Considerații teoretice

Formatori:
Tutor: Stângaciu Valentin
Tutor: Belu Claudiu-Marcel
+3

Data de începere a cursului:
25.09.2023
Utilizatori înscriși
Calendar
Note

Considerații teoretice

Pointerii sunt variabile care au ca valori ADRESE DE MEMORIE. De la acest aspect le vine și numele, „to point" însemnând „a indica" sau „a arăta", adică variabila de tip pointer *pointează/indică/arată* către o valoare a cărei adresă este stocată chiar în pointerul respectiv. Pointerii pot conține adrese de memorie care sunt alocate unor variabile, dar pot conține și adrese de memorie care au alte semnificații (zone de cod binar, stivă, ...) sau chiar locații care nu au fost atribuite de către compilator.
Pointerii reprezintă unul dintre punctele forte ale limbajului C și ei se folosesc în foarte multe situații: ca iteratori, la transmiterea parametrilor prin adresă, alocare dinamică, structuri de date dinamice, etc. Tocmai din cauza puterii lor, la început pointerii pot fi mai greu de stăpânit și lucrul cu ei necesită o anumită practică. În schimb, după ce ne-am obișnuit să folosim pointerii, vom descoperi că ei ne simplifică multe alte sarcini de programare, în special când vom lucra cu structuri complexe de date sau când dorim să optimizăm cât mai mult o aplicație. În acest laborator vom face o introducere în pointeri și vom discuta folosirea lor ca iteratori. În laboratoarele viitoare vom continua cu prezentarea altor aplicații cu pointeri.

Prin intermediul pointerilor avem acces la toată memoria unui program, inclusiv la programul în sine. Sistemele moderne de operare impun o separare între programele active în memorie, iar dacă un program încearcă să acceseze memoria altui program (ca urmare a unei erori de programare sau intenționat, cu intenții distructive, gen viruși), sistemul de operare va elimina automat din memorie programul care a încercat acel acces la o memorie care nu îi aparține. În acest caz, utilizatorul va primi o eroare de tipul "*segmentation fault*". Aceasta semnifică faptul că programul a încercat să acceseze o adresă de memorie pe care nu are dreptul să o acceseze. Dacă primim asemenea erori în programele noastre, aproape sigur ele provin de la folosirea greșită a pointerilor, care fie nu au fost inițializați corect, cu o adresă validă de memorie (și deci sunt folosiți cu adrese aleatoare de memorie, care pot fi protejate), fie, ca urmare a unor operații cu pointeri, aceștia ajung să pointeze către zone protejate de memorie.

Un pointer se defineste astfel:

```
tip_pointat *numePtr;
```

și se citește: "*numePtr este o variabilă de tip pointer la tipul tip_pointat*". De exemplu "*int *p;*" declară variabila *p* ca fiind un pointer la o valoare de tip *int*. Cu alte cuvinte, *p* conține o adresă de memorie unde se află stocată o valoare de tip *int*.

Astfel, când folosim un pointer, avem acces la două valori:

adresa de memorie conținută de pointer - este chiar valoarea acelui pointer și se accesează simplu, folosind numele pointerului: *p*

valoarea pointată de pointer - este valoarea de la adresa de memorie conținută de pointer și se accesează cu steluță (*****) în fața pointerului: **p*

În C, **steluța** (*****) are 3 semnificații:

a * b - când este folosită ca operator binar, steluța este operatorul de înmulțire, la fel ca în matematică

float *p; - când este folosită într-o declarație de variabilă, steluța semnifică faptul ca variabila respectivă este un pointer la tipul specificat

a = *p; - când este folosită ca operator unar prefixat, steluța are semnificația de "**valoarea pointată de**" și este folosită pentru a accesa valoarea de la adresa de memorie conținută în pointer. În această situația, steluța este numită **operatorul de dereferențiere**.

Pentru a inițializa un pointer, trebuie să îi atribuim o adresă. În C, adresa unei variabile se poate obține folosind **ampersand** (**&**) ca operator unar prefixat:

```c
#include <stdio.h>

int main()
{
    int a=7,b=1;
    int *p=&a;        // pointerul p este initializat cu adresa variabilei a
    int *r=p;          // pointerul r este initializat cu adresa care este stocata in pointerul p
    p=&b;             // pointerului p i se atribuie adresa variabilei b
    // in acest punct pointerul p va contine adresa variabilei b
    // iar pointerul r va contine adresa variabilei a

    printf("&a=%p\n&b=%p\n",r,p);// in C se poate afisa adresa continuta de un pointer folosind %p (afișarea are loc în hexazecimal)
    return 0;
}
```

Operatorul **ampersand** (**&**) are 2 semnificații:

a & b - când este folosit ca operator binar, înseamnă **ȘI pe biți**, așa cum a fost discutat în laboratoarele anterioare

&a - când este folosit ca operator unar prefixat, înseamnă "**adresa lui**" și se folosește pentru a obține adresa unei variabile. În acest caz, *&a* înseamnă "*adresa variabilei a*"

Notă: În C++, operatorul ampersand (&) mai are și o a treia semnificație, atunci când este folosit la declararea unei variabile: "*int &x;*". În acest context, *x* este declarat ca o variabilă de tipul "*referință la int*". Această semnificație este valabilă doar în C++, nu și în C.

În exemplul anterior, se constată că am folosit operatorul *&* pentru obține adresa unei variabile și a inițializa un pointer cu această adresă. Operatorul *&* se poate folosi oriunde într-o expresie. De exemplu "*printf("%p",&a);*" va afișa adresa lui *a*.

Adresa unei variabile poate să difere în funcție de sistemul de operare sau de alți factori. Din acest motiv, același program, rulat pe calculatoare diferite, poate afișa adrese diferite. Vom discuta ulterior mai detaliat cum alocă compilatorul memorie pentru variabile.

Atenție: dacă declarăm doi pointeri folosind o singură instrucțiune, fiecare pointer trebuie să aibă propria sa steluță (*):

```c
int *p,*r;        // corect: se definesc 2 pointeri
int *p,r;         // greșit: p va fi pointer la int, iar r va fi o variabilă de tip int
```

După ce am inițializat un pointer cu adresa unei variabile, îl putem folosi pentru a accesa valoarea de la acea adresă. Prin intermediul pointerului vom putea să modificăm acea valoare, ca și când am fi operat chiar cu variabila respectivă. Acest gen de operare asupra unei variabile, prin intermediul unui pointer, se numește *adresare indirectă*.

```c
#include <stdio.h>
int main()
{
    int x = 10, y = 7;        // variabile de tip int
    int *p = &x;               // p va contine adresa lui x
    printf("%d\n", *p);      // afișează valoarea pointata de p. Din cauză ca p pointeaza la x, va afișa 10
    p = &y;                    // p va contine adresa lui y
    *p = *p - 2;               // va modifica valoarea de la adresa lui y (la care pointeaza p), scazand 2 din ea
    printf("%d\n", y);        // va afisa 5, deoarece valoarea lui y a fost modificata anterior prin intermediul lui p
    y = 1;                         // modifica valoarea lui y
    printf("%d\n", *p);      // va afisa 1, deoarece la adresa pointata de p (adresa lui y) se afla acum valoarea 1
    printf("%p\n", p);       // va afișa adresa de memorie la care se afla y
    scanf("%d", p);         // citeste o valoare si o depune la adresa continuta in p (va fi noua valoare a lui y)
    return 0;
}
```

Se poate constata că folosind adresarea indirectă, prin intermediul unui pointer, putem să efectuăm orice operații asupra valorii de la o adresă de memorie, ca și când am fi folosit chiar variabila stocată la acea adresă de memorie. Mai mult decât atât, putem avea oricâți pointeri care să pointeze la aceeași zonă de memorie și să-l folosim pe oricare dintre ei pentru a opera cu valoarea stocată acolo.

Observație: Funcția *scanf* necesită ca parametri adrese (vom vedea la laboratorul despre funcții de ce). Din acest motiv, când se citește valoarea unei variabile, îi dăm lui *scanf* adresa acelei variabile *"scanf("%d",&a);"*. Deoarece un pointer conține chiar adresa unei variabile, dacă dorim să citim cu *scanf* o valoare la adresa conținută de un pointer, vom scrie direct *"scanf("%d",p);"*, fără a mai cere cu operatorul & adresa variabilei.

Aritmetica pointerilor

O adresă de memorie de fapt este un număr întreg, pozitiv. În general, într-un calculator, adresele încep de la indexul 0 și acoperă toată memoria disponibilă. Din acest punct de vedere, putem considera un pointer ca fiind un număr întreg, număr asupra căruia putem efectua anumite operații. Acest set de operații poartă numele de *aritmetica pointerilor*.

Adunarea/scăderea unei valori întregi la un pointer

Fie *k* un număr întreg (*int k*) și *p* un pointer la valori de tip *double* (*double *p*). În *k* putem avea orice valoare, inclusiv negativă, dar în exemplele viitoare vom considera *k* ca fiind pozitiv. Pentru *k* negativ, doar se inversează operația: adunarea unui număr negativ este de fapt o scădere cu modulul numărului, iar scăderea unui număr negativ este o adunare cu modulul numărului.

În aceste condiții, îl putem aduna pe k la p (p+k), cu următoarea semnificație: *p+k este adresa care se află la o distanță de k celule de memorie, fiecare celulă având tipul double, în sensul creșterii adreselor de memorie, față de adresa stocată în p.*

De exemplu, dacă adresa stocată în p este 65280 în zecimal (0xFF00 în hexazecimal) și k=3, *p+k* va fi *65280+3*sizeof(double)* = 65304 (0xFF18), considerând că *double* este reprezentat în memorie pe 8 octeți. Reamintim că operatorul *sizeof(expr/tip)* returnează dimensiunea în octeți a expresiei/tipului date dat ca argument.

Analogic, *p-k este adresa care se află la k celule de memorie, fiecare celulă având tipul double, în sensul descreșterii adreselor de memorie, față de adresa stocată în p.* Pentru valorile de mai sus, *p-k=65280-3*sizeof(double)=65256 (0xFEE8)*.

Se poate constata că **la adunarea sau la scăderea unei constante dintr-un pointer, se consideră acea constantă ca fiind un număr de celule de memorie, fiecare celulă de memorie având tipul pointat de pointer.**

Dacă pointerul *p* ar fi pointat către tipul *char* (*char *p*), atunci *p+k* ar fi însemnat o adresă situată la *k* octeți de memorie (k celule de câte un octet) de adresa conținută în *p*, deoarece de obicei *sizeof(char)==1*, dar, la modul general, pentru a face corect operația *p+k*, trebuie ținut cont de tipul valorii pointate de *p*.

În particular, incrementarea/decrementarea unui pointer este echivalentă cu modificarea adresei conținută de acesta cu o celulă de memorie de tipul pointat, la dreapta/stânga. De exemplu, pentru "*double *p*", *p++* va face ca *p* să conțină adresa următoarei celule de memorie, situată cu 8 octeți (*sizeof(double)*) la dreapta celei inițiale.

Adunarea/scăderea folosind celule de memorie, conform dimensiunii tipului pointat de pointer, simplifică foarte mult folosirea pointerilor ca iteratori. Altfel, dacă s-ar fi adunat/scăzut octeți indiferent de tipul pointat, dacă am fi dorit sa trecem la următoarea locație de memorie, ar fi trebuit să scriem *p+sizeof(tip)*, în loc de *p+1*.

Operatorul de dereferențiere (**p*), are o precedență mai mare decât operațiile aritmetice. Astfel, **p+1* se interpretează *(*p)+1*. În schimb, incrementarea și decrementarea au o precedență mai mare decât dereferențierea, astfel încât **p++* se interpretează ca **(p++)*.

Scăderea a doi pointeri

Fie *p* și *r* doi pointeri la valori de tip *double* (*double *p,*r*). Scăderea a doi pointeri are următoarea semnificație: *n=p-r este numărul întreg, cu semn, de celule de memorie care se află între adresele pointate de cei doi pointeri, fiecare celulă fiind de tipul pointat de pointeri.*

De exemplu, dacă r=65280 (0xFF00) si p=65296 (0xFF10), *p-r=>2*, deoarece între *r* și *p* sunt două celule de memorie pentru valori de tip *double*. Dacă am fi efectuat r-p, am fi obținut -2, deoarece *r<p* (*p* conține o adresă de memorie situată la dreapta față de *r*).

De fapt, la scăderea a doi pointeri, compilatorul aplică următoarea formulă: *p-r == ((char*)p-(char*)r)/sizeof(*p)*, adică scade adresele celor doi pointeri exprimate în octeți, după care împarte valoarea obținută la tipul pointat de pointeri, pentru a obține numărul de celule de acest tip.

Scăderea a doi pointeri se folosește pentru a afla distanța dintre ei, exprimată în celule de memorie având tipul pointat de pointeri.

Comparații între pointeri

Dacă avem doi pointeri, *p* și *r*, putem aplica asupra lor toți operatorii de comparație (*p==r, p!=r, p<r, p<=r, p>=r, p>r*). Acești operatori vor compara adresele de memorie conținute de pointeri (dacă am fi vrut să comparăm valorile pointate de ei, ar fi trebuit să scriem de exemplu **p==*r*).

Un pointer se consideră ca având o valoare mai mică decât alt pointer (p<r), dacă el conține o adresă de memorie mai mică (situată mai aproape de 0 - originea adreselor de memorie / mai la stânga) decât celălalt pointer.

Prin convenție, dacă un pointer conține valoarea 0 (NULL), se consideră că el nu pointează la nicio valoare. În C există definiția **NULL**, care se reduce exact la adresa de memorie 0, astfel încât putem folosi *NULL* pentru a inițializa sau testa pointeri. Multe funcții din biblioteca standard C folosesc valoarea *NULL* tocmai pentru a indica faptul că pointerul nu pointează la nicio valoare. De exemplu, funcția *strchr(s,c)*, care caută caracterul *c* în șirul de caractere *s*, dacă găsește acel caracter, va returna un pointer la adresa unde l-a găsit, iar dacă nu-l găsește, va returna *NULL*.

Atenție: limbajul C nu garantează că două variabile declarate consecutiv vor ocupa locații consecutive de memorie. De exemplu, dacă avem "*int a,b*", variabila *b* poate fi sau nu la locația consecutivă locației lui *a* (nu se garantează că *&a+1 == &b*).

Pointeri și vectori

În C există o identitate aproape completă între pointeri și vectori. În marea majoritate a cazurilor, limbajul C consideră vectorii ca fiind niște pointeri la o zonă prealocată de memorie, unde se găsesc elementele vectorului. Reciproc, pointerii sunt considerați ca fiind vectori cu o lungime nedefinită, care încep în memorie la adresa conținută de pointeri. Următoarele construcții sunt valide în C:

```c
#include <stdio.h>

int main()
{
    int v[10];
    int *p;
    p=v;                                                // v este considerat ca pointer si i se atribuie lui p adresa de
la care incep elementele sale
    printf("%d\n",p[1]);                          // operatorul de indexare se poate aplica pointerilor; echivalent cu *
(p+1)
    printf("%d\n",*(v+1));    // operatorul de dereferentiere se poate aplica vectorilor; echivalent cu v[1]
    p++;                                                // p va pointa la urmatorul element din v: v[1]
    printf("%d\n",&v[1]==p);            // !=0 ; cele doua adrese sunt identice
    printf("%d\n",v+1==p);             // un alt fel de a scrie identitatea de mai sus
    printf("%d\n",p[-1]);              // echivalent cu *(p-1), adica v[0]
    printf("%d\n",p-v);                         // => 1 ; aritmetica pointerilor se poate aplica vectorilor sau intre
vectori si pointeri
    return 0;
}
```

Întotdeauna elementele unui vector ocupă locații consecutive de memorie, în ordinea indecșilor, fără spații între ele. Din acest motiv, când s-a incrementat *p* (*p++*) acesta a pointat către următoarea valoare din vector, v[1]. Astfel, pointerii pot fi utilizați ca iteratori în vectori.

Cu declarațiile de mai sus, diferențele cele mai semnificative dintre pointeri și vectori sunt:

```c
v++;                                        // EROARE: adresa vectorilor este fixă, spre deosebire de pointeri, cărora li
se poate schimba adresa conținută de ei
printf("%d\n",sizeof(v));          // 40 == 10*sizeof(int) ; pentru vectori, sizeof returnează dimensiunea totală în
octeți a întregului vector
printf("%d\n",sizeof(p));          // 8 (la sisteme de operare pe 64 de biți) ; pentru pointeri, operatorul sizeof
returnează dimensiunea unei adrese de memorie
```

Exemplu: Un program care citește valori până când întâlnește 0 (exclusiv), după care afișează aceste valori în ordine inversă. Programul va fi implementat atât cu indecși, cât și cu pointeri, pentru a se ilustra diferențele dintre cele două moduri de abordare.

```c
// varianta cu indecsi
#include <stdio.h>

int main()
{
    int v[10];
    int i,j,n,tmp;
    n=0;                                    // n indica numarul de elemente din vector
    for(;;){                                 // bucla infinita pentru citire
        printf("v[%d]=",n);
        scanf("%d",&v[n]);            // citeste direct in vector, dar nu va considera valoarea decat deca este !=0
        if(v[n]==0)break;               // daca s-a introdus 0, iesire din bucla
        n++;                                    // considera valoarea citita ca fiind introdusa
        }
    for(i=0, j=n-1 ; i<j ; i++, j--){  // inversare valori
        tmp=v[i];
        v[i]=v[j];
        v[j]=tmp;
        }
    for(i=0;i<n;i++){                    // afisare valori
        printf("%d\n",v[i]);
        }
    return 0;
}
```

Problema implementată folosind doar pointeri, fără niciun index:

```c
// varianta cu pointeri
#include <stdio.h>

int main()
{
    int v[10];
    int *end,*p,*r;
    int tmp;
    end=v;                                          // end pointeaza la prima pozitie libera din vector
    for(;;){                                           // bucla infinita pentru citire
        printf("v[%d]=",end-v);                 // nr de elemente deja introduse in vector
        scanf("%d",end);                      // citeste direct in vector, dar nu va considera valoarea decat deca este !=0
        if(*end==0)break;                        // daca s-a introdus 0, iesire din bucla
        end++;                                          // trece la urmatoarea adresa
    }
    for(p=v, r=end-1 ; p<r ; p++, r--){      // inversare valori
        tmp=*p;
        *p=*r;
        *r=tmp;
    }
    for(p=v;p<end;p++){                         // afisare valori
        printf("%d\n",*p);
    }
    return 0;
}
```

Se poate constata că cele două programe sunt destul de asemănătoare. De fapt, în general la iterare, indecșii au aceeași putere descriptivă ca și pointerii, deci putem scrie un program în oricare dintre cele două variante. Diferența este că în general varianta cu pointeri este mai rapidă, deoarece programul are de executat mai puține operații:

v[i] necesită citirea valorii lui *i*, înmulțirea valorii citite cu *sizeof(tip_element)*, adunarea rezultatului la adresa lui *v* și citirea valorii de la adresa rezultată

***p** necesită citirea adresei conținută în *p* și apoi citirea valorii de la acea adresă

Chiar dacă compilatorul reușește în multe cazuri să optimizeze folosirea indecșilor (ajungând la o variantă asemănătoare celei cu pointeri) totuși, la iterațiile mai complexe, folosirea directă a pointerilor depășește optimizările de care este capabil compilatorul. În biblioteca standard a limbajului C++ se folosesc pentru iterare aproape exclusiv iteratori, care sunt de cele mai multe ori doar clase care abstractizează pointeri. Vom prezenta în laboratoarele viitoare aplicații cum ar fi alocarea dinamică, în care pointerii sunt absolut necesari, ei neputând fi înlocuiți prin alte construcții.

Funcții cu pointeri

Transmiterea argumentelor prin valoare și prin adresă

Când se apelează o funcție, valorile expresiilor date ca argumente se vor copia în parametrii funcției și apoi funcția va folosi aceste copii ale valorilor originare. Acest mod de apel se numește **transmitere prin valoare**. Din acest motiv, chiar dacă modificăm valoarea unui parametru în interiorul funcției, această modificare se efectuează doar pe copia valorii cu care a fost inițializat acel parametru, fără ca valoarea originară să fie afectată.

Exemplu: Să se scrie o funcție *swap(x,y)*, care să interschimbe valorile variabilelor date ca parametri.

Varianta 1: *swap1* **nu** implementează corect cerința din exemplu!!!

```c
void swap1(int x,int y)
{
    int tmp=x;
    x=y;
    y=tmp;
}

int main()
{
    int a=5,b=7;
    swap1(a,b);
    printf("%d %d\n",a,b);// 5 7
    return 0;
}
```

Într-o primă variantă încercăm să scriem o funcție *swap1*, care are 2 parametri pe care-i interschimbă. Apelăm această funcție cu două variabile *a* și *b*. Ne-am aștepta ca după apelul funcției, valorile lui *a* și *b* să fie interschimbate, iar pe ecran să se afișeze "7 5". Cu toate astea, se vor afișa tot valorile inițiale, deci funcția *swap1* nu le-a interschimbat.

De ce *swap1* nu a interschimbat valorile lui *a* și *b*? Răspunsul constă chiar în transmiterea parametrilor prin valoare. În acest caz, la apelul lui *swap1(a,b)*, valorile lui *a* și *b* au fost copiate în *x* și *y*. Ulterior, în timpul execuției lui *swap1*, când *x* și *y* au fost interschimbate, de fapt s-au interschimbat valorile din aceste două variabile, fără ca originalele *a* și *b* să fie afectate în vreun fel.

Pentru a implementa funcții care modifică valorile originare ale variabilelor cu care au fost apelate, putem folosi **transmiterea prin adresă**. Pentru aceasta, vom transmite funcției adresele variabilelor, iar funcția va folosi pointeri pentru a opera cu valorile de la aceste adrese. În acest fel, orice modificare operată asupra valorii de la o anumită adresă, va modifica de fapt chiar variabila originară, deoarece acea variabilă se află amplasată la adresa respectivă din memorie.

Varianta 2: *swap* folosește transfer prin adresă și astfel reușește să interschimbe valorile variabilelor *a* și *b*

```
void swap(int *x,int *y)           // swap are ca parametri doi pointeri, deci două adrese de memorie
{
    int tmp=*x;                         // se operează asupra valorilor pointate de cei doi pointeri
    *x=*y;                                 // (valorile de la adresele stocate în x și y)
    *y=tmp;
}

int main()
{
    int a=5,b=7;
    swap(&a,&b);                     // transmitere prin adresa: funcția swap primește adresele variabilelor a și b
    printf("%d %d\n",a,b);// 7 5
    return 0;
}
```

Folosind transmiterea prin adresă, am reușit să implementăm funcția *swap*. Cu acest mod de transmitere, putem implementa orice funcție care trebuie să modifice valorile parametrilor săi. Putem astfel înțelege de ce funcția *scanf* are nevoie de adresele variabilelor în care va depune valorile citite (*ex: scanf("%d",&a);*): funcția *scanf* folosește adresele variabilelor pentru a ști unde anume în memorie să depună valorile citite. Altfel, dacă am fi încercat să scriem o funcție *scanf* fără pointeri, la apelul ei s-ar fi copiat valorile variabilelor în parametrii funcției, citirea s-ar fi făcut în aceste copii, iar variabilele originare ar fi rămas neschimbate.

Folosind transmiterea prin adresă, putem scrie funcții care returnează mai multe valori, nu doar una, ca în cazul folosirii lui *return*. Pentru aceasta, vom include în funcție câte un nou parametru pentru fiecare valoare returnată, parametru care va folosi transmitere prin adresă. La apelul funcției, acești parametri vor fi inițializați cu adresele variabilelor destinație pentru valorile returnate. În funcție se vor folosi parametrii respectivi pentru a seta la adresele variabilelor destinație valorile care trebuie returnate.

Transmiterea vectorilor și ca argumente

Când o funcție primește ca argument un vector, ea va primi de fapt adresa acelui vector. În acest caz, vectorul este interpretat ca pointer și se transmite funcției doar adresa lui, fără a se copia niciun element. Deci vectorii se transmit implicit prin adresă. Din acest motiv, orice modificare efectuată în funcție asupra valorilor unui vector, de fapt se va efectua chiar asupra vectorului originar.

Deoarece pointerii nu știu numărul de elemente care se află în memorie începând de la adresa pointată de ei, rezultă că la transmiterea vectorilor va trebui să transmitem funcției încă un parametru suplimentar, în care să specificăm numărul de elemente din vector.

Exemplu: Să se scrie o funcție care primește ca parametru un vector de tip *float* și numărul său de elemente și returnează 1 (true) dacă toate valorile sunt pozitive sau 0 (false) dacă există și valori negative în vector. Se va testa funcția folosind un program care cere un n<=10 și apoi *n* elemente de tip *float*.

```c
#include <stdio.h>
int pozitive(float v[], int n)                    // alternativ: float *v
{
    int i;
    for (i = 0; i < n; i++){
        if (v[i] < 0)                                  // daca a gasit un element negativ
            return 0;                                 // iese imediat din functie returnand 0
    }
    return 1;                                         // daca nu a gasit niciun element negativ
}
int main()
{
    int n, i;
    float v[10];
    printf("n=");
    scanf("%d", &n);
    for (i = 0; i < n; i++){
        printf("v[%d]=", i);
        scanf("%g", &v[i]);
    }
    if (pozitive(v, n)){
        printf("toate elementele sunt pozitive");
        }else{
        printf("exista si elemente negative");
    }
    return 0;
}
```

Vectorii transmiși ca argumente se scriu fără să se specifice dimensiunea lor (ex: *float v[]*). Se iterează vectorul, și dacă se găsește un element negativ se revine imediat din funcție, folosind *„return 0;"*, pentru că în acest caz nu mai are rost să se continue execuția funcției. Dacă s-a terminat instrucțiunea *for*, înseamnă că nu s-a descoperit niciun element negativ și atunci în final se returnează 1. În programul principal, valoarea returnată de funcție se folosește direct în *if*, pe baza faptului că în C „0" înseamnă *false* și orice altă valoare înseamnă *true*.

Notă: Dacă o aplicație cere să nu se folosească variabile index, nu se vor folosi construcții de forma *v[i]*, deoarece *i* este o variabilă folosită ca index. Se poate folosi însă operatorul de indexare aplicat pointerilor (ex: *p[-1]*), deoarece în această construcție nu există nicio variabilă index, ci doar constanta *-1* folosită ca index.

◄ Test evaluare 1

Sari la...

Teme și aplicații ►

Contactați serviciul de asistență

Sunteți conectat în calitate [redacted]
S1-L-AC-CTIRO1-PC

Meniul meu
Profil
Preferinte
Calendar
ZOOM
Română (ro)
English (en)
Română (ro)

Rezumatul păstrării datelor
Politici utilizare site