 0;  /* Loop counter. */
      /*-------------------- CODE ------------------*/
      /*
      Body of funcţion just_return.
      */
      return(0);
}
```

- Membrii unei structuri, uniuni, enumerări vor fi plasaţi pe linii distincte la declararea lor
- Componentele logice ale unei expresii condiţionale vor fi grupate cu paranteze chiar dacă acestea nu sunt necesare

```
if ((x == y) && (a == b))
```

IB.12.2.4 Utilizarea acoladelor şi parantezelor

Una dintre convenţii se referă la modul de scriere a acoladelor care încadrează un bloc de instrucţiuni ce face parte dintr-o funcţie sau dintr-o instrucţiune *if, while, for* etc. Cele două stiluri care pot fi întâlnite în diferite programe şi cărti sunt ilustrate de exemplele următoare:

- **Stil Kernighan & Ritchie**

```
// exemplu bucla for
for (i = 0; i < loop_cntrl; i++){
      /* Corp for. */
}

// descompunere in factori primi
int main(){
  int n, k, p ;

  printf("\n n= ");
  scanf("%d",&n);
  printf("1");                        // pentru simplificarea afisarii
  for (k=2; k<=n && n>1; k++) {
     p=0;                          // puterea lui k in n
```

```
      while (n % k == 0) {     // cat timp n se imparte exact prin k
        p++;
        n = n / k;
      }
      if (p > 0)               // nu scrie factori la puterea zero
        printf (" * %d^%d",k,p);
    }
}
```

- **Stil Linux:** Toate acoladele vor fi câte una pe linie

```
// exemplu bucla for
for (i = 0; i < loop_cntrl; i++)
{
      /* Corp for. */
}

// descompunere in factori primi
int main()
{
 int n, k, p ;

 printf("\n n= ");
 scanf("%d",&n);
 printf("1");                  // pentru simplificarea afisarii
 for (k=2; k<=n && n>1; k++)
 {
    p=0;                       // puterea lui k in n
    while (n % k ==0)          // cat timp n se imparte exact prin k
    {
       p++;
       n = n / k;
    }
    if (p > 0)                 // nu scrie factori la puterea zero
       printf (" * %d^%d",k,p);
 }
}
```

- Uneori se recomandă utilizare de acolade chiar şi pentru o singură instrucțiune, anticipând adăugarea altor instrucțiuni în viitor la blocul respectiv.

```
if (p > 0) {               // scrie numai factori cu putere nenula
     printf(" * %d^%d",k,p);
}
```

IB.12.2.5 Indentarea

Pentru alinierea spre dreapta la fiecare bloc inclus într-o structură de control se pot folosi caractere Tab (‘\t’) sau spații, dar evidențierea structurii de blocuri incluse este importantă pentru oamenii care citesc programe.

Recomandări:

- Toate definițiile de funcții încep în coloana 1
- Acoladele ce definesc corpul unei funcții vor fi în coloana 1 sau imediat după antet
- Acoladele corespunzătoare unei instrucțiuni, unei inițializări de structură, vector, etc vor fi în aceeaşi coloană cu instrucțiunea sau inițializarea respectivă

```
for (i = 0; i < loop_cntrl; i++)
{
      /* corp for */
```

```
}
```

- Instrucțiunile aflate la acelaşi nivel de includere vor fi indentate la aceeaşi coloană

```
if (conditie == TRUE)
{
      /* corp prim if */
}
else
      if
      {
            /* corp al doilea if */
      }
      else
      {
            /* else al doilea if */
      }
```

- Toate blocurile incluse în alt bloc vor fi indentate cu 2 până la 4 spații albe

Vor fi indentate cu 2 - 4 spaţii:

- câmpurile unui tip de date

```
struct example_type
{
    int x;
    double y;
};
```

- ramurile *case* ale unei instrucțiuni *switch*
- continuarea unei linii, față de operatorul de atribuire sau față de paranteza deschisă în cazul unei instrucțiuni sau a unui apel de funcție

```
num = this example test structure.example struct field1 *
    this_example_test_structure.example_struct_field2;

if ((very_long_result_variable_name >=
     lower_specification_value))
```

IB.12.2.6 Comentarii

O serie de recomandări se referă la modul cum trebuie documentate programele folosind comentarii.
Astfel, fiecare funcție C ar trebui precedată de comentarii ce descriu

- rolul acelei funcții
- semnificația argumentelor funcției
- rezultatul funcției pentru terminare normală şi cu eroare
- precondiții - condiții care trebuie satisfăcute de parametri efectivi primiți de funcție (limite, valori interzise, etc.) şi care pot fi verificate sau nu de funcție
- plus alte date despre:
  - autor
  - data ultimei modificări
  - alte funcții utilizate sau asemănătoare, etc.

Exemplu:

```
/*
 Functie de conversie numar întreg pozitiv
 din binar în sir de caractere ASCII terminat cu zero
 "value" = numar intreg primit de functie (pozitiv)
 "string" = adresa unde se pune sirul rezultat
```

```
 "radix" = baza de numeratie (intre 2 şi 16, inclusiv)
 are ca rezultat adresa sir sau NULL in caz de eroare
 trebuie completata pentru numere cu semn
*/
char *itoa(int value, char *string, int radix) {
      char digits[] = "0123456789ABCDEF";
      char t[20], *tt=t, * s=string;
      if ( radix > 16 || radix < 0 || value < 0) return NULL;
      do {
            *tt++ = digits[ value % radix];
      } while ( (value = value / radix) != 0 );
      while ( tt != t)
               *string++= *(--tt);
      *string=0;
      return s;
}
```

Alte observaţii legate de comentarii:

- Vor completa codul, nu îl vor dubla!
- Explică mai mult decât este subînțeles din cod
- Nu trebuie să fie foarte multe comentarii (îngreunează codul) dar nici foarte puține (nu este explicat codul)

Pot fi comentarii bloc, pe o linie, sau in-line:

- Comentarii bloc: descriu secțiunile principale ale programului şi vor fi indentate la acelaşi nivel cu codul pe care il comentează

```
/*
Acesta este un format care poate fi folosit pentru comentariile bloc
*/

/**********************************************************
*
Si acesta este un format care poate fi folosit pentru comentariile bloc
*
**********************************************************/

/*
**********************************************************
*
Si acesta este un format care poate fi folosit pentru comentariile bloc
*
**********************************************************
*/
```

- Comentarii pe o linie : Se indentează la acelaşi nivel cu codul pe care îl comentează

```
if (argc > 1)
{
      /* ia numele fisierului de intrare din linia de comanda. */
      if ((freopen(argv[1], 'r', stdin) == NULL)
      {
            /* Corp if */
      }
}
```

- Comentarii in-line (pentru descrierea declarațiilor): trebuie să fie indeajuns de scurte încât să intre pe aceeaşi linie cu codul comentat

```
int i = 0;              /* Contor bucla */
int status = TRUE;      /* Rezultatul funcției*/
```

- Pentru comentarii pe o linie sau in- line se pot folosi şi comentarii C++:

```
int i = 0;              // Contor bucla
int status = TRUE;      // Rezultatul funcției
```

Reguli generale privind comentariile

- Comentariile nu vor fi incluse unele în altele
- Fiecare variabilă locală va avea un comentariu ce va descrie utilizarea ei dacă aceasta nu reiese din nume
- Comentariile in-line trebuie să fie aliniate pe cât posibil la stânga în cadrul unei funcții

```
j = 5;      /* Assign j to the starting string position */
k = j + 9;  /* Assign k to the ending string position  */
```

- Instrucțiunile condiționale sau buclele complexe (mai mult de 10 linii necomentate) vor avea ataşat un comentariu la acolada de închidere ce va indica închiderea instrucțiunii şi unul la începutul sau în interiorul blocului ce va indica scopul acestuia

```
if (a>b){
      /* scop
      ….
      ….*/
      …
} // end of if(a>b)
```

IB.12.3. Anexa: Directive preprocesor utile în programele mari. Macrouri

Directivele preprocesor C au o sintaxă şi o prelucrare distinctă de instrucțiunile şi declarațiile limbajului, dar sunt parte a standardului limbajului C.
Directivele sunt interpretate într-o etapă preliminară compilării (traducerii) textului C, de către un preprocesor.
O directivă începe prin caracterul # şi se termină la sfârşitul liniei curente (dacă nu există linii de continuare a liniei curente).
Nu se foloseşte caracterul **;** pentru terminarea unei directive!

Cele mai importante directive preprocesor sunt:

Sintaxa	Descriere
#define *ident text*	înlocuieşte toate aparițiile identificatorului *ident* prin şirul *text*
#define *ident* (a1,a2,...) *text*	defineşte o macroinstrucțiune cu argumente
#include *fişier*	include în compilare continutul fişierului sursa *fişier*
#if *expr*	compilare condiționată de valoarea expresiei *expr*
#if defined *ident*	compilare condiționată de definirea unui identificator (cu *#define*)
#endif	terminarea unui bloc introdus prin directiva *#if*

Directiva define are multiple utilizări în programele C:

- Definirea de constante simbolice de diferite tipuri (numerice, text)
  Exemple:

```
#define begin {     // unde apare begin acesta va fi înlocuit cu {
#define end }       // unde apare end acesta va fi înlocuit cu }
#define N 100       // unde apare N acesta va fi înlocuit cu 100
```

- Definirea de **macrouri** cu aspect de funcție, pentru compilarea mai eficientă a unor funcții mici, apelate în mod repetat.
  Exemple:

```
// maxim dintre a si b
#define max(A,B)   ( (A)>(B) ? (A):(B) )

// generează un număr aleator între 0 şi num!
#define random(num)(int) (((long)rand()*(num))/(RAND_MAX+1))

// initializare motor de generare numere aleatoare
#define randomize() srand((unsigned)time(NULL))

// valoarea absoluta
#define abs(a) (a)<0 ? -(a) : (a)

// numar par cu utilizare!
#include<stdio.h>
#define PAR(a) a%2==0 ? 1 : 0
int main(void)
{
      if (PAR(9+1)) printf("este par\n");
      else printf("este impar\n");
      return 0;
}

Atenție!
9+1%2==0 va conduce la 9+0 == 0 F ->”este impar”
Ar trebui:
#define PAR(a) (a)%2==0 ? 1 : 0
```

Macrourile pot conține şi declarații, se pot extinde pe mai multe linii şi pot fi utile în reducerea lungimii programelor sursă şi a efortului de programare.
În standardul din 1999 al limbajului C s-a preluat din C++ cuvântul cheie *inline* pentru declararea funcțiilor care vor fi compilate ca macroinstrucțiuni în loc de a folosi macrouri definite cu *define*.

- Definirea unor identificatori specifici fiecărui fişier şi care vor fi testați cu directiva ***ifdef***. De exemplu, pentru a evita declarațiile *extern* în toate fişierele sursă, mai puțin fişierul ce conține definițiile variabilelor externe, putem proceda astfel:
  - Se defineşte în fişierul sursă cu definițiile variabilelor externe un nume simbolic oarecare:

```
// fişier ul DIRLIST.C
#define MAIN
```

  - În fişierul ***dirlist.h*** se plasează toate declarațiile de variabile externe, dar încadrate de directivele if şi endif:

```
// fişier ul DIRLIST.H
#if !defined(MAIN)            // sau ifndef MAIN
      extern char path[MAXC], mask[MAXC], opt[MAXC];
#endif
```

Directiva *include* este urmată de obicei de numele unui fişier antet (de tip *H = header*), fişier care grupează declarații de tipuri, de constante, de funcții şi de variabile, necesare în mai multe fişiere sursă (C sau CPP).

- Fişierele antet nu ar trebui să conțină definiții de variabile sau de funcții, pentru că pot apare erori la includerea multiplă a unui fişier antet.

- Un fişier antet poate include alte fişiere antet.
- Pentru a evita includerea multiplă a unui fişier antet (standard sau nestandard) se recomandă ca fiecare fişier antet să înceapă cu o secvență de felul următor:

```
#ifndef  HDR
#define HDR
  // continut fişier  HDR.H ...
#endif
```

Fişierele antet standard (*stdio.h*, etc.) respectă această recomandare.
O soluție alternativă este ca în fişierul ce face includerea să avem o secvență de forma următoare:

```
#ifndef STDIO_H
#include <stdio.h>
#define _STDIO_H
#endif
```

Directivele de compilare condiționată de forma ***if...endif*** au şi ele mai multe utilizări ce pot fi rezumate la adaptarea codului sursă la diferite condiții specifice, cum ar fi:

- dependența de modelul de memorie folosit (în sistemul MS-DOS)
- dependența de sistemul de operare sub care se foloseşte programul (de ex., anumite funcții sau structuri de date care au forme diferite în sisteme diferite)
- dependența de fişierul sursă în care se află (de exemplu *tcalc.h*).

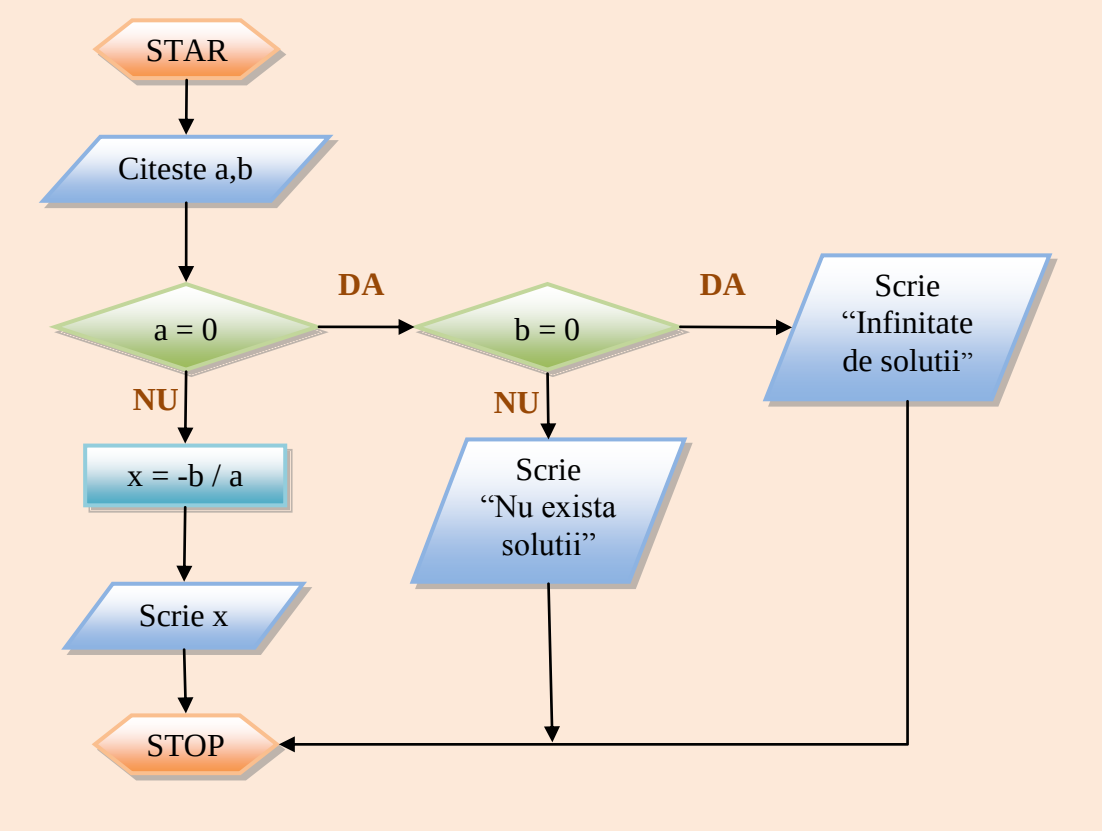
Directivele din grupul ***if*** au mai multe forme, iar un bloc ***if ... endif*** poate conține şi o directivă ***elseif***.

Capitolul IB.13. Autoevaluare

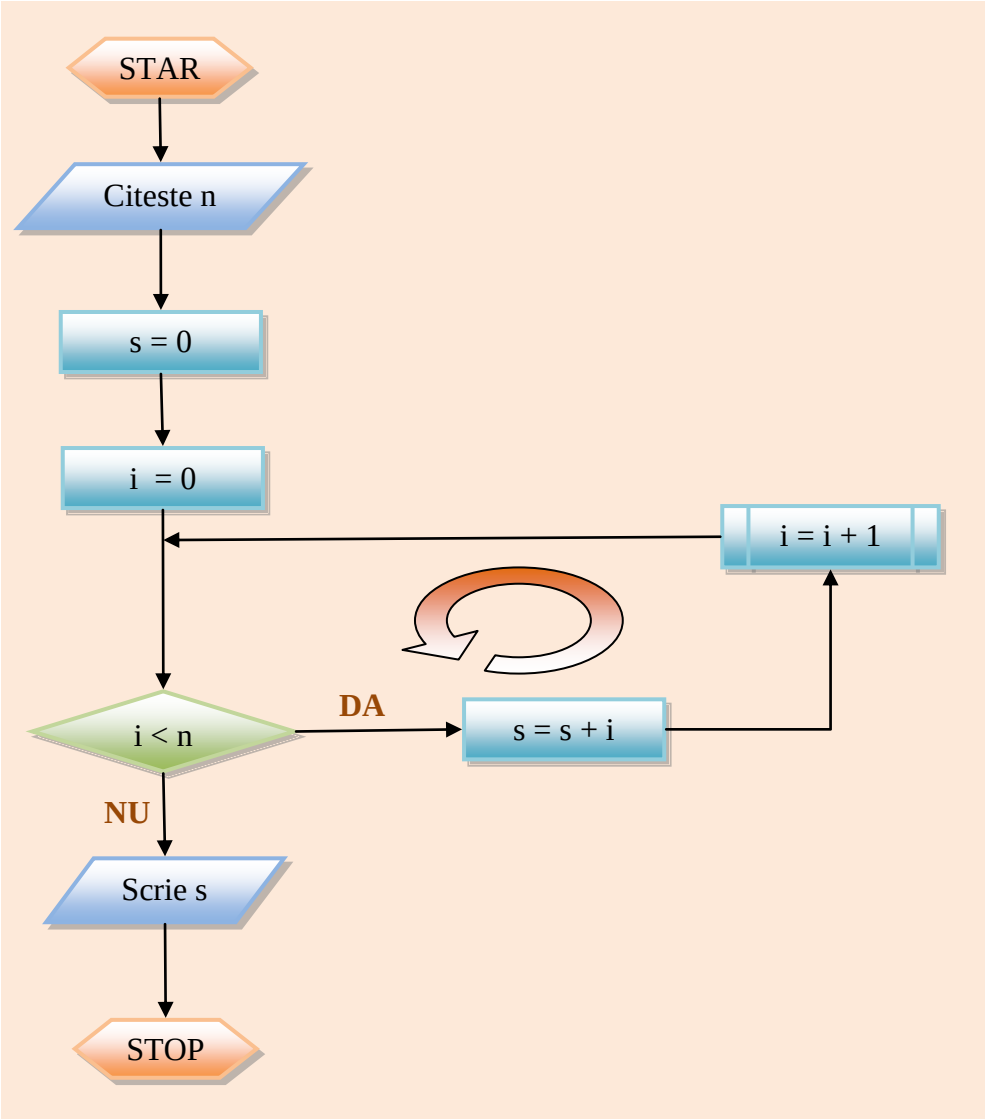
Capitol IB.01. Rezolvarea algoritmică a problemelor

● **Probleme rezolvate în schemă logică**

Problema 1: Rezolvarea ecuației de grad 1: ax+b=0. Atenție la cazurile speciale: a egal zero şi/sau b egal zero!
Date de intrare: a şi b, variabile reale
Date de ieşire: x, variabilă reală

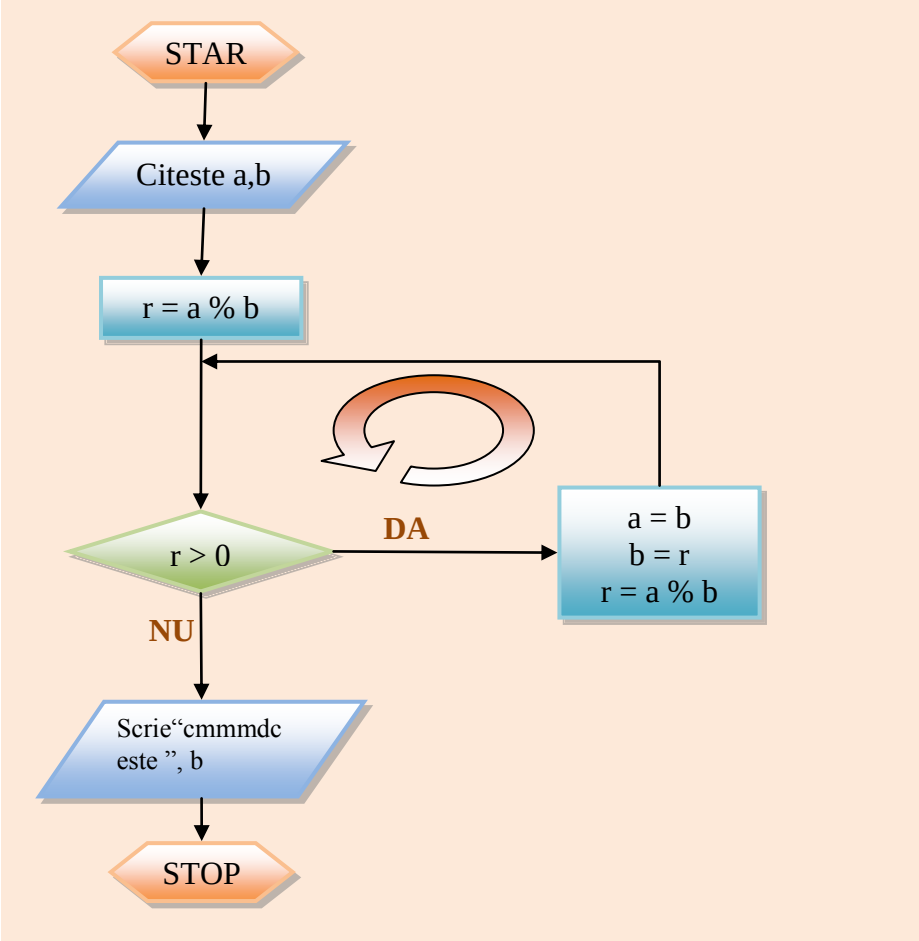


Problema 2: Să se afişeze suma primelor n numere naturale, n citit de la tastatură.
Date de intrare: ***n***, variabilă întreagă
Date de ieşire: ***s,*** variabilă întreagă pozitivă ce stochează suma primelor n numere naturale
Variabile auxiliare: ***i***, variabilă naturală de tip contor



Problema 3: Algoritmul lui Euclid care determină cel mai mare divizor comun a doi întregi, prin împărțiri repetate.
Date de intrare: a şi b, variabile întregi, pozitive. Se consideră că a este mai mare ca b.
Date de ieşire: cmmdc care este calculat în b.
Variabile auxiliare: ***r***, variabilă naturală ce păstrează restul împărțirii lui a la b.

Atenție! Dacă la final vrem să afişăm ceva de genul „Cel mai mare divizor comun al numerelor: ” şi vrem să urmeze valorile inițiale pentru a şi respectiv b, nu vom putea face acest lucru deoarece valorile de intrare ale lui a şi b se pierd, ele modificându-se pe parcursul execuției algoritmului.
Pentru a rezolva această problemă, vom păstra aceste valori inițiale în două variabile auxiliare: copie_a şi copie_b, iar la final vom afişa valorile din aceste două variabile.



● **Probleme propuse şi rezolvate în pseudocod**
Problema 1. Interschimbul valorilor a două variabile a şi b.

Rezolvare: Atenție! Trebuie să folosim o variabilă auxiliară. Nu funcționează a=b şi apoi b=a deoarece deja am pierdut valoarea inițială a lui a!

```
start
citeste a, b;
aux = a;
a = b;
b = aux;
scrie a, b;
stop.
```

Problema 2. Rezolvarea ecuației de grad 2: $ax^2+bx+c=0$.

Rezolvare: Atenție la cazurile speciale! Dacă a este 0 ecuația devine ecuație de gradul 1!

```
start
citeste a, b, c;
daca a = 0
      daca b = 0
            daca c = 0
               scrie "Ecuatia are o infinitate de solutii";
            altfel  scrie "Ecuatia nu are nici o solutie";
      altfel     {
            x =  -c / b;
            scrie "Ecuatia este de gradul I cu solutia:";
            scrie x;
            }

altfel  {
      d = b * b + 4 * a * c;
      daca d < 0
            scrie "Ecutia nu are radacini reale";
      altfel
            daca d = 0 {
                  x= - b / (2 * a);
                  scrie "Ecuatia are 2 radacini egale cu :"
                  scrie x;
            }
            altfel  {
                  x1=  -b + sqrt( d ) / (2 * a );
                  x2 = -b - sqrt( d ) / (2 * a );
                  scrie "Ecuatia are 2 radacini distincte:"
                  scrie x1, x2;
            }
}
stop.
```

Problema 3. Să se afişeze în ordine crescătoare valorile a 3 variabile a, b şi c.

Rezolvare: Putem rezolva această problemă comparând două câte două valorile celor 3 variabile. În cazul în care nu sunt în ordine crescătoare interschimbăm valorile lor.

```
start
citeste a,b,c;
daca (a > b) {
                  aux = a; a = b; b = aux;}
daca (a > c) {
                  aux = a; a = c; c = aux;}
daca (b > c) {
                  aux = b; b = c; c = aux;}
scrie a,b,c;
stop.
```

O altă variantă este următoarea, în care valorile variabilelor nu se modifică ci doar se afişează aceste valori în ordine crescătoare:

```
start
citeste a,b,c
daca (a < b si b < c) scrie a, b, c;
daca (a < c si c < b) scrie a, c, b;
```

```
daca (b < a si a < c) scrie b, a, c;
daca (b < c si c < a) scrie b, c, a;
daca (c < a si a < b) scrie c, a, b;
daca (c < b si b < a) scrie c, b, a;
stop.
```

Problema 4. Să se calculeze şi să se afişeze suma: S=1+1*2+1*2*3+..+n!

Rezolvare: Vom folosi o variabilă auxiliară, p, în care vom calcula produsul parțial.

```
start
citeste n;
s = 0;
pentru i de la 1 la n cu pasul 1 {
      p = 1;
      pentru j de la 2 la i cu pasul 1 {
            p = p * j;
      }
      s = s + p;
}
scrie s;
stop.
```

O altă posibilitate de calcul este următoarea, în care produsul parțial este actualizat la fiecare pas, fără a mai fi calculat de fiecare dată de la 1:

```
start
citeste n;
s = 0; p = 1;
pentru i de la 1 la n cu pasul 1 {
      p = p * i;
      s = s + p;
}
scrie s;
stop.
```

Problema 5. Să se calculeze şi să se afişeze suma cifrelor unui număr natural n.

Rezolvare: Vom folosi o variabilă auxiliară ***c*** în care vom calcula rând pe rând cifrele. Pentru aceasta vom obține ultima cifră a lui n ca rest al împărțirii lui n la 10, după care micşorăm n împărțindu-l la 10, astfel încât înurmătoarea iterație (pas al calculului repetitiv) să obținem următoarea cifră, ș.a.m.d.

```
start
citeste n;
s = 0;
atata timp cat n > 0 {
      c = n mod 10;
      s = s + c;
      n = n / 10;
}
scrie s;
stop.
```

Problema 6. Să se calculeze şi să se afişeze inversul unui număr natural n.

Rezolvare: Vom folosi o variabilă auxiliară ***c*** în care vom calcula rând pe rând cifrele ca în problema anterioară, şi vom construi inversul pe măsură ce calculăm aceste cifre.

```
start
citeste n;
inv = 0;
atata timp cat n > 0 {
      c = n mod 10;
      inv = inv * 10 + c;
      n = n / 10;
      }
scrie inv;
stop.
```

Problema 7. Să se afişeze un mesaj prin care să se precizeze dacă un număr natural dat ***n*** este prim sau nu.

Rezolvare: Pentru aceasta vom împărți numărul dat, pe rând, la numerele de la 2 până la radical din n (este suficient pentru a testa condiția de prim, după această valoare numerele se vor repeta). Dacă găsim un număr care să-l împartă exact vom seta o variabilă auxiliară b (numită *variabila flag,* sau *indicator*) pe 0. Ea are rolul de a indica faptul că s-a găsit un număr care divide exact pe n. Inițial presupunem că nu există un astfel de număr, şi deci, b va avea valoarea 1 inițial.

```
start
citeste n;
b = 1;
pentru d de la 2 la √n cu pasul 1
      daca (n mod d = 0)
            b = 0;
daca (b == 1)
      scrie "Numarul este prim";
altfel
      scrie "Numarul nu este prim";
stop.
```

Problema 8. Să se afişeze primele n numere naturale prime.

Rezolvare: Folosm algoritmul de la problema anterioară la care adăugăm o variabilă de contorizare /numărare, k.

```
start
citeste n;
k = 0;
i = 2;
atata timp cat k < n {
      b = 1;
      pentru d de la 2 la √i cu pasul 1
            daca (i mod d = 0)
                  b = 0;
      daca (b == 1) {
            scrie i;
            k = k + 1;
            }
      i = i + 1;
stop.
```

Problema 9. Să se descompună în factori primi un număr dat n.

Rezolvare: Pentru aceasta vom împărți numărul pe rând la numerele începând cu 2. Dacă găsim un număr care să-l împartă exact vom împărți pe n de câte ori se poate la numărul găsit, calculând astfel puterea. În modul acesta nu va mai fi necesar să testăm că numerele la care împărțim sunt prime!

```
start
citeste n;
div = 2;
scrie "x = ";
atata timp cat x != 0 {
      daca (x mod div = 0) {
            nr =0;
            atata timp cat x mod div = 0 {
                  x = x / div;
                  nr = nr + 1;
                  }
            scrie div,"^",nr,"+";
      }
      div = div + 1;
}
stop.
```

Problema 10. Să se afişeze toate numerele naturale mai mici decât 10000 care se pot descompune în două moduri diferite ca sumă de două cuburi.

Rezolvare: Această problemă prezintă foarte bine avantajul utilizării unui calculator în rezolvarea anumitor probleme. Calculăm, pe rând, suma de cuburi a perechilor de numere de la 1 la 21 (cel mai apropiat întreg de radical de ordin 3 din 10000). Căutăm apoi, pentru fiecare sumă, o a doua pereche a cărei sumă de cuburi este aceeaşi. Dacă este o pereche diferită de prima, afişăm numerele.

```
start
pentru a de la 1 la 21 cu pasul 1
      pentru b de la 1 la 21 cu pasul 1 {
            x = a * a * a + b * b * b;
            pentru c de la 1 la 21 cu pasul 1
                  pentru d de la 1 la 21 cu pasul 1
                        y = c * c * c + d * d * d;
                        daca (y = c si a != c si b != d)
                              scrie a, b, c, d;
                  }
stop.
```

Problema 11. Să se calculeze valoarea minimă, respectiv maximă, dintr-o secvență de n numere reale.

Rezolvare: Vom utiliza două variabile, max şi min pe care le inițializăm cu o valoare foarte mică şi respectiv cu o valoare foarte mare. Vom compara pe rând valorile citite cu max şi respectiv cu min, iar dacă găsim o valoare mai mare, respectiv mai mică decât acestea modificăm max (sau min, după cum e cazul) la noua valoare maximă, respectiv minimă.

```
start
citeste n;
```

```
min = +∞;
max = -∞;
pentru i de la 1 la n cu pasul 1 {
      citeste x;
      daca (x > max)
            max = x;
      daca (x < min)
            min = x;
      }
scrie "valoarea minima este:",min,"valoarea maxima este:",max;
stop.
```

Problema 12. Se dă o secvență de n numere întregi pozitive. Să se afişeze cele mai mari numere de 2 cifre care nu se află în secvența respectivă.

Rezolvare: În cadrul acestei probleme şi a următoarei vom folosi o structură numită vector. Aceasta este utilă atunci când trebuie să memorăm ca date mai multe valori de acelaşi tip. De exemplu, dacă avem mai multe valori întregi, putem folosi un vector de numere întregi. Numărul acestor valori poate fi variabil iar ele vor fi păstrate sub un acelaşi nume, la adrese consecutive de memorie. Accesarea unei anumite valori din structura vector se face folosind un index. Dacă numele vectorului este v, atunci pentru a accesa prima valoare din vector vom folosi notația v[0], pentru a doua valoare v[1], ş.a.m.d.
În rezolvarea acestei probleme vom folosi un vector ca variabilă auxiliară, în care vom memora dacă un număr de două cifre a fost citit de la tastatură sau nu (***v[nr]*** este 0 dacă ***nr*** nu a fost citit şi ***v[nr]*** devine 1 dacă ***nr*** a fost citit de la tastatură). Inițial, toate valorile din vector sunt 0.
Pentru a afla cele mai mari numere de două cifre care nu sunt în secvența citită vom parcurge vectorul v de la sfârşit (indexul 99) la început, până întâlnim două valori zero.
Atenție! În cazul în care nu există una sau două valori de două cifre care să nu aparțină secvenței citite nu se va afişa nimic!

```
start
citeste n;
pentru i de la 10 la 99 cu pasul 1
        v[i] = 0;
pentru i de la 1 la n cu pasul 1
        citeste nr;
        dacă (nr > 9 si nr < 100)
                v[nr] = 1;
i = 99;
atata timp cat(v[i] != 0 si i > 0)
        i = i - 1;
dacă (i > 9) atunci
        scrie i, " ";
i = i - 1;
atata timp cat (v[i] != 0 si i > 0)
        i = i - 1;
daca (i > 9) atunci
        scrie i, " ";
stop.
```

Problema 13. Se dă o secvență de n numere întregi, ale căror valori sunt cuprinse în intervalul 0-100. Să se afişeze valorile care apar cel mai des.

Rezolvare: Vom utiliza de asemenea un vector în care vom memora de câte ori a apărut fiecare valoare citită: v[nr] memorează de câte ori a fost citit nr. Inițial toate valorile din vector sunt 0. Vom

determina apoi valoarea maximă din acest vector, după care, pentru a afişa toate numerele cu număr maxim de apariţii mai parcurgem încă o dată vectorul şi afişăm indicii pentru care găsim valoarea maximă.

```
start
citeste n;
pentru i de la 0 la 100 cu pasul 1
        v[i] = 0;
pentru i de la 1 la n cu pasul 1
        citeste nr;
        v[nr] = v[nr] + 1;
max=0;
pentru i de la 0 la 100 cu pasul 1
        dacă (v[i] > max) atunci
                max = v[i];
pentru i de la 0 la 100 cu pasul 1
        dacă (v[i] == max) atunci
                scrie i;
stop.
```

Capitol IB.02. Introducere în limbajul C. Elemente de bază ale limbajului

● Probleme propuse şi rezolvate

1. Scrieţi cele două constante şir de caractere care conţin caracterele de mai jos:

```
a.
Informatii 100% corecte:
 I.Ionescu / 24 ani \ zis "a lu' Vasile"
b.
slash /; backslash \; procent%;
ghilimele "; apostrof '.
```

Rezolvare:

```
a."Informatii 100%% corecte:\n\I.Ionescu / 24 ani \\ zis \"a
   lu' Vasile\""
```

2. Care este spaţiul de memorie ocupat de constanta caracter şi cea şir: 'z' şi "z"?
Răspuns:
1 octet, respectiv 2 octeţi

3. Cum se defineşte corect o variabilă *suma* de tip întreg?
Răspuns:

```
int suma;
```

4. Cum se defineşte o constantă simbolică având numele TRUE şi valoarea 1?
Răspuns:

```
#define TRUE 1
```

5. Folosind operatorul condiţional, sa se determine maximul dintre a,b, relatia dintre a,b şi maximul dintre a,b,c.

Rezolvare:

```
max_a_b=a>b?a:b;
char                                    c=a>b?'>':a<b?'<':'=';
max_a_b_c=a>b?(a>c?a:c):(b>c?b:c);
```

6. Adăugați comentariile legate de conversiile implicite şi explicite pentru secvența următoare:

```
char c='a',cc;
int i=4;
float f=5.95;
i=f;
f=i+100000;
i=-99.001;
f='a';
c=0x3239; cc=-i;
float r1=5/2,
r2=(float)5/2,
r3=(float)(5/2),
r4=5/(float)2,
r5=(float)5/(float)2;
```

Rezolvare:

```
char c='a', cc;
int i=4;
float f=5.95;
i=f;        // conversie implicita, trunchiere
f=i+100000; // conversie implicita a rezultatului expresiei
i=-99.001; // conversie implicita, trunchiere
f='a';
c=0x3239; cc=-i;
float  r1=5/2,
r2=(float)5/2,          // conversie explicită
r3=(float)(5/2),        // conversie explicită
r4=5/(float)2,          // conversie explicită
r5=(float)5/(float)2;   // conversie explicită
```

● **Probleme propuse**

1. Care este valoarea variabilei *real* definită double real=26/4?

2. Dacă avem următoarea secvență:

```
int a=29,b=7;
a%=b--;
```

care va fi valoarea lui a?

3. Dacă avem următoarea secvență:

```
int i=1;
char c='1';
```

care va fi valoarea expresiei i<c?

4. Dacă avem următoarea secvență:

```
int x=-1,y;
y=++x?5:7;
```

care va fi valoarea lui y?

5. Găsiți valorile expresiilor de mai jos.

```
a. sizeof(int)
b. sizeof(double)
c. sizeof(long double)
d. sizeof('a')
e. sizeof((char)'a')
f. sizeof(33000)
g. sizeof(1.3+'a')
h. sizeof(1.3f)
i. sizeof(1.3)
j. sizeof(1.3l)
k.  sizeof(5/2)
l. sizeof((float)5/2)
```

6. Spuneți care sunt valorile variabilelor în timpul execuției următoarei secvențe:

a.

```
int i=20000,j=15000,k;
float r;
k=i+j;
r=i+j;
r=(float)i+j;
r=(long)i+j;
r=i/j;
k=i/j;
r=(float)i/j;
k=i%j;
k=+ - - -i;
k=+ - --i;
k=- +j--;
i=k/(j-j);
```

b.

```
int a,b,c; float z;
a=25000;b=20000;
c=a+b;
z=a+b;
c=(float)a+b;
z=(float)a+b;
c=a/b;
c=a%b;
c=a>b;
c=a==b;
a=3;b=11;
c=a++ + ++b;
c=a&b;
c=a|b;
c=a^b;
c=b<<2;
c=-a>>3;
c=~a;
```

```
a=0;
c=a&&b;
c=a||b;
c=!a;
a=4;
c=a&&b;
c=a||b;
c=!a;
a=2;c=3;
c*=a;
b=5;
c=(a>b)?a:b;
```

7. Să se scrie o secvență care determină maximul dintre a şi b, iar daca a<b, le interschimbă, astfel încât secvența a şi b să fie descrescătoare, utilizând operatorul condițional.

Răspuns: `max_a_b=a<b?(t=b,b=a,a=t):a;`

8. Pentru un n dat să se scrie secvența care calculează valorile mai mari/mai mici decat n de 2,4,8,16 ori obținute prin înmulțiri/împărțiri, respectiv deplasări.

9. Se citesc întregii x, y, a, b (0<=a, b<16). Se cere să se scrie secvențele care calculează:
bitul (a+b) din x
- **a.** valoarea obținută prin setarea biților a şi b din x
- **b.** valoarea obținută prin stergerea biților a şi b din x
- **c.** valoarea obținută prin inversarea biților a şi b in x
- **d.** valoarea obținută prin negarea, respectiv complementarea lui y
- **e.** valorile obtinute prin aplicarea lui x şi y a operatorilor şi logic, sau logic, şi pe biți, sau pe biți, sau exclusiv pe biți.

10. Fiind date următoarele definiții:

```
int i = 3, j = 5,c1,c2,c3,c4;
```

determinați valorile tuturor variabilelor, după execuția secvenței:

```
c1=(i/2) + 'b' + i-- - - - 'c';
c2=(j%8) * i;
c3=(i++) - (--j);
c4= j = (i += 2);
```

11. Fiind date definițiile:

```
int a=2, b=2, c=1, d=0, e=4, i = 2, j = 4;
```

determinați valorile următoarelor expresii:

```
a. a++ / ++c * --e
b.  --b * c++ -a
c.  -b - --c
d.  e / --a * b++ /c++
e.  e / --a * b++ / c++
f.  a %= b = d = 1 + e /2
g.  j = (i++ , i -j)
```

12. Se citesc 2 numere întregi x şi n unde n este între 0 şi 15. Să se afişeze:
a. bitul n din x
b. numărul x în care se setează pe 1 bitul n
c. numărul x în care se şterge bitul n.

Aplicații module 3-12

Pentru fiecare din următoarele laboratoare există

- **Probleme propuse**
- **Rezolvările problemelor propuse**
- **Evaluator al soluțiilor trimise de student**

În plus, a fost necesară crearea unui tutorial de utilizare a site-ului şi a evaluatorului: http://elf.cs.pub.ro/programare/

Capitol IB.03. Funcții de intrare/ieşire în limbajul C
Probleme propuse şi rezolvate
http://elf.cs.pub.ro/programare/runda/lab-02

Capitol IB.04. Instrucțiunile limbajului C
Probleme propuse şi rezolvate
http://elf.cs.pub.ro/programare/runda/lab-03

Capitol IB.05. Tablouri. Definire şi utilizare în limbajul C
Probleme propuse şi rezolvate
http://elf.cs.pub.ro/programare/runda/lab-04

Capitol IB.06. Funcții. Definire şi utilizare în limbajul C
Probleme propuse şi rezolvate
http://elf.cs.pub.ro/programare/runda/lab-05
http://elf.cs.pub.ro/programare/runda/lab-06

Capitol IB.07. Pointeri. Pointeri şi tablouri. Pointeri şi funcții
Probleme propuse şi rezolvate
http://elf.cs.pub.ro/programare/runda/lab-07

Capitol IB.08. Şiruri de caractere. Biblioteci standard
Probleme propuse şi rezolvate
http://elf.cs.pub.ro/programare/runda/lab-07

Capitol IB.09. Structuri de date. Definire şi utilizare în limbajul C
Probleme propuse şi rezolvate
http://elf.cs.pub.ro/programare/runda/lab-09

Capitol IB.10. Alocarea memoriei în limbajul C
Probleme propuse şi rezolvate
http://elf.cs.pub.ro/programare/runda/lab-08

Capitol IB.11. Operații cu fişiere în limbajul C
Probleme propuse şi rezolvate
http://elf.cs.pub.ro/programare/runda/lab-011
http://elf.cs.pub.ro/programare/runda/lab-012

Capitol IB.12. Convenții şi stil de programare
Probleme propuse şi rezolvate
http://elf.cs.pub.ro/programare/runda/lab-10
http://elf.cs.pub.ro/programare/runda/lab-13

Bibliografie

[B01] Gary J. Bronson, *Program Development and Design using C++*
[D01] Deitel & Deitel, *C++ How to Program*
[K01] Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*
[M01] Moraru F., *Programarea Calculatoarelor*, editura Bren, 2005
[M02] Moraru F., *Programarea Calculatoarelor (culegere)*, editura Bren, 2005
[N01] Negrescu L, *Limbajele C şi C++ pentru începători*, volumul 1: Limbajul C, Ed. Albastră, Cluj-Napoca, 2002
[P01] Plauger, *The Standard C Library*, PrenticeHall, 1992
[S01] Stroustrup B., *The C++ Programming Language*
[S02] Stroustrup B., *The Design and Evolution of C++*
[W01] http://www.eskimo.com/~scs/cclass/notes/top.html
[W02] http://www.eskimo.com/~scs/cclass/int/top.html
[W03] http://cermics.enpc.fr/~ts/C/cref.html
[W04] http://www.cs.bath.ac.uk/~pjw/NOTES/ansi_c/
[W05] http://www.chris-lott.org/resources/cstyle/
[W06] http://www.infoiasi.ro/fcs/absolvire4info.html
[W07] http://www.timsoft.ro/aux/module.shtml
[W08] http://www3.ntu.edu.sg/home/ehchua/programming/#Cpp
[W09] http://www.cplusplus.com, *C++ documents*, *tutorials*, *library references*
[W10] curs.cs.pub.ro, *Programarea Calculatoarelor*, seria 1CC