Programarea Calculatoarelor
Curs 10

Iulian Năstac

4. Apelarea prin argumentele unei funcţii

În limbajul C apelarea unei funcţii se poate face (relativ la natura argumentelor sale) în două feluri:

- prin valoare;

- prin referinţă.

Recapitulare

a. Apelarea prin valoare

În acest caz, care este şi cel mai uzual, se copiază valoarea unui argument într-un parametru formal al subrutinei. Modificările efectuate asupra parametrului utilizat nu au efect asupra argumentului efectiv al funcţiei.

b. Apelarea prin referinţă

Atunci când parametrul funcţiei este un pointer, în parametrul formal se copiază de fapt adresa unui argument. În interiorul subrutinei adresa este folosită pentru acces la argumentul folosit efectiv la apelare. Aceasta înseamnă că modificările efectuate asupra parametrului afectează argumentul.

5. Valori returnate de o funcţie

Pentru a returna o valoare, funcţia trebuie să utilizeze instrucţiunea ***return***. Această instrucţiune prezintă trei aspecte importante:

- Forţează ieşirea imediată din funcţia în care se află execuţia programului.
- Poate returna o valoare de tipul funcţiei.
- Determină execuţia programului să se reîntoarcă la instrucţiunea imediat următoare aceleia din care a fost apelată funcţia.

6. Funcţii recursive

În principiu o funcţie poate să se apeleze pe ea însăşi, ceea ce implică automat un fenomen de recursivitate.

Definiţie: o funcţie este recursivă dacă o instrucţiune din corpul ei apelează chiar acea funcţie.

7. Eficienţa funcţiilor

Funcţiile sunt esenţiale pentru a scrie eficient programele, mai puţin cele simple. Însă, în unele aplicaţii specializate este mai bine să se elimine o funcţie şi să se înlocuiască cu un *cod inline*.

Codul *inline* efectuează aceleaşi acţiuni ca şi ale funcţiei dar fără dezavantajul produs de o apelare de funcţie (printre care se numără şi micşorarea vitezei).

8. Declararea funcţiilor cu număr variabil de parametri

În C pot fi specificate funcţii cu un număr variabil de parametri.

Ca exemplu, funcţia:

func(int a, int b, ...);

va avea cel puţin doi parametrii de tip întreg şi un număr necunoscut (inclusiv 0) de parametrii după cei deja declaraţi.

Variabile

- Locale
- Globale

Variabile locale pot fi:

- automatice
- statice
- registru

Observații privind variabilele:

- Variabilele globale (precum și cele statice) sunt inițializate cu 0 în mod automat.

- Variabilele locale iau valori arbitrare, ce depind de configurația biților din stiva unde sunt efectiv alocate.

Inițializarea variabilelor

- Prin inițializare se acordă o valoare inițială pentru o ***variablă***, de obicei imediat la declararea sa.

- Sintaxa:

  clasa tip nume = expresie;

Observații:

- Evitați utilizarea variabilelor locale (auto) care nu sunt inițializate. Fără inițializare, aceste variabile au valori arbitrare, fenomen ce se datorează faptului că memoria stivă, ce le conține, nu șterge efectiv valorile vechilor variabile dezalocate (adica nu pune valoarea 0 în zonele dezalocate).

- **Atenție: parametrii unei funcții nu se inițializează!**

Cap. **Matrici (Tablouri)**

- Tehnica tablourilor reprezintă o metodă comodă de stocare a matricilor în memoria unui sistem de calcul.
- **În programarea C,** o matrice (sau tablou) este o colecţie de variabile de acelaşi tip, oricare dintre elemente putând fi apelate cu acelaşi nume**.**
- C-ul utilizează principiul "Row Major", adică elementele, de pe fiecare rând al unei matrici, sunt stocate succesiv în memorie (la adrese succesive raportate la dimensiunea tipului).

Declararea unei matrici

- Sintaxa:

  tip Nume_matr [lim_1] [lim_2] … [lim_n];

unde:

- *tip* → tipul elementelor matricei
- *lim_i* → limita superioară a indicelui al i-lea (ce poate avea valorile:0, 1, 2,… , lim_i – 1)

Observaţii:

- Valorile *lim_i* (unde *i* = 0, 1, …, *n*) sunt expresii constante întregi pozitive.
- La elementele tabloului se poate face referire folosind variabile cu indici.
- În *C* numele unui tablou este un simbol care are ca valoare adresa primului său element.
- La întâlnirea unei declaraţii de tablou, compilatorul alocă o zonă de memorie necesară pentru a păstra valorile elementelor sale.
- Tablourile şi pointerii prezintă o strânsă legătură în C (a se vedea capitolul viitor dedicat pointerilor).

Exemplu:

- **int a[3][4];**

  se declară o matrice de numere întregi de 3 linii și 4 coloane. Indexul de linie pornește de la 0 și ajunge până la 2.

	Coloana 0	Coloana 1	Coloana 2	Coloana 3
rândul 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
rândul 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
rândul 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Legătura dintre pointeri şi matrici

- Numele unui tablou este un pointer deoarece el are ca valoare adresa primului său element.

- Există o diferenţă între numele unui tablou şi o variabilă de tip pointer. Astfel, unei variabile de tip pointer i se pot atribui valori la execuţie, în timp ce acest lucru nu este posibil să se realizeze pentru numele unui tablou (acesta are tot timpul ca valoare adresa primului său element).

- Se obişnuieşte să se spună că numele unui tablou este un ***pointer constant***.

Matrici unidimensionale (vectori)

- Vectorii reprezintă matricele (tablourile) cu o singură dimensiune.

- Format:

tip nume_vect [lim];

Observaţii:

1. În C toate matricele au ca indice pentru primul element valoarea zero (0). La declararea:

char p[10];

se creează o matrice de 10 elemente de tip caracter de la *p[0]* la *p[9]*.

2. Cantitatea de memorie necesară pentru înregistrarea unei matrice este direct proporţională cu tipul şi mărimea sa. Mărimea totală în octeţi se calculează:

*total octeţi=sizeof(tip de bază)*mărimea matricei*

3. *C* nu controlează limitele unei matrice. Se pot depăşi ambele margini ale unei matrice şi scrie în locul altor variabile sau peste codul programului. Rămâne în sarcina programatorului să controleze limitele acolo unde este necesar.

Exemplu de greșeală:

```
…
int mat[100], i;
…
for (i=0; i<=100;  i++)
        mat[i]=i;

…
```

Obs.: Codul de mai sus va fi compilat fără eroare, însă el este greşit deoarece bucla *for* va determina ca matricea *mat* să-şi depăşească limitele.

- Un vector este de fapt o listă de elemente de același tip stocate în locații succesive de memorie, în ordinea indicilor.

Examplu de relație între un vector și un pointer:

```
……
int t[10];
int *p;
int x;
……
p=t;
……
x=t[0];
```

Transmiterea vectorilor ca argumente de funcţii

- Funcţiile utilizează *parametrii formali* (argumente).

- În limbajul *C* nu se poate transmite o matrice întreagă ca argument al unei funcţii.

- Un parametru formal, ce corespunde unui parametru efectiv care este nume de tablou unidimensional, poate fi declarat fie ca *tablou,* fie ca *pointer spre tipul tabloului.*

Dacă o funcţie primeşte o matrice unidimensională, parametrul său formal poate fi declarat în unul din următoarele 3 moduri:

- ca pointer;
- ca matrice cu dimensiune;
- ca matrice fără dimensiune.

Exemple:

a) *void funct (int *x)* /* pointer*/

{.....

}

b) *void funct (int x[10])* /* matrice cu dimensiune*/

{.....

}

c) *void funct (int x[ ])* /* matrice fără dimensiune */

{.....

}

Observaţii:

1) Toate cele trei metode de declarare, anterior exemplificate, determină rezultate similare deoarece fiecare transmite compilatorului că urmează să primească un pointer către un întreg (de fapt în primul exemplu se foloseşte clar un pointer).
2) În cazul folosirii ca argument de funcţie, mărimea matricei nu contează efectiv, deoarece *C* nu efectuează controlul limitelor. Va fi absolut corectă şi forma:

```
void funct (int x[50])
{.....
}
```

deoarece *C*-ul creează un cod care instruieşte *funct( )* să primească un pointer către matrice şi nu creează efectiv o matrice de 50 de elemente.

Exemplu: program pentru ordonare crescătoare ale unor numere

Date inițiale:

- Programul citește un șir (vector) de numere și le afișează ulterior în ordine crescătoare.

- Numărul maxim de elemente este 1000.

- Se va folosi o funcție pentru ordonarea numerelor:

  void ordcresc(double tab[ ], int n)

```
#include <stdio.h>
#include<conio.h>
#define MAX 1000
void ordcresc(double tab[ ],int n);
int main()
{ double v[MAX];
  int m,s,i;
  printf("\nIntroduceti nr. de elemente = ");   scanf("%d", &m);
  printf("\nIntroduceti elementele sirului :  \n");
  for(s=0 ; s<m ; s++)
    scanf("%lf",&v[s]);
    ordcresc(v,m);
    for(i=0;i<m;i++)
    {  printf("v[%d]=%g\n",i,v[i]);
       if((i+1)%23==0)
              {  printf("Actionati o tasta pentru a continua\n");  getch();
              }
    }
    getch();
}
…
```

```
…
void ordcresc(double tab[ ], int n)
 /* ordoneaza elementele lui tab in ordine crescatoare*/
 {
   int i,ind;
   double t;
   ind=1;
   while(ind)
     {
       ind=0;
       for(i=0;i<n-1;i++)
             if(tab[i]>tab[i+1])
              { t=tab[i];
                tab[i]=tab[i+1];
                tab[i+1]=t;
                ind=1;
              }   /*sfarsit if*/
     }  /*sfarsit while*/
 }
```

Modificatorul *const*

- În limbajele *C* si *C++* există posibilitatea de a defini date constante folosind modificatorul *const* în declaraţii.
- Printr-o astfel de declaraţie, unui nume i se atribuie o valoare iniţială, care nu poate fi modificată ulterior printr-o altă expresie de atribuire (ca în cazul variabilelor).
- Deci o dată declarată cu ajutorul modificatorului *const,* o variabilă se consideră că este constantă.
- Reamintim că multe constante pot fi declarate cu ajutorul lui *#define*.

Formate posibile ale unei declaraţii cu modificatorul *const*:

1) tip *const* nume = valoare;

2) tip *const* *nume = valoare;

3) *const* tip nume = valoare;

4) *const* tip *nume = valoare;

5) *const* nume = valoare;

Observaţii:

- Avem echivalenţa 1) ⇔ 3) şi 2) ⇔ 4).

- La punctele 2) şi 4) se poate modifica nume (*nume=t* unde *t* este pointer de tipul *char*) dar atribuirile de genul:

**nume='a'*

sau

**(nume+1)='b'*

sunt incorecte.

Observaţii (continuare):

Dacă valoarea de atribuire lipseşte înseamnă că avem de-a face cu un parametru formal al unei funcţii (doar pentru pointeri, pentru că în rest apelarea prin valoare a unei funcţii nu modifică valorile variabilei folosite).

De exemplu în cazul:

tip f (const tip *nume);

o atribuire de forma **nume=valoare* este interzisă.

Prin folosirea modificatorului *const* funcţia nu mai poate modifica nimic din argumentul folosit la apelare (*nici adresa locaţiei*, şi *nici conţinutul locaţiei de memorie*).

Atenţie:

- Există posibilitatea de a eluda restricţiile impuse de modificatorul *const*.
- Dacă folosim o altă variabilă care se referă la aceeaşi locaţie de memorie, aceasta (locaţia) îşi poate modifica conţinutul:

```
...
char const *s = ”sir de caractere”;
char *p;
p = (char *)s;
*p = ’a’;
*(p+1) = ’b’;
...
```

Şiruri de caractere în C

- Şirul de caractere reprezintă una din cele mai utilizate clase de aplicaţii în cadrul matricelor unidimensionale.

Caracteristici ale şirurilor de caractere:

1) Un şir de caractere se păstrează într-o zonă de memorie organizată ca *tablou unidimensional* de tip ***char***.

2) Fiecare caracter se păstrează pe câte un octet prin codul său numeric.

3) Cel mai frecvent utilizat în acest scop este codul **ASCII**.

4) După ultimul caracter al şirului se păstrează caracterul **NUL** (sau *null*) care este ’\0’ (şi are valoarea vidă).

5) Deoarece există *null* în poziţie finală, se declară matricele de tip caracter cu un caracter mai mult decât cel mai mare şir pe care îl conţine.

De exemplu, pentru a crea matricea sir care conţine cuvântul "hi", vom scrie:

```
char sir[3];

sir[0] = 'h';

sir[1] = 'i';

sir[2] = '\0';

printf("\n %s \n", sir);
```

Caracteristici (continuare):

6) Deşi *C*-ul nu are explicit date de tip *şir*, el permite constante de tip *şir*. O constantă de tip *şir* este o listă de caractere închise între ghilimele (de exemplu “salut”).

7) Nu este necesar să fie introdus manual caracterul ***null*** la sfârşitul şirului de caractere (compilatorul o face automat).

De exemplu, dacă pentru formarea şirului se utilizează funcţia

gets(şir);

atunci **‘\0’** este introdus automat de către compilator.

8) Pentru a opera un şir de caractere se poate utiliza:

- *numele tabloului* (ca pointer constant spre şirul respectiv);
- *pointeri variabili* spre şirul de caractere.

Exemplu:

Dacă declarăm:

char tab[ ] =”Acesta este un sir”;

atunci:

tab – adresa caracterului *A*
tab+1 – adresa caracterului *c*
tab+2 – adresa caracterului *e*
… *etc.*

sau:

tab[0] → codul ASCII al caracterului *A*
tab[1] → codul ASCII al caracterului *c*
… *etc.*

sau:

**tab* → codul ASCII al caracterului *A*
**(tab+1)* → codul ASCII al caracterului *c*
… *etc.*

Observaţie:

Un efect similar se obţine cu declaraţia:

char const *p =”Acesta este un sir”;

unde:

p[0] *sau* ****p*** *- conține codul ASCII al primului caracter (`A`)*

p[1] *sau* ****(p+1)*** *- conține codul ASCII al celui de-al doilea caracter (`c`)*

…

Funcţii standard utilizate în prelucrarea şirurilor de caractere

- Bibliotecile standard ale compilatoarelor C conţin o serie de funcţii care permit operaţii cu şiruri de caractere.

- Majoritatea acestor funcţii au prototipul în fişierul ***string.h***.

Operaţiile pe care le execută funcţiile de prelucrare a şirurilor de caractere pot fi:

- calculul lungimii şirurilor de caractere (ex.: *strlen*)
- copierea şirurilor de caractere (ex.: *strcpy*)
- concatenarea şirurilor de caractere (ex.: *strcat*)
- compararea şirurilor de caractere (ex.: *strcmp*)
- căutarea (identificarea) şirurilor sau caracterelor (ex.: *strchr* sau *strstr*)