Matrice şi şiruri – Laborator 4

Cea mai simplã matrice în C este şirul, care este o matrice de caractere terminate cu un caracter null.

C nu controlează limitele unei matrice. Puteți depăşi ambele margini ale unei matrice şi scrie în locul altor variabile sau peste codul programului. Este treaba dvs. Ca programator să asigurați controlul limitelor acolo unde este necesar. De exemplu, acest cod va fi compilat fără eroare, dar este incorect deoarece bucla for va determina că matricea numara să-şi depăşească limitele:

```
int numara[100], i;
/* aceasta determină numara să depăşească limitele */
  for (i=0; i<100;i++) numara[i] = i;
```

Crearea unui pointer la o matrice

Un pointer la primul element al matricei se creează simplu, specificând numele matricei, fără nici un indice. De exemplu, având

```
int proba[10];
```

puteți să creați un pointer la primul ei element utilizând numele proba. Următorul fragment de program atribuie lui p adresa primului element din proba.

```
int *p;
int proba[10];
p = proba;
```

Transmiterea matricelor unidimensionale către funcții

În C nu puteți transmite o matrice întreagă ca argument al unei funcții. Puteți, totuşi, să introduceți un pointer la o matrice specificând numele acesteia fără indice. De exemplu, următorul fragment de program introduce adresa lui i în funcția func1().

```
void main (void)
{
   int i[10];
   func1(i);
   .
}
```

Dacă funcția primeşte o matrice unidimensională, puteți să declarați parametrul sau formal în unul dintre aceste trei moduri: ca pointer, ca matrice cu dimensiune sau ca matrice fără dimensiune. De exemplu, pentru a primi pe x funcția cu numele func1() poate fi declarată astfel:

```
void func1(int *x) /* pointer */
{ . . .}
sau
void func1(int x[10]) /* matrice cu dimensiune */
{...}
sau
void func1(int x[]) /* matrice fara dimensiune */
{...}
```

Fiecare metodã spune compilatorului cã va primi un pointer cãtre un întreg. După cum vedeți, mãrimea matricei nu contează pentru funcție, deoarece C nu efectuează controlul limitelor. De fapt, din punctul de vedere al compilatorului, va fi corectã şi forma:

```
void func1 (int x[32])
{...}
```

deoarece compilatorul de C creează un cod care instruieşte func1() sã primească un pointer – el nu creează efectiv o matrice cu 32 de elemente.

Şiruri

Deşi C nu are date de tip şir, el permite constante şir. O constantã şir este o listã de caractere închise între ghilimele simple. De exemplu, 'salut'

```
#include <stdio.h>
#include <string.h>
void main (void)
{
  char s1[80], s2[80];
  gets(s1);
  gets(s2);
  printf("lungimi: %d %d\n", strlen(s1), strlen(s2));
  if(!strcmp(s1, s2)) printf("Sirurile sunt egale\n");
  strcat(s1, s2);
  printf("%s\n", s1);
  strcpy(s1, "Acesta este un test.\n");
  printf(s1);
  if(strchr("hello", 'e')) printf("e este in hello\n");
  if(strstr("te salut", "te")) printf("am gasit te");
}
```

Matrice bidimensionale

Următorul exemplu încarcã numere de la 1 la 12 într-o matrice bidimensionalã şi le afişează rând cu rând.

```
#include <stdio.h>
void main (void)
{
  int t, i, num[3][4];
  for(t=0; t<3; ++t)
    for(i=0; i<4; ++i)
      num[t][i] = (t*4)+i+1;
  /* acum le afiseaza */
  for(t=0; t<3; ++t)
    {
      for(i=0; i<4; ++i) printf("%3d ", num[t][i]);
      printf("\n");
    } }
```

Când o matrice bidimensională este utilizată ca un argument pentru o funcție, este transmis doar un pointer către primul element al matricei. Însă, parametrul care primeşte o matrice bidimensională trebuie să definească cel puțin numărul de coloane din dreapta deoarece compilatorul de C/C++ trebuie să ştie lungimea fiecărui rând pentru a indexa corect matricea. De exemplu, o funcție care primeşte o matrice bidimensională de întregi cu dimensiunea 10, 10 este declarată astfel:

```
void func1(int x[][10])
{...}
```

Puteți specifica şi dimensiunea din stânga dacă doriți, dar nu este necesar. În ambele cazuri, compilatorul trebuie să ştie mărimea celei din dreapta pentru a executa corect expresii ca:

```
x[2][4]
```

în interiorul funcției. Dacă mărimea rândului nu se cunoaşte, compilatorul nu poate să determine unde începe al treilea rând.

Matrice de şiruri

```
char matrice_siruri[30][80];
```

Matrice cu 30 de şiruri, fiecare din ele cu lungimea maximă de 79 de caractere. Este uşor de căpătat acces la un şir individual: specificați pur şi simplu doar indicele din stânga. De exemplu, următoarea instrucțiune apelează gets() pentru al treilea şir din matrice_siruri.

```
gets(matrice_siruri[2]);
```

Matrice multidimensionale

Când transmiteți o matrice multidimensională unei funcții, trebuie să declarați toate dimensiunile, în afară de cea din extrema stânga. De exemplu, dacă declarați o matrice cu m dimensiuni ca aceasta:

```
int m[4][3][6][5];
```

o funcție func1() care primeşte pe m va arăta astfel:

```
void func1(int d[][3][6][5])
{...}
```

Desigur, dacă doriți, puteți să introduceți prima dimensiune.

Pointeri de indexare

Numele unei matrice fără un indice este un pointer la primul element al matricei. Numele unei matrice fără un indice generează un pointer. Invers, un pointer poate sa aibă un indice ca şi cum ar fi fost declarat ca o matrice.

```
int *p, i[10];
p = i;
p[5] = 100; /* atribuire utilizand indice */
*(p+5) = 100; /* atribuire utilizand aritmetica pointerilor */
```

Presupunând că a este o matrice de întregi de 10 pe 10, aceste două instrucțiuni sunt echivalente:

```
a
&a[0][0]
```

La elementul 0, 4 al lui a se poate face referire în două moduri: ori prin indecşii matricei a[0][4], ori prin pointerul *(a+4).

În general, pentru orice matrice bidimensională
a[i][k]
este echivalentă cu
*(a+(i*lungime_rand)+k)

O matrice bidimensională poate fi redusă la un pointer la o matrice unidimensională. Utilizarea unei variabile de tip pointer separată este o cale uşoară de a folosi pointerii pentru a avea acces la elementele dintr-un rând al unei matrice bidimensionale.

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

Puteți generaliza această rutină stabilind ca argumente de apelare rândul, lungimea sa şi un pointer la primul element al matricei, aşa cum se arată aici:

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

Matricele mai mari de două dimensiuni pot fi reduse într-un mod similar. De exemplu, o matrice cu trei dimensiuni poate fi redusă la un pointer la o matrice bidimensională care poate fi redusă la un pointer la o matrice unidimensionala. În general, o matrice n-dimensională poate fi redusă la un pointer la o matrice cu (n-1) dimensiuni. Această nouă matrice poate fi redusă prin aceeaşi metodă. Procesul se încheie când se ajunge la o matrice unidimensională.

Inițializarea matricelor

C permite inițializarea matricelor în acelaşi timp cu declararea lor. Forma generală de inițializare:

Specificator_de_tip nume_matrice[marime1]…[marimeN] = {lista_de_valori};

Lista_de_valori este o listă separată prin virgule de un tip compatibil cu specificator_de_tip. Prima constantă este plasată în prima poziție a matricei, a doua constanta în a doua poziție şi aşa mai departe.

În următorul exemplu o matrice de întregi cu 10 elemente este inițializată cu numerele de la 1 la 10:

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

Aceasta înseamnă că i[0] va avea valoarea 1 iar i[9] va avea valoarea 10. Matricele de caractere care conțin şiruri permit o inițializare prescurtată care are formă:

```
char nume_matrice[marime] = "sir";
```

De exemplu, următorul fragment de cod inițializează sir cu fraza "Imi place C".

```
char sir[14] = “Imi place C++”;
```

Aceasta este similar cu a scrie:

```
char sir[14] = {‘I’, ‘m’, ‘i’, ‘ ‘, ‘p’, ‘l’, ‘a’, ‘c’, ‘e’, ‘ ‘, ‘C’, ‘+’, ‘+’, ‘\0’};
```

Când folosiți constante de tip şir, compilatorul asigurã automat caracterul null de sfârşit.

Matricele multidimensionale sunt inițializate la fel cu cele cu o singurã dimensiune. De exemplu, următorul cod inițializează patrat cu numerele de la 1 la 10 şi cu pãtratele lor.

```
int patrat[10][2] = {
        1,1,
        2,4,
        3,9,
        4,16,
        5,25,
        6,36,
        7,49,
        8,64,
        9,81,
        10,100
};
```

Inițializarea matricelor fãrã mãrime

Imaginați-vã cã folosiți o inițializare pentru a construi un tabel de mesaje de eroare, astfel:

```
char e1[18] = “Eroare de citire\n”;
char e2[19] = “Eroare de scriere\n”;
char e3[28] = “Nu pot sa deschid fisierul\n”;
```

Este greu de numărat manual caracterele din fiecare mesaj pentru a determina mãrimea exactã a matricei. Puteți lãsa compilatorul sã calculeze singur această mãrime utilizând matricele cu mãrime nedeterminatã.

Tabelul de mesaje devine:

```
char e1[] = “Eroare de citire\n”;
char e2[] = “Eroare de scriere\n”;
char e3[] = “Nu pot sa deschid fisierul\n”;
```

Cu aceste inițializări următoarea instrucțiune

```
printf (“%s are marimea %d\n”, e1, sizeof e1);
```

va afişa

```
Eroare de citire
 are marimea 18
```

Inițializarea matricei fãrã dimensiune vã permite schimbarea oricãrui mesaj fãrã teama cã utilizați incorect dimensiunile matricei.

Aici este prezentatã declararea lui patrat ca o matrice fãrã dimensiune:

```
int patrat[][2] = { 1,1, 2,4,
        …
        10,100
};
```

Instrucțiuni de atribuire pentru pointeri

```
#include <stdio.h>
void main (void)
{
        int x;
        int *p1, *p2;
        p1=&x;
        p2=p1;
        printf (“ %p”, p2); /*afiseaza adresa lui x, nu valoarea sa! */
}
```

Adresa lui x este afişatã utilizând specificatorul de formatare printf() %p, care face ca printf() sã afişeze adresa în formatul utilizat de calculatorul pe care se lucrează.

Aritmetica pointerilor

Există doar două operații aritmetice care se pot efectua cu pointeri: adunarea şi scăderea.

Să luăm un pointer de tip întreg p1 cu valoarea efectivă 2000. Să mai presupunem că întregii au doi octeți. După expresia:

```
p1++;
```

p1 va conține 2002, nu 2001. De fiecare dată când p1 este incrementat, el va indica spre următorul întreg.

Puteți aduna şi scădea întregi la sau din pointeri. Expresia:

```
p1 = p2 + 12;
```

face ca p1 să indice al doisprezecelea element de acelaşi tip cu p1 după cel pe care îl indică în mod curent.

Puteți să scădeți un pointer din altul pentru a determina numărul de obiecte de acelaşi tip care separa cei doi pointeri.

Compararea pointerilor

Fiind dați doi pointeri, p şi q, următoarea instrucțiune este perfect valabilă:

```
if (p<q) printf (“p indica o memorie mai mica decat q\n”);
```

Matrice de pointeri

Declararea unei matrice de pointeri de tipul int, cu mărimea 10 este:

```
int *x[10];
```

Pentru a atribui adresa unei variabile de tip întreg cu numele var elementului al treilea al matricei de pointeri, scrieți:

```
x[2] = &var;
```

Pentru a obține valoarea var, scrieți:

```
*x[2]
```

Dacă doriți să transmiteți o matrice de pointeri unei funcții, puteți folosi aceeaşi metodă pe care o folosiți pentru a transmite alte matrice – apelați funcția cu numele matricei fără nici un indice. De exemplu, o funcție care primeşte matricea x aratã astfel:

```
void afis_matrice (int *q[])
{
   int t;
   for (t=0; t<10;t++)
            printf (“%d “, *q[t]);
}
```

Rețineți cã q nu este un pointer cãtre întregi, ci un pointer cãtre o matrice de pointeri cãtre întregi. De aceea, trebuie sã declarați parametrul q ca o matrice de pointeri cãtre întregi.

Indirectare multiplã

Puteți sã aveți un pointer care sã indice pe un altul care indicã valoarea țintã. Aceastã situație este denumitã indirectare multipla sau pointeri catre pointeri.

O variabilã care este pointer cãtre pointer se declarã plasându-se încã un asterisc în fața numelui. De exemplu, urmãtoarea declarare spune compilatorului cã noulbilant este un pointer câtre un pointer de tipul float.

```
float **noulbilant;
```

Pentru a avea acces la valoarea doritã indicatã de un pointer la un pointer, trebuie sã aplicați de douã ori operatorul asterisc, ca în exemplul urmãtor:

```
#include <stdio.h>
void main (void)
{
   int x, *p, **q;
   x = 10;
   p = &x;
   q = &p;
   printf (“%d”, **q); /* afiseaza valoarea lui x */
}
```

Inițializarea pointerilor

După ce este declarat un pointer, dar înainte de a i se atribui o valoare, el poate sã conținã o valoare necunoscutã.

ATENȚIE: Dacã veți încerca sã folosiți un pointer înainte de a-i da o valoare validã, probabil cã vã veți bloca programul – eventual şi sistemul de operare al calculatorului dvs. – un tip de eroare foarte neplacutã!

Inițializare este şi urmãtorul tip de declarare de şir:

```
char *p = “salut lume”;
```

Pointerul p nu este o matrice. Motivul pentru care acest tip de inițializare merge se datorează modului de funcționare a compilatorului. Toate compilatoarele de C/C++ creează ceea ce se numeşte tabel de siruri, care este utilizat de calculator pentru a memora constantele de tip şir pe care le foloseşte programul. De aceea, instrucțiunea precedentã de declarare plasează adresa lui salut lume, aşa cum este ea stocatã în tabela de şiruri, în pointerul p. De-a lungul programului p poate fi utilizat ca oricare alt şir. De exemplu:

```
#include <stdio.h>
#include <string.h>
char *p = “salut lume”;
void main(void)
{
   register int t;
   /* afiseaza sirul inainte si inapoi */
   printf(p);
   for (t=strlen(p)-1; t>-1; t--) printf(“%c”, p[t]);
}
```

Pointeri către funcții

Adresa unei funcții este punctul de intrare în funcție. Trebuie să cunoaşteți puțin modul în care este compilată şi apelată o funcție. Pe măsură ce este compilată fiecare funcție, codul sursă este transformat în cod obiect şi se stabileşte un punct de intrare. În timpul rulării programului, atunci când este apelată o funcție, acest punct de intrare este apelat de limbajul maşină. De aceea, dacă un pointer conține adresa punctului de intrare, poate fi folosit pentru a apela acea funcție. Adresa unei funcții se obține utilizând numele funcției fără nici o parantezã sau argumente (similar cu modul în care se obține adresa unei matrice când este utilizat doar numele ei, fără indici).

```
#include <stdio.h>
#include <string.h>
void cauta (char *a, char *b, int (*comp)(const char *, const char *));
void main (void)
{
   char s1[80], s2[80];
   int (*p)(const char *, const char *);
   p = strcmp;
   gets(s1);
   gets(s2);
   cauta (s1, s2, p); }
void cauta (char *a, char *b, int (*comp)(const char *, const char *))
{
   printf(“testeaza egalitatea\n”);
   if(!(*comp)(a, b))printf(“egal”);
   else printf(“diferit”); }
```

Când este apelată funcția cauta(), sunt transmişi ca parametri doi pointeri de tip caracter şi unul către o funcție. Rețineți modul în care este declarat pointerul către funcție. Parantezele pentru *comp sunt necesare pentru compilator pentru a interpreta corect această instrucțiune.

Expresia: (*comp) (a,b) din interiorul lui cauta() apelează cu argumentele a şi b pe strcmp(), care este indicată de comp.

Puteți apela cauta() folosind direct strcmp(), aşa cum se prezintă aici:

```
cauta (s1, s2, strcmp);
```

Aceasta elimină cerința unei variabile în plus de tip pointer.

Funcții de alocare dinamică în C

Variabilelor globale li se alocă memorie în timpul compilării. Variabilele locale folosesc memoria stivă.

NOTĂ: Chiar dacă C++ acceptă pe deplin sistemul de alocare dinamică al lui C, el îşi defineşte propriul sistem, care conține mai multe îmbunătățiri față de cele din C.

Memoria alocată de funcțiile de alocare dinamică în C este obținută din heap – regiunea de memorie liberă care există între zona permanentă de memorie a programului dvs. şi cea stivă. Chiar dacă mărimea zonei heap este necunoscută, ea conține, în general, o cantitate destul de mare de memorie liberă.

Nucleul sistemului de alocare din C constă din funcțiile malloc() şi free(). Aceste funcții lucrează în pereche folosind zona de memorie liberă pentru a stabili şi a păstra o listă cu memoria disponibilă. Funcția malloc() alocă memorie iar free() o eliberează. Aceasta înseamnă că de fiecare dată când ii este cerută memorie lui malloc(), se alocă o zonă din memoria rămasă liberă. De fiecare dată când este apelată funcția free(), memoria este returnată sistemului. Orice program care foloseşte aceste funcții trebuie să includă fişierul antet STDLIB.H

Funcția malloc() are acest prototip:

```
void *malloc(size_t numar_de_octeti);
```

Aici, numar_de_octeti este numărul de octeți din memorie pe care doriți să îl alocați. Tipul size_t este definit în STDLIB.H aproximativ ca un întreg de tip unsigned. Funcția malloc() returnează un pointer de tipul void, ceea ce înseamnă că îl puteți atribui oricărui tip de pointer. După o apelare reuşită, malloc() returnează un pointer spre primul octet al regiunii de memorie alocate în memoria liberă. Dacă nu există suficientă memorie disponibilă pentru a satisface cerința lui malloc(), apare o blocare de alocare iar malloc() returnează null.

Fragmentul de cod prezentat aici alocă 1000 de octeți de memorie contiguă:

```
char *p;
p = malloc(1000); /*preia 1000 octeti */
```

După alocare, p indică spre primul din cei 1000 de octeți de memorie liberă. Observați că nu este necesar nici un modelator de tip pentru a atribui lui p valoarea returnată de malloc(). În C, un pointer de tip *void este convertit automat în tipul pointerului din partea stângă a instrucțiunii de atribuire. Această conversie automată nu are loc în C++. În C++ este necesar un modelator explicit de tip când este atribuit un pointer de tip *void unui alt tip de pointer. În C++, atribuirea precedentă trebuie să fie scrisă astfel:

```
p = (char *) malloc(1000);
```

Ca regulă generală, în C++ trebuie să folosiți un modelator de tip când atribuiți (sau altfel spus, convertiți) un tip de pointer într-altul. Aceasta este una dintre puținele diferențe fundamentale între C şi C++.

Următorul exemplu alocă spațiu pentru 50 de întregi. Rețineți utilizarea lui sizeof pentru asigurarea portabilității.

```
int *p;
p = malloc(50*sizeof(int));
```

Deoarece memoria nu este infinită, când alocați memorie trebuie să verificați valoarea returnată de malloc() înainte de a folosi pointerul, pentru a vă asigura că nu este null.

Utilizând un pointer null sigur veți bloca programul. Modul corect de alocare a memoriei şi de testare a validității unui pointer este ilustrat în acest fragment de cod:

```
if(!(p=malloc(100))
{
   printf("Depasire de memorie.\n");
   exit(1);
}
```

Funcția free() este opusă lui malloc() deoarece ea returnează în sistem memoria alocată anterior. O dată memoria eliberată, ea poate să fie refolosită de o apelare ulterioară a lui malloc(). Funcția free() are următorul prototip:

```
void free(void *p);
```

Aici, p este un pointer spre memoria care a fost alocată anterior folosind malloc(). Este esențial să nu apelați niciodată free() cu un argument impropriu; aceasta v-ar distruge lista de memorie liberă.

Probleme ale pointerilor

Exemplul clasic de greşeală pentru pointeri este pointerul neinițializat.

```
/* Acest program este greşit. */
void main(void)
{
  int x, *p;
  x = 10;
  *p = x;
}
```

Acest program atribuie 10 unei locații de memorie necunoscute. Iată de ce. De vreme ce pointerului p nu i s-a dat o valoare, el conține una necunoscută atunci când are loc atribuirea *p = x, ceea ce face ca valoarea din x să fie scrisă într-o locație de memorie necunoscută. Acest tip de problemă scapă de obicei neobservată când programul este mic deoarece cele mai mari şanse sunt ca p să conțină o adresă "sigură" – una care nu intra în zona de program, de date ori a sistemului de operare. Dar, pe măsură ce programul dvs. se măreşte, creşte şi probabilitatea ca p să conțină ceva vital. În cele din urmă, programul se va opri. Soluția este să vă asigurați mereu, înainte de a utiliza un pointer, că acesta indica spre ceva valid.

Probleme

1. Scrieti un program care citeste de la tastatura 2 matrici patratice de numere si calculeaza suma si produsul lor.

2. Scrieti un program care citeste de la tastatura o matrice de numere si inlocuieste cel mai mare element de pe fiecare coloana cu suma elementelor de pe coloana respectiva. Daca pe o coloana valoarea maxima se atinge in mai multe locuri, se va inlocui doar una din aparitii.

3.Sa se citeasca o matrice de siruri. Sa se gaseasca si sa se tipareasca sirul din matrice care are numarul maxim de consoane.

4. Pentru o matrice de elemente intregi, alocata dinamic, sa se scrie urmatoarele functii ce primesc parametrii prin pointeri:

- citire - citeste elementele matricii pe coloane
- tiparire - tipareste elementele matricii pe linii
- det_maxim - determina si afiseaza valoare si pozitia elementului maxim din matrice
- constr_tab - construieste un tablou unidimensional declarat alocat dinamic, ale carui elemente sunt sumele elementelor de pe cate o linie a matricii
- interschimbare - interschimba elementele de pe doua coloane ale matricii cu indicii cititi
- cauta - cauta in matrice o valoare citita, afisandu-i pozitia.

5. Sa se scrie un program in care utilizand o matrice de siruri:

- citeste cuvintele tastate fiecare pe cate un rand nou pana la introducerea unui cuvant vid si le retine in matrice
- afiseaza cuvantul cel mai scurt
- afiseaza cuvintele ce incep cu o vocala

6. Scrieți o funcție care stabileşte dacă un cuvânt dat se găseşte sau nu într-un tablou de cuvinte.

7. Sa se scrie un program pentru ordonarea unui vector de numere prin determinarea repetată a valorii maxime dintr-un vector şi schimbarea cu ultimul element din vector. Sa se scrie o funcție care determină poziția valorii maxime dintr-un vector.

8.Adăugați câte o funcție pentru fiecare din prelucrările:

citeşte o matrice
returnează suma elementelor unei matrici
însumează două matrici într-o a treia
înmulțeşte două matrici într-o a treia.

9. Se citesc trei şiruri s1, s2 şi s3. Să se afişeze şirul obținut prin înlocuirea în s1 a tuturor aparițiilor lui s2 prin s3. (Observație: dacă s3 este şirul vid, din s1 se vor şterge toate subşirurile s2).

.

10. Sa se scrie un program pentru citirea unor nume, se va aloca dynamic memoria pentru fiecare şir (în funcție de lungimea şirului citit) şi se vor memora adresele şirurilorîntr-un vector de pointeri. În final se vor afişa numele citite, pe baza vectorului de pointeri.

Studiu si exercitii suplimentare(facultative):

http://www.cprogramming.com/tutorial/c/quiz/quiz6.html
http://www.cprogramming.com/tutorial/c/quiz/quiz8.html
http://forum.junowebdesign.com/articles-tutorials-guides/14502-quiz-10-common-pointer-memory-allocation-pitfalls-c-c.html

http://oreilly.com/catalog/pcp3/chapter/ch13.html
http://www.sethi.org/classes/comp217/lab_notes/lab_10_pointers.html