LUCRAREA 7

Scopul lucrării *este acela de a aprofunda studiul vectorilor început în* ***Lucrarea 6*** *prin utilizarea funcţiilor de bibliotecă asociate vectorilor de caractere precum şi introducerea în analiza* **pointerilor**, *subiect esenţial pentru limbajul C.*

I. OBSERVAŢII TEORETICE

1.1. Vectori (*continuare*)

1.1.1. Vectori de caractere = şiruri de caractere

Limbajul C nu aduce implicit tipul de dată şir, şi deci un şir trebuie declarat sub o altă formă.

Reamintim că tripletul (mulţimea cu trei elemente) care defineşte complet un vector este dat de:

{tip dată, nume vector, număr elemente}

Tipul de bază al unui vector poate fi şi *char*. În această situaţie vectorul se poate numi *vector de caractere*, sau, consacrat, ***şir de caractere***. Se preferă denumirea scurtă, **şir**.

Exemplu:

char sir1[20] ; // definiţia unui şir cu lungime 20.

Dintre *aplicaţiile* în care şirul de caractere joacă un rol esenţial, amintim:

- clasificarea tip dicţionar;
- aplicaţii cu **liste** de abonaţi (telefonici, întreţinere ş.a.) ;
- compresia de date
- ş.a.

Există însă operaţii asociate şirurilor: copiere, căutare de caractere sau şir în alt şir, comparaţie. Toate acestea au funţii predefinite, în header-ul *string.h*.

Observaţie:

După cum am arătat, pentru şiruri există un întreg fişier *header*, dedicat. Aceasta înseamnă că acest tip de dată derivat (nefiind unul de bază) are o atenţie specială din partea creatorilor limbajului.

1.1.2. Funcţii de lucru asociate şirurilor

Există câteva operaţii asociate şirurilor care sunt tratate de către limbajul ***C***, prin funcţii dedicate. Pentru a le înţelege, le vom re-compune, într-o variantă proprie.

1.1.2.1. Parcurgerea şirurilor

Parcurgerea semnifică operaţia prin care, de regulă, plecând dintr-o extremitate a sa, se ating toate componentele (se *vizitează*) şi eventual se aplică o serie de prelucrări asupra lor. Această operaţie este tradiţional secvenţială, adică pentru a ajunge la componenta *i+1*, s-a trecut în pasul anterior prin componenta *i*.

După cum s-a văzut în lucrarea anterioară, o parcurgere se poate face şi 'sărind' peste un număr de componente (*condiţia de pas*).

1.1.2.2. Lungimea unui şir de caractere

Pentru a calcula lungimea unui şir de caractere, acesta trebuie parcurs. Parcurgerea se poate face cu ajutorul pointerilor sau cu ajutorul *operatorului de indexare*, *[ ]*.

Observaţie:
Varianta cu pointeri este mai rapidă, pentru şiruri mari, din cauza evitării folosirii operaţiei de adunare asociată aplicării operatorului de indexare.

Toate şirurile de caractere se încheie cu un caracter special, numit 'end of string', sau *terminator de şir*. Acesta este marcat cu ajutorul *secvenţei escape*:
\0
şi apare la sfârşitul unui şir de caractere.

Observaţie:
1. Caracterul '*\0*' NU este caracter util în lungime (nu se contorizează, nu se numără) ;
2. Plecând de la ideea unui terminator de şir, ne putem gândi la o tehnică similară şi în cazul vectorilor de alt tip de cât *char*. Trebuie însă să fim atenţi cu valoarea pe care o considerăm drept sfârşit de şir, pentru că aplicaţiile respective o pot folosi, şi atunci nu ne va apărea ca o situaţie singulară, ca o excepţie, de care să ne folosim ca atare.

Funcţia de bibliotecă asociată: *strlen( )*.

1.1.2.3. Copierea unui şir în altul

Această operaţie se explică prin ea însăşi. Există un *şir sursă* (***din care***) şi un *şir destinaţie* (***în care***).

Este evident că dacă lungimea şirului destinaţie este mai mică decât a sursei, copierea se va face cu erori. Din această cauză trebuie verificată apriori lungimea şirului destinaţie, după care să înceapă operaţia efectivă de copiere.

Copierea caracterelor dintr-un şir în altul de face unul câte unul, pe măsură ce operaţia de parcurgere atinge indexul respectiv. La aceeaşi poziţie, în ambele şiruri, vom avea în final acelaşi caracter.

Observaţie:

Se mai poate face copierea unui şir în altul şi folosind structurile, situaţie în care copierea se face dintr-odată.

Funcţia de bibliotecă asociată: *strcpy( )*.

1.1.2.4. Comparaţia şirurilor

A compara două şiruri înseamnă a stabili care dintre ele este situat înainte/după celălalt.

Funcţia de comparaţie va returna o valoare întreagă fără semn, după următoarea ***convenţie***:

0 = şiruri identice
1 = primul şir este mai mare (alfabetic situat după) ;
2 = al doilea şir este mai mare ca primul.

Funcţia de bibliotecă asociată: *strcmp()*. Valorile pe care acesta le returnează sunt cele pe care le folosim în convenţia făcută anterior.

1.1.2.5. Căutarea de caractere sau de sub-şiruri într-un şir

Este posibilă situaţia în care avem nevoie să ştim dacă un şir conţine un caracter, sau un alt şir. O situaţie concretă în care avem nevoie de aceste operaţii este cea în care, într-o Bază de date, un câmp de tip *Adresă* conţine întreaga descriere a domiciliului unei persoane. În cadrul acestui câmp, care este evident de tip şir de caractere, sun cuprinse, aşadar, şi numele străzii, şi al blocului, şi numărul apartamentul şi oraşul ş.a. De exemplu, putem avea nevoie doar de numerele impare ale unei străzi date, pentru a face o ofertă de un anume gen. Aceasta înseamnă că trebuie căutată în câmpurile de tip *Adresă*, numele străzii impuse, împreună cu numerele impare presupuse.

Operaţia aceasta este una clasică pentru şirurile de caractere, şi presupune parcurgerea şirului până la:

- întâlnirea *primei* potriviri de tip caracter, sau a *ultimei* potriviri, sau a numărului de apariţii a unui caracter într-un şir;
- întâlnirea unui întreg şir, ca potrivire exactă.

Intervine, deci, marea majoritate a operaţiilor prezentate deja.

1.2. Pointeri (Introducere)

1.2.1. Semnificația pointerului. Segmentarea memoriei

Aşa cum am amintit până în acest moment, un program rulează predominant î memorie. Aici spațiul este împărțit în ***segmente***, care se asociază diferitelor părți ale unui program: date, stivă, cod (sursă = textul programului). Există, deci următoarele segmente importante:

- *segmentul de date* - ce conține declarațiile/definițiile tuturor variabilelor asociate unui program;
- segmentul de cod (de program) - conține textul programului (codul sursă);
- segmentul de stivă - reprezintă o zonă de memorie rezervată apelurilor de funcții. Particularitatea sa este aceea că creşte în sens invers creşterii zonelor de memorie (sugestiv, se spune că stiva creşte în jos, sau scade în sus).

Toate aceste segmente sunt automat valide (valabile, utilizabile) în orice program ***C***.

Este absolut normal ca în cadrul unui segment, compilatorul să-şi poată identifica diferitele entități (variabile, structuri de date, funcții) cu ajutorul unui mecanism. Acest mecanism poartă numele de ***adresare*** şi are la bază ceea se numeşte o ***adresă de memorie***.

Adresa de memorie este un simplu număr întreg, fără semn, adică *unsigned int*. Rolul ei este de a oferi un ***identificator*** (*un număr unic*) fiecărei locații elementare de memorie (***octet*** de memorie). Cum dimensiunea unui segment dintre cele menționate este *64KB*, pentru a reuşi această identificare sunt necesari exact *16b*, pt. că $2^{16} = 64$.

1.2.2. Declarația/definiția variabilelor: efecte d.p.d.v. compilator

Ca orice variabilă obişnuită, şi un pointer se bucură de cele două ***caracteristici*** definitorii, în urma declarării/definirii sale:

- ***Lvalue*** (valoare stânga = adresă) - adică un pointer are la rândul lui un loc în memorie;
- ***Rvalue*** (valoare dreapta) - ceea ce conține pointerul, adică acea valoare întreagă fără semn cu semnificația de locație în memorie a unei alte variabile.

1.2.3. Operatorii de indirectare (*) şi referențiere (*&* = *ampersand*)

În programul anterior, ca şi în programul următor (***7_3***) sunt folosiți operatorii proprii pointerilor, anume *referențierea* (*&*) şi *de-referențierea* sau *indirectarea* (*).

Operatorul & se aplică doar asupra unei entități ce posedă *Lvalue*, cum este numele unei variabile, sau numele unei funcții. Efectul său este luarea adresei entității asupra cărei s-a aplicat.

Operatorul * se aplică asupra numelui unui pointer, dar ***nu*** asupra numelui unei funcții. Efectul său este de anulare a calității de pointer, cu revenire în domeniul *Rvalue* al variabilei pe care o referise (la care punctase) acel pointer.

Operatorul de luare a referinței (*&*) a trebuit să fie folosit în programul anterior pentru că oferă unui pointer sensul pentru care a fost creat (gândit): ***să puncteze*** la o variabilă. De aceea, s-a impus folosirea sa în programul anterior, înainte de a vorbi efectiv despre această operație.

Observație:

Operatorii proprii pointerilor ***sunt duali, complementari***. Sensul acestei afirmații este următorul: dacă asupra unei atribuiri se aplică în ambii membri unul dintre operatori, atunci prin aplicarea operatorului complementar asupra atribuirii modificate se revine la varianta sa inițială.

Dar, ei nu sunt comutativi întotdeauna. Adică se poate spune că sunt comutativi, doar dacă rezultatul final este acelaşi indiferent de ordinare în care ei apar într-o expresie.

Exemplu:

```
int a = 2, *pA ;            // variabila şi pointerul asociat, deocamdată
                            // ne-inițializat.
pA = &a ;                   // inițializarea pointerului.
*(pA) = *(&a) ;             // aplicarea în ambii membri a operatorului de
                            // indirectare (*).
printf("\n Valoarea lui a: %i", a) ;

*&pA = &(*(&a)) ;           // Atenție la ordinea operatorilor din membrul
                            // stâng.
printf("\n Valoarea lui a: %i", a) ;
```

Observație importantă:

Dacă se schimbă ***ordinea*** operatorilor din ultima atribuire, compilatorul marchează următoarea eroare:

14 C:\BC\MyApp\Laborator\Lab_7\lab7_3.cpp - non-lvalue in assignment

Este vorba despre prioritatea operatorilor. Atât *&* cât şi * fac parte din aceeaşi clasă de prioritate (a doua), dar între ei, dacă apar simultan într-o expresie, primul se evaluează *&*. Astfel că, în cazul de față, dacă ordinea este:

*&*pA = &(*(&a)) ;*

atunci, în membrul stâng, înainte să se poată aplica *&*, trebuie evaluată expresia **pA*. Rezultatul acestei evaluări este un număr real (operatorul * aplicat numelui unui pointer îi anulează acestui calitatea de pointer, aducând expresia în gama ***Rvalue*** a variabilei la care punctează (arată) acesta). Dar, operatorul *&* trebuie aplicat, conform definiției, doar asupra numelui unei variabile, şi nu unui număr real. Astfel, compilatorul va marca eroare pe linia respectivă.

Dacă ordinea din membrul stâng este inversă, atunci evaluarea va începe cu operatorul *&*, şi apoi va urma *. Prin aplicarea operatorului *&* asupra numelui pointerului, va rezulta o adresă, anume adresa din memorie unde este stocată variabila de tip pointer. Pe scurt, adresa unei adrese se numeşte ***pointer dublu***. Acum, în momentul în care se aplică şi operatorul de indirectare, el ajunge să fie aplicat asupra unui pointer dublu, efectul fiind un pointer simplu, adică un pointer. În membrul drept situația este următoarea: *&a* înseamnă pointer (adresă), apoi * asupra lui (*&a*) are ca efect revenirea la numele variabilei (am stabilit că operatorii sunt complementari, deci îşi anulează reciproc efectele). Acum, prin folosirea operatorului *&*, rezultatul final este o adresă (*unsigned int*).

Concluzia este clară: atât membrul stâng, cât şi cel drept sunt de tip identic (adresă, sau pointer). Astfel, atribuirea este corectă.

1.2.4. Pointeri constanți

Următoarea declarație:
int a =2, b =4 ;
*int *const pA = &a ;*

semnifică faptul că pointerul denumit *pA* nu-şi poate schimba adresa din memorie la care a fost stabilit. Se spune că avem un 'pointer constant la o variabilă de tip int'. Adică pointerul arată la variabila 'a' (o punctează pe 'a') ***pe tot parcursul*** programului. Astfel că atribuirea:
pA = &b ;

este marcată cu următoarea eroare de către compilator:
25 C:\BC\MyApp\Laborator\Lab_7\lab7_3.cpp - assignment of read-only variable `pA

Trebuie remarcată, deci, ordinea de apariție a tipului de dată şi a lui *const*, dar şi locul de apariție a fiecăruia dintre cuvintele-cheie. *const* apare între asterisc şi numele pointerului, iar tipul *int* este primul, situat în fața asteriscului.

Dacă schimbăm acum ordinea (şi poziția) cuvintelor-cheie astfel:
*int const *pA = &a ;*

sau, în mod echivalent:
*const int *pA = &a ;*

în ambele situații avem de-a face cu un 'pointer la o constantă de tip întreg'. Aici nu mai contează ordinea lui *int* şi a lui *const*, dar ambele, ca poziție, precedă asteriscul.

1.3. Detalii de programare

Pentru înțelegerea programelor utilizate în lucrare studiați *ANEXA* lucrării.

1.4. Comenzile compilatorului

Înainte de compilare codul sursă trebuie salvat (cu combinația de taste *CTRL+S*), sau, echivalent, din meniul *File*, cu comanda *Save as…*.

Compilarea se va face cu comanda *CTRL+F9*, iar rularea cu comanda *CTRL+F10*. Compilarea şi rularea se pot executa, succesiv, printr-o singură tastă, anume *F9*. Acțiunile echivalente din meniu: meniul *Execute* comanda *Compile*, respectiv *Compile + Run*.

Observații:

1. Orice nouă modificare a codului-sursă trebuie urmată *obligatoriu* de salvarea acestuia (*CTRL+S*). Abia după aceea pot urma celelalte acțiuni dorite.
2. Dacă nu se realizează aducerea la zi a programului, compilarea şi rularea ce urmează se fac tot pe varianta veche, adică ce de dinaintea ultimelor modificări.

II. DESFĂŞURAREA LUCRĂRII

2.1. Căutarea de caractere sau de sub0-şiruri într-un şir are o aplicație în programul *lab7_1.cpp*. Declarația/definiția vectorilor şi inițializarea componentelor acestora sunt studiate în ***Lucrarea 6***. Este dată aici o variantă funcțională anume programul este scris folosind apeluri de funcții (proprii sau de bibliotecă). Pentru această aplicație se presupune cunoaşterea operațiilor tipice pe şiruri şi anume *parcurgere* - paragraful 1.1.2.1., *calculul lungimii* unui şir - paragraful 1.1.2.2.).

2.2. Semnificația pointerului este studiată în programul *lab7_2.cpp*. Aici se observă ce este şi ce semnifică pointer-ul şi se tratează operații specifice acestuia: declararea, definirea, inițializarea, afişarea de valori folosind operatorii specifici (indirectare (*) şi referențiere (&)).

2.3. Un studiu mai aprofundat asupra pointerilor se face în programul *lab7_3.cpp*. Sunt reluate noțiunile din programul anterior (indirectarea şi referențierea).

Prin înlăturarea comentariilor de pe liniile *19* şi *31* un compilator de C va marca erori. Acestea se referă la aplicarea incorectă a operatorilor proprii pointerilor. Comentariile de pe liniile 25 şi 26 trebuie înlăturate disjunctiv, adică numai un comentariu din cele două odată. Pentru fiecare dintre situaţii notaţi-vă mesajele asociate acestor erori şi comparaţi-le cu cele din prezentarea teoretică (paragrafele 1.2.3. şi 1.2.4.). Corectaţi în fiecare caz codul sursă.

Observaţie importantă: este posibil ca *Developer C++* să ***nu*** marcheze eroare pe nici una dintre liniile incorecte din exemplul dat. Aceasta pentru că este astfel proiectat încât să înlăture intern cât mai multe dintre erorile frecvente de programare. Nu este poate o modalitate acceptabilă, pentru că un programator va avea în majoritatea cazurilor impresia corectitudinii codului-sursă proiectat, impresie care este, evident, eronată.

2.4. Pentru şirurile simetrice faţă de mijloc numite *şiruri palindrom* (şirul care citit din oricare din capete este identic) rulaţi programul *lab7_4.cpp*. Puteţi observa folosirea funcţiilor de bibliotecă pentru operaţii aritmetice (*floor()* şi *ceil()*), realizarea parităţii şi imparităţii unor numere, funcţia de calcul a lungimii unui şir (*strlen()*, prezentă în fişierul antet *string.h*), utilizarea *switch-case* şi a instrucţiunii de ciclare *for()*.

III. ÎNTREBĂRI ŞI EXERCIŢII

3.1. Creaţi un program prin care să parcurgeţi un vector de caractere, de dimensiune *80*, de la dreapta spre stânga. Afişaţi fiecare componentă vizitată.
Observaţie: ***nu*** există funcţie de bibliotecă asociată care să îndeplinească această cerinţă.

3.2. Modificaţi programul anterior şi număraţi şi trecerile prin ciclul folosit. Lungimea unui şir de caractere nu va cuprinde şi *caracterul de sfârşit de şir*.
Observaţie: Funcţia de bibliotecă asociată: *strlen( )*.

3.3. Concepeţi un program în care, după ce aţi citi elementele unui şir sursă, să le copiaţi într-un şir destinaţie.
Observaţii:
- Aveţi în vedere situaţiile particulare ale depăşirilor ce pot apărea în parcurgerea şirurilor ;
- dimensiunea şirului destinaţie trebuie să fie cel puţin egală cu a şirului sursă ;
- Funcţia de bibliotecă asociată: *strcpy( )*.

3.4. Folosind parcurgerea şirurilor şi operația de comparație (aplicată caracterelor) realizați programul care să spună care dintre două şiruri este situat înaintea celuilalt. Funcția va returna o valoare întreagă fără semn, după următoarea ***convenție***:

0 = şiruri identice
1 = primul şir este mai mare (alfabetic situat după) ;
2 = al doilea şir este mai mare ca primul.

Observație: Funcția de bibliotecă asociată: *strcmp()*. Valorile pe care acesta le returnează sunt cele pe care le folosim în convenția făcută anterior.

3.5. Se dau variabilele şi pointerii următori:

```
float v1 = -12,  v2 ;
float *pV1 = &v1, *pV2=&v2 ;
```

Afişați valorile variabilelor şi adresele lor din memorie, în cel puțin două moduri. Stabiliți apoi o valoare pentru variabila *v2*, în mod direct şi indirect.

IV. ANEXA - sursele complete ale programelor

// lab7_1.cpp

```
// funcții de lucru cu şiruri (proprii sau de bibliotecă)
#include<stdio.h>
#include<string.h>

int lungime (char []) ;  // argumentul este un sir (vector de caractere)

int main(void){
 char sir1[] = "Bine ati venit!" ;          // primul mod de initializare
 char sir2[] = {'B', 'i', 'n', 'e', '\0'} ;  // al doilea mod de initializare

// calculul lungimii sirurilor
// lungimea sirului folosind functii proprii
 printf("\n Lungime 1: %i", sizeof(sir1)-1) ;
 printf("\n Lungime 2: %i", sizeof(sir2)-1) ;
 printf("\n Lungimea proprie a primului sir: %i", lungime(sir1)) ;
 printf("\n Lungimea proprie a celui de-al doilea sir: %i", lungime(sir2)) ;

// lungimea sirului folosind functia de biblioteca
 printf("\n Lungimea primului sir (strlen()): %i", strlen(sir1)) ;
 printf("\n Lungimea celui de-al doilea sir (strlen()): %i", strlen(sir2)) ;

// final
 printf("\n Gata...") ;
```

```
 scanf("%s", sir1) ;
}

int lungime (char sir[]) {
 char *pCh = sir ;
 int lungime=0;

 while(*pCh) { lungime++ ; pCh++ ;}
                                    // avansez cu pointerul variabil, prin sir.

 return lungime ;
}
```

// lab7_2.cpp

// despre pointeri: iniţializare, definire, afişare

```
#include<stdio.h>

int main(void){
 float varReal = 1.1 ;    // variabila, in aces tmoment, are un spatiu rezervat
          // in memorie, adica are Lvalue (adresa); proprietatea sa
          // Rvalue este data de multimea de valori posibile, adica
          // multimea numerelor reale (incazul de fat variabila este de tip real).
 float *pReal = &varReal ;
          // pointerul are si el, in acest moment, un loc in
          // memorie, adica are la randul sau o adresa; proprietatea
          // sa Rvalue este data de adresa din memorie a variabilei
          // la care puncteaza, adica de Lvalue (varReal).

// afisarea proprietatilor pointerului
 printf("\n Lvalue pt. pointer: %p sau, echivalent: %u", &pReal, &pReal) ;
 printf("\n Rvalue pt. pointer: %p sau, echivalent: %u", pReal, pReal) ;

 // afisarea proprietatilor variabilei 'varReal'
 printf("\n Lvalue pt. variabila: %p sau, echivalent: %u", pReal, pReal) ;
 printf("\n Rvalue pt. variabila: %f.", varReal) ;

 // modificarea Rvalue a variabilei
 *pReal = 2.2 ; // in memorie, se scrie valoarea 2.2 in locaţia rezervata
                          // lui varReal, indirect, prin pointer.
 printf("\n Noua Rvalue pt. variabila: %f, sau, echivalent: %f",
                                                    varReal, *pReal) ;
// incheiere
 printf("\n Gata...") ;
 scanf("%f", &varReal) ;
}
```

// lab7_3.cpp
// despre pointeri 2: indirectare si referențiere.

```
#include<stdio.h>

int main(void){
// operatorii proprii pointerilor sunt complementari
int a = 2, *pA ; // variabila si pointerul asociat, deocamdata
                 // ne-initializat.

pA = &a ;              // initializarea pointerului
*(pA) = *(&a) ;        // aplicarea în ambii membri a operatorului de
                       // indirectare (*).
printf("\n Valoarea lui a: %i", a) ;

*&pA = &(*(&a)) ;      // aici membrul stang rezulta din anteriorul, prin
                       // aplicarea referentierii.
printf("\n Valoarea lui a: %i", a) ;

// erori de aplicare a operatorilor proprii pointerilor
/*
 {
  int a = 2 ;
  const int b = 4 ;              // orice constanta nu are Lvalue (!!)
  int *const pA = &a ;           // corect, initializarea pointerului

//  a = *2 ;       // incorect; 2 nu este numele unui pointer
//  pA = &b ;      // incerc sa schimb locul unde arata pointerul 'pA', de la
                   // variabila 'a' la 'b'. Dar 'b' u poate fi adresata. Aceasta
                   // ar insemna posibilitatea modificarii valorii sale stocate
                   // in memorie.
 printf("\n Valoarea lui b: %i, %i", b, *pA) ;
 }
*/
// final
 printf("\n Sfarsit...") ;
 scanf("%i", &a) ;
}
```

// lab7_4.cpp
// şir palindrom = este identic citit din oricare din ambele capete.
// variantă

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

```
#define DIM 16
typedef unsigned int BOOL ;     // pentru DA sau NU
typedef char *const pChar ;     // tipul de data pointer constant la caracter

BOOL palindrom(char*const) ;
int main(void){
 char sir1[DIM] = "AnnA" ;              // un palindrom
 char sir2[DIM] = "20011002" ;          // alt palindrom
 char sir3[DIM] = "10011002" ;          // nu e palindrom
 int i=1 ;

 switch(i){
        case 1:
                        if(palindrom(sir1))
                                printf("\n Sirul dat este palindrom.");
                        else printf("\n Mai incercati...") ;
                        ++i ;
                        printf("\n Continuati cu un caracter...") ;
                        getch() ;
        case 2:
                        if(palindrom(sir2)) printf("\n Sirul este palindrom.");
                        else printf("\n Mai incercati...") ;
                        ++i ;
                        printf("\n Continuati cu un caracter...") ;
                        getch() ;
        case 3:
                        if(palindrom(sir3))
                                printf("\n Sirul dat este palindrom.");
                        else printf("\n Mai incercati...") ;
                        ++i ;
                        break ;
 }

 // incheiere
 puts("\n Gata...") ;
 scanf("%s", sir1) ;
}

BOOL palindrom(char *const sir) {
 int i, lungime = strlen(sir);
 pChar inceput = sir ;  // incep cu primul caracter din sirul luat ca parametru
 BOOL marcat[DIM] ;

 for(i=0; i<lungime; i++) marcat[i] = 0 ;
 if(!(lungime % 2)) printf("\n Numar par de caractere %i...", lungime) ;
 else printf("\n Numar impar de caractere %i.", lungime) ;
```

```
for(i=0; i<floor(strlen(sir)/2); i++){
  printf("\n Se compara pozitiile %d si %d", i, (lungime-1)-i) ;
  if(*(inceput+i) == *(inceput+(lungime-1)-i)) {
        marcat[i] = marcat[lungime-1-i] = 1 ;
                        // marchez pozitiile 'parcurse si comparate',
                        // din sirul preluat ca parametru.
   continue ;
   }
   else {
        printf("\n\n Sirul \"%s\" NU este palindrom...", sir) ;
        for(i=0; i<lungime; i++) printf("\n Pozitia %i din sirul marcat: %i",
                                                    i, marcat[i]) ;
        return 0;
        }
  }

printf("\n\n Sirul \"%s\" ESTE palindrom", sir) ;
for(i=0; i<lungime; i++) printf("\n Pozitia %i din sirul marcat: %i",
                                                    i, marcat[i]) ;
return 1 ;
}
```