

Limbaje de programare

Tablouri. Adrese. Şiruri de caractere

15 noiembrie 2010

Declararea tablourilor

Tablou (vector) = un șir de elemente de *același tip* de date

Tabloul x asociază la un *indice* n , o *valoare* $x[n]$

În matematică, același lucru face un *șir* x_n sau o *funcție* $x(n)$

Declarare: `tip nume-tablou[nr-elem];`

```
double x[20];    int mat[10][20];
```

Inițializare: între acolade, cu virgule: `int a[4] = { 0, 1, 4, 9 };`

Dimensiunea tabloului (nr. de elemente) = o *constantă* pozitivă

C99 acceptă și dimensiuni variabile, cu valoare cunoscută în momentul declarării

```
void f(int n) { int tab[n]; /* n e cunoscut în momentul apelului */ }
```

Sintaxa declarației: `tip a[dim];` sugerează că `a[indice]` are tipul `tip`

Folosirea tablourilor

Un *element* de tablou *nume-tablou*[*indice*] e folosit ca orice *variabilă* are o valoare, poate fi folosit în expresii, poate fi atribuit

```
x[3] = 1; n = a[i]; t[i] = t[i + 1]
```

Indicele poate fi orice *expresie* cu valoare *întreagă*

ATENȚIE! În C, indicii de tablou sunt de la *zero* la *dimensiune - 1*
`int a[4];` are elemente `a[0]`, `a[1]`, `a[2]`, `a[3]`, **NU există** `a[4]`

Exemplu de traversare și atribuire a unui tablou:

```
int a[10]; for (int i = 0; i < 10; ++i) a[i] = i + 1;
```

Constante simbolice ca dimensiuni de tablou

E util să folosim un nume de *constantă (macro)* pentru dimensiune

```
#define NUME val
```

Preprocesorul C înlocuiește NUME în sursă cu val înainte de compilare

```
#define LEN 30
double t[LEN];
for (int i = 0; i < LEN; ++i) { // tabelam fct. sin cu pasul 0.1
    t[i] = sin(0.1*LEN); printf("%f ", t[i]);
}
```

Programul e mai ușor de citit, e clar că LEN e lungimea tabloului.

Pentru a schimba dimensiunea, modificăm programul doar într-un loc
⇒ evităm greșelile din neatenție sau uitare

Exemplu: Calculul primelor numere prime

```
#include <stdio.h>
#define MAX 100          // preprocesorul inlocuieste MAX cu 100
int main(void) {
    unsigned p[MAX] = {2};    // primul element initializat cu 2
    unsigned cnt = 1, n = 3;  // avem un prim, 3 e urmatorul candidat
    do {
        for (int j = 0; n % p[j]; ++j) // cat timp nu am gasit divizor
            if (p[j]*p[j] > n) {      // daca nu mai sunt altii e prim
                p[cnt++] = n; break;   // il inregistram si iesim din ciclu
            }
        n += 2;                      // trecem la numarul impar urmator
    } while (cnt < MAX);             // pana nu e plin tabloul
    for (int j = 0; j < MAX; ++j)
        printf("%d\n", p[j]);       // tiparim cate un element pe rand
    return 0;
}
```

Tablouri multidimensionale (matrice)

Sunt de fapt tablouri cu elemente care sunt la randul lor tablouri.

Declarație: *tip nume*[*dim1*][*dim2*]. . . [*dimN*];

Exemple: `double m[6][8]; int a[2][4][3];`

`m`: tablou de 6 elemente, fiecare un tablou de 8 reali. Element: `m[4][3]`

Și aici: dimensiuni *constante* (în C99: cunoscute la declarare)

Elementele tabloului sunt dispuse succesiv în memorie:

`m[i][j]` e pe poziția $i * COL + j$

Un exemplu cu matrice

```
#define LIN 2    // numarul de linii
#define COL 5   // numarul de coloane
int main(void) {
    double a[LIN][COL] = { {0, 1, 2, 3, 4}, 5, 6, 7, 8, 9 };
    // initializare: cu acolade la fiecare linie, sau un singur șir

    for (int i = 0; i < LIN; ++i) { // parcurge linii
        for (int j = 0; j < COL; ++j) // parcurge coloane
            printf("%f ", a[i][j]);
        putchar('\n'); // sfarsit de linie
    }
    return 0;
}
```

Variabile și adrese

Orice variabilă `x` are o adresă: acolo e memorată valoarea ei

Operatorul prefix & dă adresa operandului: `&x` e adresa variabilei `x`

Operandul lui `&`: orice *lvalue* (destinație validă de atribuire): variabile, elemente de tablou. NU au adrese: alte expresii, constantele

Numele unui tablou e chiar *adresa* tabloului.

Fie `int a[6];` Numele `a` reprezintă *adresa* tabloului.

Numele `a` NU reprezintă toate elementele împreună!

O adresă poate fi tipărită (în hexazecimal) cu formatul `%p` în `printf`

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

```
int main(void) {
```

```
    double d; int a[6];
```

```
    printf("Adresa lui d: %p\n", &d); // folosim operatorul &
```

```
    printf("Adresa lui a: %p\n", a); // a e adresa, nu e nevoie de &
```

```
    return 0;
```

```
}
```


Tablouri ca parametri la funcții

Declarația unui tablou alocă și memorie pentru elementele sale

dar *numele* reprezintă *adresa* sa și nu tabloul ca tot unitar

⇒ numele tabloului *NU* poartă informații despre dimensiunea lui

excepție: `sizeof(numetab)` este $nr\text{-}elem * sizeof(tip\text{-}elem)$

La funcții trebuie transmis *numele* tabloului (*adresa*) *ȘI lungimea* sa

NU scriem lungimea între [] la parametru, nu e luată în considerare

```
#include <stdio.h>
void printtab(int t[], unsigned len) {
    for (int i = 0; i < len; ++i) printf("%d ", t[i]);
    putchar('\n');
}
int main(void) {
    int prim[10] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };
    printtab(prim, 10); // ATENTIE: NU prim[10], NU prim[]
    return 0;
}
```

Tablouri ca parametri la funcții

Transmiterea parametrilor în C se face *prin valoare*

⇒ un parametru tablou e transmis prin *valoarea adresei sale*

Având adresa, funcția poate accesa (*citi ȘI scrie*) elementele tabloului

```
void sumvect(double a[], double b[], double r[], unsigned len) {  
    for (unsigned i = 0; i < len; ++i) r[i] = a[i] + b[i];  
}
```

```
#define LEN 3    // macro pt. constanta utilizata de mai multe ori  
int main(void) {  
    double a[LEN] = {0, 1.41, 1}, b[LEN] = {1, 1.73, 1}, c[LEN];  
    sumvect(a, b, c, LEN);  
    return 0;  
}
```

Inițializare

Tablourile neinițializate au elemente de valoare necunoscută.

Tablourile inițializate parțial au restul elementelor nule.

Tablouri de dimensiune variabilă (C99)

cu dimensiune cunoscută la declarare (ex. parametru la funcție)

```
#include <stdio.h>

void fractie(unsigned m, unsigned n) {
    int apare[n]; // dimensiune data de parametrul n
    for (int i = 0; i < n; ++i) apare[i] = 0; // init
    printf("%u.", m/n); // catul
    while (m %= n) { // rest nenul
        if (apare[m]) { printf("%u...", 10*m/n); break; } // periodic
        apare[m] = 1; // marcam ca apare
        m *= 10; putchar(m/n + '0'); // urmatoarea cifra
    }
    putchar('\n');
}

int main(void) {
    fractie(5, 28); // 5/28 = 0.178571428...
    return 0;
}
```

Tablouri multidimensionale ca parametri la funcții

$m[i][j]$ e pe poziția $i*COL+j \Rightarrow$ trebuie cunoscut $COL \Rightarrow$ la parametri trebuie *toate* dimensiuni. în afară de prima. Ex: $A_{lin \times 10} \times B_{10 \times 6} = C_{lin \times 6}$

```
void matmul(double a[][10],double b[][6],double c[][6],int lin) {
    for (int i = 0; i < lin; ++i)    // functia e buna doar pentru
        for (int j = 0; j < 6; ++j) { // matrici cu dim. 10 si 6
            c[i][j] = 0;
            for (int k = 0; k < 10; ++k) c[i][j] += a[i][k]*b[k][j];
        }
} // pentru folosire vom scrie (de exemplu in main):
double m1[8][10], m2[10][6], m3[8][6]; // le dam apoi valori
matmul(m1, m2, m3, 8); // NU: m1[][], NU: m2[][6], NU: m3[8][6]
```

În C99: parametri la funcții pot fi tablouri de dimensiuni variabile (dar cunoscute în momentul apelului – dimensiunile sunt tot parametri)

```
void matmul(int lin, int n, int p, double a[][n], double b[n][p],
            double c[][p]); // n, p declarati inainte de folosire
```

Tablouri și șiruri de caractere

```
char cuvant[20]; // tablou de caractere neinitializat
char msg[] = "test"; // 5 octeti, terminat cu '\0'
char msg[] = {'t','e','s','t','\0'}; // acelasi, scris altfel
char nume[3] = { 'E', 'T', 'C' }; // nu are '\0' la sfarsit !
char sir[20] = "test"; // restul pana la 20 sunt '\0'
```

În C, termenul *șir de caractere* înseamnă un tablou de caractere încheiat în memorie cu caracterul/octetul '\0' (la fel *constantele șir*: "salut\n")
(la memorare, nu în reprezentarea la intrare: nu citim/tipărim '\0')

ATENȚIE: toate funcțiile standard pentru șiruri depind de aceasta!
nu au nevoie de parametru lungime, dar șirul trebuie terminat cu '\0'

La șiruri inițializate, dar fără dimensiune specificată (ex. `msg` mai sus) se alocă dimensiunea inițializatorului + 1 caracter '\0'

Tipul pointer

Rezultatul unei operații *adresă* are un tip, ca și orice expresie

Pentru o variabilă declarată *tip x*; *tipul adresei* sale *&x* e *tip **
(citit: *pointer la tip*, adică: adresă unde se află un obiect de acel *tip*)

În particular, *numele* unui tablou are tipul pointer la tipul elementului
`int a[4];` `a` are tipul `int *` `char s[8];` `s` are tipul `char *`

La declararea parametrilor funcției, `void f(tip a[])` înseamnă de fapt
`void f(tip *a)` (de aceea dimensiunea: `void f(tip a[6])` *nu contează*)

Tipul unei constante șir de caractere "sir" este `char *`:
adresa unde se găsește șirul în memorie

Valoarea specială NULL (0 de tip `void *` = adresă de tip neprecizat)
e folosit pentru a indica o adresă *invalidă*

Funcții cu șiruri de caractere (string.h)

```
size_t strlen(const char *s); // returneaza lungimea sirului s
char *strchr(const char *s, int c); // cauta caract. c in sirul s
// returneaza adresa unde l-a gasit sau NULL (0) daca nu-l gaseste
char *strcpy(char *dest, const char *src); // copiaza src in dest
char *strcat(char *dest, const char *src); // concat. src la dest
// pentru ambele e necesar ca la dest sa fie loc suficient
int strcmp (const char *s1, const char *s2); // compara 2 siruri
// returneaza intreg < 0 sau == 0 sau > 0 dupa cum e s1 fata de s2
char *strncpy(char *dest, const char *src, size_t n);
// copiaza cel mult n caractere din src in dest
char *strncat(char *dest, const char *src, size_t n);
// concateneaza cel mult n caractere din src la dest
int strncmp (const char *s1, const char *s2, size_t n);
// compara sirurile pe lungime cel mult n caractere
size_t: tip întreg fără semn pentru dimensiuni
const: specificator de tip, indică că obiectul respectiv nu e modificat
```