SDA (PC2)

Curs 2
Matrici multidimensionale

Iulian Năstac

Matrici (Recapitulare 1)

- Declaraţia de matrice se face respectând următorul format:

tip nume [lim_1][lim_2]……[lim_n];

unde: *tip* → tipul elementelor matricei

lim_i → limita superioară a indicelui al i-lea (indicele respectiv poate avea valorile:0, 1, 2,… , lim_i – 1)

Matrici (Recapitulare 2)

Observaţii:

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

Matrici (Recapitulare 3)

- **Vectori**
- **Transmiterea vectorilor ca argumente de funcţii**

Exemple:

a) *void funct (int *x)* /* pointer*/
{.....
}

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

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

Matrici (Recapitulare 4)

- Şiruri de caractere

- Funcţii standard utilizate în prelucrarea şirurilor de caractere

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***, ***strstr***)
- Funcţiile standard prin care se realizează aceste operaţii au fiecare un nume care începe cu prefixul *str* (prescurtare de la *string*).

Matrici (Recapitulare 5)

- **Matrici bidimensionale**

- **Matrice bidimensională ca argument pentru funcţii**

```
void funct(int x[ ][10])
{
    ...
}
```

- **Matrici de şiruri**

Matrici multidimensionale

- Limbajul C permite utilizarea matricelor cu mai mult de două dimensiuni.
- Limita maximă a dimensiunilor este condiţionată cel mult de către versiunea de compilator.

Exemplu: O matrice de caractere cu 4 dimensiuni de mărime 10×4×5×10:

char mat[10][4][5][10];

necesită 10 x 4 x 5 x 10 = 2000 de octeţi.

În plus dacă matricea memorează alte date de tip:

- întreg - sunt necesari 4000 de octeţi;
- double - sunt necesari 16000 de octeţi.

Observaţii:

- Memoria necesară creşte foarte mult cu numărul dimensiunilor.
- În matricele multidimensionale calculatorul necesită timp pentru prelucrarea indicilor, deci accesul la elementele matricei va fi lent.
- Când se transmite o matrice multidimensională unei funcţii trebuie declarate toate dimensiunile, cu excepţia celei din extrema stângă.
- Variaţia indicilor este mai rapidă pentru cei din dreapta când se evaluează ordinea în memorie a elementelor matricei.

De exemplu, fie matricea:

int mat[4][3][6][5];

O funcţie care poate primi pe ***mat*** ca argument va fi definită în următorul mod:

```
void funct(int d[ ][3][6][5])
    {...
    }
```

Pointeri de indexare

- Numele unei matrice fără indice este un pointer la primul element al matricii.

Exemple:

1) *char p[10];*
/ următoarele instrucţiuni sunt identice*/*
p; <=> ***&p[0];***
Expresia: ***p==&p[0]*** *este adevărată;*

2) *int *p,i[10];*
p=i;
p[5]=100; <=> ****(p+5)=100***;
/ instrucţiuni identice ce au acelaşi rezultat*/*

3) La fel se întâmplă şi în cazul matricilor multidimensionale.
int a[10][10];
a <=> ***&a[0][0]***;
a[1][2] <=> ****(a+12)***;

Se evidenţiază următorul set de reguli:

- Pentru o matrice bidimensională:

tip a[lim1][lim2];

Elementul: **a[i][k] <=> *(a + i·lim2 + k)**

- Pentru o matrice tridimensională:

tip a[lim1][lim2][lim3];

Elementul: a[i1][i2][i3] <=> *(a+i1·lim2·lim3+i2·lim3+i3)

- Generalizarea pentru o matrice cu un număr oarecare de dimensiuni este evidentă.

Observaţii:

- Pointerii sunt adesea folosiţi pentru a avea acces în matrice deoarece aritmetica pointerilor este mai rapidă decât accesul prin indicii matricei.

- O matrice bidimensională poate fi redusă la un pointer la o matrice unidimensională (a se vedea *Matrice de şiruri* din cursul semestrului trecut).

Exemplu: funcţie care afişează un rând specificat al unei matrice:

```
int num[10][10]; /* variabilă globală*/
...
void afis_rand(int j)
{ int *p,t;
  p=&num[j][0]; /*preia adresa primului element
                  al rândului j*/
  for (t=0;t<10;t++) printf("%d",*(p+t));
}
```

Observație: **&num[j][0]** este echivalent cu **num[j]**

Îmbunătăţim exemplul anterior stabilind ca argument al funcţiei: rândul, lungimea sa şi un pointer la primul element al matricei.

```
void afis_rand(int j, int dim_rand, int *p)
{  int t;
    p=p+(j*dim_rand);
   for(t=0;t<dim_rand; t++)
        printf("%d",*(p+t));
}
```

Observație

- În C, o matrice tridimensională poate fi redusă la un pointer la o matrice bidimensională care poate fi redusă la un pointer la o matrice unidimensională.

- Prin generalizare, o matrice *n*-dimensională poate fi redusă la un pointer la o matrice *n-1* dimensională. Şi aşa mai departe până se ajunge la o matrice unidimensională.

De exemplu, fiind dată matricea:

int mat[lim_1][lim_2][lim_3]...[lim_n];

atunci elementul ***mat[i_1][i_2]...[i_n]*** este echivalent cu:

****(mat + i_1·lim_2·lim_3·...·lim_n +***
i_2·lim_3·lim_4·...·lim_n + ... + i_(n-1)·lim_n +
i_n)

Iniţializarea matricilor

Orice matrice poate fi iniţializată direct încă de la declarare:

tip nume_matrice[lim1][lim2]...[limn]={lista de valori};

Un lucru deja ştiut: cu cât este mai în dreapta un indice cu atât variază mai repede la enumerarea listei de valori în memoria sistemului de calcul (aspect esenţial la accesul prin pointeri).

Exemple:

1) În declaraţia:

int i[10]={1,2,3,4,5,6,7,8,9,10};

i[0] are valoarea 1;
i[9] are valoarea 10.

2) Matricile de caractere care conţin şiruri permit o iniţializare prescurtată:

char nume_matrice[mărime]=”şir de caractere”;

3) Declaraţia:

char str[12]=”Programul C”;

este similară cu:

char sir[12]={‘P’,’r’,’o’,’g’,’r’,’a’,’m’,’u’,’l’,’ ‘,’C’,’\0’};

Ca observaţie, a nu se omite la şirurile de caractere adăugarea lui ‘\0’, la sfârşit, atunci când nu se utilizează iniţializarea prescurtată (cu ghilimele).

4) Iniţializarea unei matrice bidimensionale:

```
int pătrat [3][2]   =       { 1, 1,
                              2, 4,
                              3, 5
                              };
```

5) Pentru iniţializarea matricelor fără mărime, dacă avem nevoie de un mesaj:

char er[18]=”Eroare de citire \n”;

sau altele mai lungi este destul de greu de numărat caracterele. Este corect (şi recomandabil) a se declara:

char er[ ]=”Eroare de citire \n”;

Astfel, instrucţiunea:

printf(“%s are mărimea %d \n”,er,sizeof er);

va afişa mesajul:

Eroare de citire are mărimea 18

6) Iniţializarea matricelor fără dimensiune nu este restrânsă la matricele unidimensionale. Pentru cele multidimensionale trebuie să se specifice toate dimensiunile în afară de cea din extrema stângă.
De exemplu:

```
int patrat[ ][2]   =   { 1, 1,
                         2, 4,
                         3, 5,
                       };
```

Avantajul este că se poate mări sau scurta tabelul (adică numărul de linii) fără a modifica dimensiunile matricei (cu referire la prima dimensiune din exemplu, care rămâne neprecizată).
Pe de altă parte ca dezavantaj, dacă programul este foarte mare, compilatorul acordă deja mai mult spaţiu decât este necesar matricei (nedefinită ca dimensiune) şi se ocupă mai multă memorie decât este strict necesară.

Cap. Pointeri
1. Introducere

- Un pointer este o variabilă care conţine o adresă din memorie, unde se află valoarea altei variabile. În consecinţă, pointerii se utilizează pentru a face referire la date cunoscute prin adresele lor.

Avantaje:

- Pointerii oferă posibilitatea de a modifica argumentele de apelare ale funcţiilor.
- Pointerii facilitează alocarea dinamică a memoriei.
- Pointerii pot îmbunătăţi eficienţa anumitor rutine.

2. Declararea pointerilor

Declararea variabilei de tip pointer se face respectând următorul format:

tip *nume_pointer;

Deoarece declaraţia uzuală a unei variabile este „*tip nume;*” putem spune că ***tip ** dintr-o declaraţie de pointer reprezintă *tip* dintr-o declaraţie obişnuită. Prin urmare ***tip ** va fi asociat unui tip nou, *tipul pointer*.

Observaţii:

- Tipul de bază (*tip* din construcţia *tip**) al pointerului defineşte tipul de variabilă către care indică acesta.

- Toată aritmetica pointerilor este creată relativ la tipul lor de bază, astfel încât este important să ca pointerul să fie corect declarat.

3. Operatori pentru pointeri

Asupra pointerilor pot fi executate un număr variat de operaţii, însă există doi operatori speciali pentru pointeri întâlniţi în expresii (unare) de tipul:

operator *operand*

Aceşti doi operatori sunt:

- ***operatorul de indirectare*** (*) → expresia care-l urmează este un **pointer** iar rezultatul este o ***lvaloare***.

- ***operatorul de obţinere a unui pointer*** (&) → operandul, asupra căruia acţionează acest operator, este o ***lvaloare*** iar rezultatul este un **pointer**.

Observaţii:

- Operatorii unari **&** şi ***** au prioritate faţă de toţi operatorii aritmetici, cu excepţia lui minus unar care are acelaşi ordin de precedenţă.
- Trebuie să ne asigurăm că variabilele folosite de tip pointer indică întotdeauna tipul corect de date. De exemplu, dacă se declară un pointer de tipul ***int***, compilatorul tratează conţinutul adresei indicate de pointer ca pe o variabilă de tip întreg, chiar şi în cazul în care acest lucru nu este adevărat.

Următorul program scris în C este compilat fără mesaj de eroare (mai puţin în C++) dar nu produce rezultatul dorit.

```
#include<stdio.h>
void main(void)
{ double x = 100.3, y;
   int *p;
   p=&x, /*forţează p să indice către un double*/
   y=*p; /*nu va lucra aşa cum ne aşteptăm*/
…
}
```

- În exemplul anterior nu se va atribui valoarea din ***x*** lui ***y*** deoarece ***p*** este un pointer de tip întreg şi vor fi transferaţi în ***y*** doar doi octeţi de informaţii (nu toţi cei 8 octeţi care formează în mod normal un număr în virgulă mobilă de tip double).

- Mai mult în C++ este interzisă convertirea unui pointer în altul fără utilizarea explicită a unui modelator de tip (cast).

- Utilizarea * cu atenție!

Observaţie

Simbolul * are în C un număr de patru utilizări, complet diferite, în expresii de genul:

- ***tip *nume*** – pentru definirea unui pointer.
- ****nume*** – unde exprimă o lvaloare.
- ***operand_1 * operand_2*** – unde exprimă o înmulţire.
- /* … */ – pentru comentariu.

4. Expresii cu pointeri

4.1. Instrucţiuni de atribuire pentru pointeri

Unui pointer i se poate atribui:

- o adresă
- un pointer

Ca observaţie, specificatorul de format folosit în funcţii de ieşire (cum ar fi *printf*) pentru afişarea valorii unui pointer (adrese) este %p.

Exemplu:

```
#include<stdio.h>
int main(int)
{ int x=100;
   int *p1, *p2;
   p1=&x ;
   p2=p1 ;
   printf(" %p", p2) ; /*afişează adresa lui x şi nu
                          valoarea sa*/
   printf(" %d", *p2); /*afişează valoarea lui x*/
}
```

4.2. Aritmetica pointerilor

- Operaţiile aritmetice ce se pot efectua cu ajutorul pointerilor sunt adunarea şi scăderea, la care se adaugă cele asociate de incrementare (**++**) si respectiv decrementare (**--**).

- **Trebuie precizat că aritmetica pointerilor este relativă la tipul lor de bază.**

Exemple:

1) *int *p1;* */*presupunem că p1 are valoarea 2000*/*
/*variabilele de tip int ocupă 2 octeţi*/

După expresia

p1++;

p1 va conţine 2002 şi nu 2001.
Iar dacă am fi avut expresia:

p1 - -;

şi p1 cu valoarea iniţială de 2000, atunci p1 va conţine după decrementare valoarea1998.

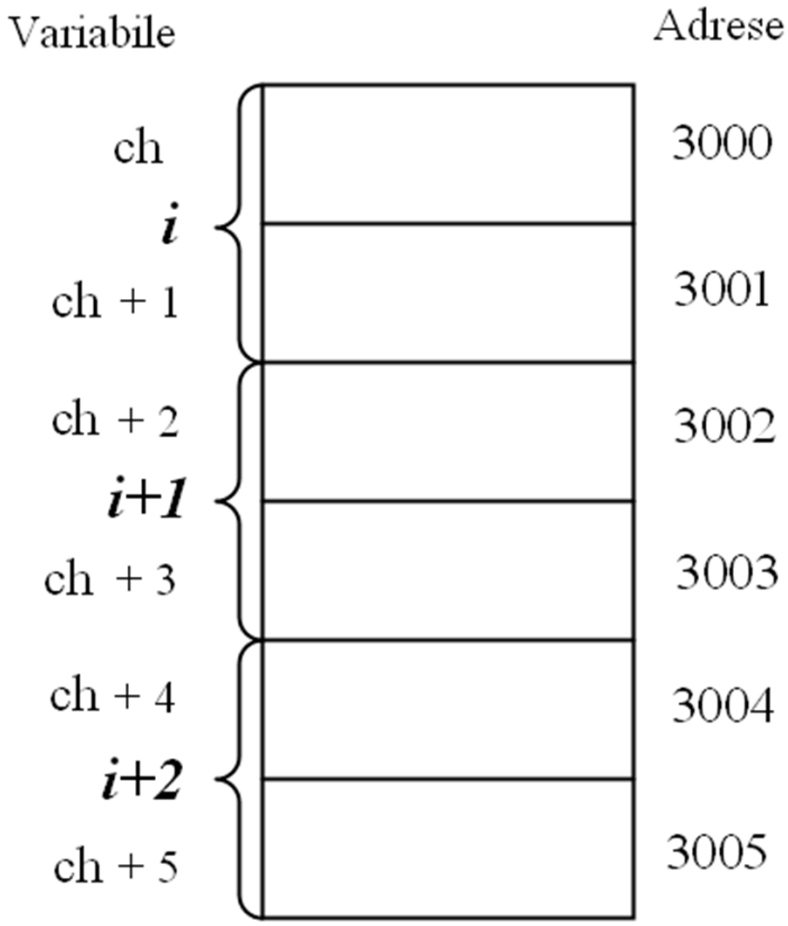
2) Cu privire la plasarea în memorie a unei variabile, prezentăm două situaţii şi precizăm că ele nu sunt simultane ci aparţin de programe diferite la momente de timp diferite:

*char *ch=3000;*

şi

*int *i=3000;* !

Primul pointer se incrementează succesiv de cinci ori, iar cel de-al doilea de două ori (într-un spațiu echivalent de memorie). Dacă am suprapune ipotetic cele două cazuri, zona locală de memorie ar arăta astfel: →



3) Nu suntem limitaţi (în cazul pointerilor) doar la operaţii de incrementare sau decrementare. Putem aduna sau scădea valori întregi la sau din pointeri.
Dacă ***p1*** şi ***p2*** sunt pointeri de acelaşi tip, atunci expresia:

p1=p2+12;

face ca **p1** să indice al 12-lea element de acelaşi tip cu **p2** după cel pe care îl indică p2.

4) Fie secvenţa de cod:

```
...
int x,y;
int *p;
...
```

Prezentăm câteva echivalenţe:

```
y=x+100;  ⇔   p=&x;
              y=*p+100;
```

```
x=y;  ⇔   p=&x;        ⇔   p=&y;
          *p=y;            x=*p;
```

```
x++;  ⇔   p=&x;
          (*p)++;
```

Observaţii:

- Ceilalţi operatori aritmetici (exceptând +, ++, -, şi - -) sunt interzişi în operaţii ce vizează pointeri (adrese de memorie).
- Nu se pot aduna sau scădea variabile sau constante de tip *float* sau *double* din pointeri (adică numere cu virgulă).
- Aritmetica pointerilor nu trebuie confundată cu aritmetica conţinutului locaţiilor de memorie ale căror adrese sunt valorile unor pointeri. În acest caz sunt permise toate operaţiile posibile contextului dat.

4.3. Utilizarea unui pointer cu mai multe tipuri de date

- Există cazuri în care acelaşi pointer se poate utiliza pentru mai multe tipuri de date, în acelaşi program, însă nu simultan. Acest lucru se poate realiza declarând iniţial pointerul ca fiind de tip *void*:

*void *nume;*

```
...
int x;
float y;
char c;
void *p;
...
p=&x;
...
p=&y;
...
p=&c;
...
```

Lui ***p*** i se pot atribui adrese din zone de memorie care pot conţine date de tipuri diferite (*int*, *float*, *char*, etc.) numai dacă se fac conversii explicite prin expresii *de tip cast*, pentru a preciza tipul datei spre care pointează un astfel de pointer. Conversia de tip *cast* respectă următorul format:

(tip) operand

Pentru exemplul de mai sus, expresia:

**p=10;*

nu este corectă, în timp ce

**(int *)p=10;*

este corectă.

Regula conversiilor implicite
(Reamintire din semestrul anterior)

Acţionează când un operator binar se aplică la 2 operanzi.

Paşii regulii:

- Mai întâi se convertesc operanzii de tip char şi enum la tipul int;
- Dacă operatorul curent se aplică la operanzi de acelaşi tip atunci rezultatul va fi de acelaşi tip. Dacă rezultatul reprezintă o valoare în afara limitelor tipului respectiv atunci rezultatul este eronat (au loc depăşiri).
- Dacă operatorul binar se aplică la operanzi diferiţi atunci se face o conversie înainte de execuţia operatorului conform paşilor următori:

- Dacă un operand este long double rezultă că şi celălalt se converteşte spre long double şi rezultatul este de tip long double.
- Altfel, dacă unul dintre operanzi este de tip double şi celălalt se converteşte spre double şi rezultatul este de tip double .
- Altfel, dacă unul dintre operanzi este de tip float rezultă că celălalt este tot float, iar rezultatul este float.
- Altfel, dacă unul dinre operanzi este unsigned long rezultă că şi celălalt se converteşte la unsigned long, iar rezultatul este de tip unsigned long.
- Altfel, dacă unul dintre operanzi este de tip long rezultă că şi celălalt se converteşte la tipul long, iar rezultatul operaţiei este tot de tip long.
- Altfel, dacă unul dintre operanzi este de tip unsigned şi celălalt int, rezultă automat că ambii devin unsigned, iar rezultatul este unsigned.

4.4. Compararea pointerilor

- **În ce situații putem compara doi pointeri?**

- Se pot compara doi pointeri printr-o expresie relaţională. Această comparaţie are totuşi justificare numai dacă cei doi pointeri pointează către elementele aceluiaşi tablou (matrice). Altfel, nu are relevanţă efectul comparaţiei atât timp cât compilatorul plasează variabilele la adrese diferite de memorie funcţie de disponibilităţile de moment ale maşinii de calcul (e foarte posibil ca la două rulări diferite ale aceluiaşi program, compilatorul să plaseze o variabilă în locaţii diferite de memorie).

Exemplu:

- Generăm o listă condusă pe principiul „ultimul venit primul plecat” (sau LIFO – Last Input First Output). Aceasta este o stivă care va stoca şi furniza valori întregi funcţie de numerele introduse. Astfel, dacă se introduce:
  - *V*aloare diferită de 0 sau -1 → aceasta se plasează în vârful stivei;
  - 0 → se scoate o valoare din stivă;
  - -1 → se opreşte programul.

```
#include<stdio.h>
#include<stdlib.h>
#define MARIME 50
void pune(int i);
int scoate(void);
int *b, *p, stiva[MARIME];
int main()
{       int valoare;
        b=stiva;  /*b indică baza stivei*/
        p=stiva;   /*iniţializează p*/
        do
          {printf("\n Introduceti valoarea:");
           scanf("%d", & valoare);
           if(valoare!=0)    pune(valoare) ;
                    else printf("\n Valoarea din varf este %d\n",
                                                scoate());
         }while(valoare !=-1) ;
}
```

```
void pune(int i)
{
        p++;
        if(p==(b+MARIME)) /* sau >=  */
          {printf("\nStiva
             supraincarcata");
            exit (1);
           }
        *p = i;
}
```

```
int scoate(void)
{ if(p==b) /* sau <=  */
    {printf("\n Stiva vida" );
     exit(1) ;
     }
   p - - ;
   return *(p+1) ;
}
```