Tablouri. Particularizare - vectori. Aplicaţii

Responsabili:

- Darius Neaţu (CA 2019-2020) [mailto:neatudarius@gmail.com]
- Dorinel Filip (CA 2019-2020) [mailto:ion_dorinel.filip@cti.pub.ro]
- Andrei Pârvu [mailto:andrei.parvu@cti.pub.ro]

Teste: Link [https://drive.google.com/file/d/11NU3wwFm7CAlkjqNBpWrYDTWqoRqEoqe/view?usp=sharing]

Obiective

În urma parcurgerii acestui laborator, studentul va fi capabil:

- să declare şi să iniţializeze vectori (din declaraţie şi prin structuri iterative)
- să implementeze algoritmi simpli de sortare şi căutare pe vectori
- să folosească practici recunoscute şi recomandate pentru scrierea de cod sursă care implică lucrul cu vectori
- să recunoască şi să evite erorile comune de programare legate de aceste structuri
- să aplice lucrul cu vectori pentru rezolvarea unor probleme cu dificultate medie

Noţiuni teoretice

Vectori

Printr-un vector se înţelege `o colecţie liniară şi omogenă` de date. Un vector este liniar pentru că datele(elementele) pot fi accesate în mod unic printr-un `index`. Un vector este, de asemenea, omogen, pentru că toate elementele sunt de acelaşi `tip`. În limbajul C, indexul este un număr întreg pozitiv şi indexarea se face începând cu 0.

Declaraţia unei variabile de tip vector se face în felul următor:

```
<tip_elemente> <nume_vector>[<dimensiune>];
```

De exemplu, avem următoarele declaraţii de vectori:

```
int a[100];
float vect[50];

#define MAX 100
...
unsigned long numbers[MAX];
```

Este de remarcat că vectorul este o `structură statică`: dimensiunea acestuia trebuie să fie o constantă la compilare şi nu poate fi modificată în cursul execuţiei programului. Astfel, programatorul trebuie să estimeze o dimensiune maximă pentru vector, şi aceasta va fi o limitare a programului. De obicei, se folosesc constante simbolice (ca în ultimul exemplu) pentru aceste dimensiuni maxime, pentru ca ele să poată fi ajustate uşor la nevoie. De asemenea, în cadrul unei declaraţii, se pot iniţializa cu valori constante componente ale vectorului, iar în acest caz, dimensiunea vectorului poate rămâne neprecizată (compilatorul o va determina din numărul elementelor din listă). De exemplu:

```
int a[3] = {1, 5, 6};                    // Toate cele 3 elemente sunt initializate
float num[] = {1.5, 2.3, 0.2, -1.3};     // Compilatorul determina dimensiunea - 4 - a vectorului
unsigned short vect[1000] = {0, 2, 4, 6}; // Sunt initializate doar primele 4 elemente
```

În cazul special în care specificăm dimensiunea şi doar un singur element la initializare, primul element va fi cel specificat, iar toate celelalte elemente ale vectorului vor fi iniţializate la `0`:

```
char string[100] = {97}; // Sirul va fi initializat cu: 97 (caracterul 'a') pe prima poziţie şi 99 de 0
int v[100] = {0};        // Vectorul va fi umplut cu 0.
```

Este important de remarcat faptul că elementele **neiniţializate** pot avea valori **oarecare**. La alocarea unui vector, compilatorul nu efectuează nici un fel de iniţializare şi nu furnizează nici un mesaj de eroare dacă un element este folosit înainte de a fi iniţializat. **Un program corect va iniţializa**, în orice caz, fiecare element înainte de a-l folosi. Elementele se accesează prin expresii de forma `<nume_vector>[<indice>]`. De exemplu, putem avea:

```
char vect[100];
vect[0] = 1;
vect[5] = 10;

int i = 90;
vect[i] = 15;
vect[i + 1] = 20;
```

Stil de programare

Exemple de programe

Citirea unui vector de intregi de la tastatura:

```
int main()
{
  int a[100], n, i; /* vectorul a are maxim 100 de intregi */

  scanf("%d", &n); /* citeste nr de elemente vector */

  for (i = 0; i < n; i++) {
    scanf("%d", &a[i]); /* citire elemente vector */
  }

  for (i = 0; i < n; i++) {
    printf("%d ", a[i]); /* scrie elemente vector */
  }

  return 0;
}
```

Generarea unui vector cu primele n numere Fibonacci:

```
#include <stdio.h>
int main()
{
  long fib[100] = {1, 1};
  int n, i;

  printf("n = ");
  scanf("%d", &n);

  for (i = 2; i < n; i++) {
    fib[i] = fib[i - 1] + fib[i - 2];
  }
  for (i = 0; i < n; i++) {
    printf("%ld ", fib[i]);
  }

  return 0;
}
```

Deşi modul în care se manifestă erorile din program poate fi surprinzător şi imprevizibil, cauzele care produc aceste erori sunt destul de comune şi pot fi grupate în mai multe categorii. Câteva dintre acestea sunt prezentate mai jos

- Depăşirea limitelor indicilor (index out of bounds) este o eroare frecventă, ce poate duce la blocarea programului sau a sistemului şi poate fi evitată prin verificarea încadrării în intervalul valid.
- Indici folosiţi greşit în bucle imbricate (index cross-talk). Sunt multe cazuri în care pe un nivel al buclei se foloseşte, de exemplu vect[i], şi pe nivelul imbricat vect[j], când de fapt se dorea folosirea lui i. Mare atenţie şi în astfel de cazuri!

Definiţi dimensiunile prin constante şi folosiţi-le pe acestea în locul tastării explicite a valorilor în codul sursă. Astfel veţi evita neconcordanţe în cod dacă doriţi ulterior să modificaţi dimensiunile şi uitaţi să modificaţi peste tot prin cod.

```
#define MAX   100

int vect[MAX];
```

va fi de preferat în locul lui

```
int vect[100];
```

Verificaţi că indicii se încadrează între marginile superioară şi inferioară a intervalului de valori valide. Acest lucru trebuie în general făcut în cazul în care datele provin dintr-o sursă externă: citite de la tastatură sau pasate ca parametri efectivi unei funcţii, de exemplu.

Exemplu:

```
// program care citește un index și o valoare, și atribuie valoarea elementului din vector care se găsește la poziția respectivă
#include <stdio.h>
#define N 10

int main()
{
  int i, val;
  int v[N];

  scanf("%d%d", &i, &val);

  /* !!! Verific daca indexul este valid */
  if (i >= 0 && i < N) {
   v[i] = val;
  } else {
    printf("Introduceti un index >= 0 si < %d\n", N);
  }

  return 0;
}
```

Folosiţi comentarii pentru a explica ce reprezintă diverse variabile. Acest lucru vă va ajuta atât pe voi să nu încurcaţi indici, de exemplu, cât şi pe ceilalţi care folosesc sau extind codul vostru.

Exemplu:

```
#include <stdio.h>
#define N 100

int main()
{
  int v[N];
  int i, j; /* indecsii elementelor ce vor fi interschimbate */
  int aux;  /* variabila ajutatoare pentru interschimbare */

  /*... initializari */

  /* Interschimb */
  aux = v[i];
  v[i] = v[j];
  v[j] = aux;

  return 0;
}
```

Aplicaţii cu vectori

Căutări

Căutare secvenţială

Când avem de a face cu un vector nesortat (şi nu numai în acest caz), cea mai simplă abordare pentru a găsi o valoare, este căutarea secvenţială. Cu alte cuvinte, se compară, la rând, fiecare valoare din vector cu valoarea căutată. Dacă valoarea a fost găsită, căutarea se poate opri (nu mai are sens să parcugem vectorul până la capăt, dacă nu se cere acest lucru explicit).

Exemplu:

```
#define MAX 100

...

int v[MAX], x, i;

/* initializari */
...

int found = 0;
for (i = 0; i < MAX; i++) {
  if (x == v[i]) {
    found = 1;
    break;
  }
}

if (found) {
   printf("Valoarea %d a fost gasita in vector\n", x);
} else {
   printf("Valoarea %d nu a fost gasita in vector\n", x);
}
...
```

Căutare binară iterativă

Dacă vectorul pe care se face căutarea este sortat, algoritmul mai eficient de folosit în acest caz este căutarea binară. Presupunem că vectorul este sortat crescător (pentru vectori sortaţi descrescător, raţionamentul este similar).

Valoarea căutată, x, se compară cu valoarea cu indexul N/2 din vector, unde N este numărul de elemente. Dacă x este mai mic decât valoarea din vector, se caută în prima jumătate a vectorului, iar dacă este mai mare, în cea de-a doua jumătate. Căutarea în una dintre cele două jumătăţi se face după acelaşi algoritm.

Conceptual, căutarea binară este un algoritm recursiv, dar poate fi implementat la fel de bine într-un mod iterativ, folosind indecşii corespunzători bucăţii din vector în care se face căutarea. Aceşti indecşi se modifică pe parcursul algoritmului, într-o buclă, în funcţie de comparaţiile făcute. Evoluţia algoritmului este ilustrată în imaginea de mai jos.

Pseudocodul pentru căutarea binară:

```
// cauta elementul x in vectorul sortat v, intre pozitiile 0 si n-1  si returneaza pozitia gasita sau -1
int binary_search(int n, int v[NMAX], int x) {
  int low = 0, high = n - 1;

  while (low <= high) {
    // De ce preferăm această formă față de (low + high) / 2 ?
    int middle = low + (high - low) / 2;

    if (v[middle] == x) {
```

```
      // Am gasit elementul, returnam pozitia sa
      return middle;
    }

    if (v[middle] < x) {
      // Elementul cautat este mai mare decat cel curent, ne mutam in jumatatea
      // cu elemente mai mari
      low = middle + 1;
    } else {
      // Elementul cautat este mai mic decat cel curent, ne mutam in jumatatea
      // cu elemente mai mici
      high = middle - 1;
    }
  }

  // Elementul nu a fost gasit
  return -1;
}
```

Preferăm calcularea mijlocului intervalului [low, high] folosind formula `x = low + (high - low) / 2` deoarece formula perfect analogă `x = (low + high) / 2` poate da overflow pentru valori mari ale low si high. De altfel, acest bug a existat în biblioteca Java timp de 9 de ani. Puteți citi mai mult despre asta în acest articol [http://googleresearch.blogspot.ro/2006/06/extra-extra-read-all-about-it-nearly.html].

Sortări

Bubble Sort

Metoda bulelor este cea mai simplă modalitate de sortare a unui vector, dar şi cea mai ineficientă. Ea funcţionează pe principiul parcurgerii vectorului şi comparării elementului curent cu elementul următor. Dacă cele două nu respectă ordinea, sunt interschimbate. Această parcurgere este repetată de suficiente ori până când nu mai există nici o interschimbare în vector.

Sortarea prin selecţie

Sortarea prin selecţie oferă unele îmbunătăţiri în ceea ce priveşte complexitatea, însă este departe de a fi considerat un algoritm eficient. Presupunând că se doreşte sortarea crescătoare a vectorului, se caută minimul din vector, şi se interschimbă cu primul element - cel cu indexul 0. Apoi se reia acelaşi procedeu pentru restul vectorului. Motivul pentru care algoritmul de sortare prin selecţie este mai eficient este acela că vectorul în care se caută minimul devine din ce în ce mai mic, şi, evident, căutarea se face mai repede la fiecare pas.

Studiul unor algoritmi mai avansaţi de sortare, precum şi studiul complexităţii lor nu constituie obiectul acestui laborator. Acestea se vor relua mai detaliat în cadrul altor cursuri (AA/PA).

Exerciții Laborator CB/CD

1. Primul exercitiu presupune modificarea/adaugarea de instructiuni unui cod existent pentru a realiza anumite lucruri. In momentul actual programul aduna la elementul curent vecinul din dreapta sa (daca acesta exista).
    - Nu uitati ca trebuie sa utilizam un coding style [http://ocw.cs.pub.ro/courses/programare/coding-style] adecvat atunci cand scriem sursele.

ex1.c

```
#include <stdio.h>

#define N 100

void sum_right_neighbour(int v[N], int n)
{
    int i;

    for (i = 0; i < n - 1; i++) {
        v[i] += v[i+1];
    }
}

void print_vector(int v[N], int n)
{
    int i;

    for (i = 0; i < n; i++) {
        printf("%d ", v[i]);
    }
    printf("\n");
}

int main(void)
{
    int v[N] = {1, 2, 3, 4, 5};

    print_vector(v, 5);
    sum_right_neighbour(v, 5);
    print_vector(v, 5);

    return 0;
}
```

Cerinte:

- Sa se creeze o functie care sa adauge la fiecare element din vector vecinul din stanga sa (daca acesta exista).
- Sa se creeze o functie care sa construiasca un alt vector ce contine in vector[i] produsul tuturor elementelor, mai putin elementul de pe pozitia i din vectorul initial.
- Se citesc de la tastatura caractere (se va citi cate un caracter pe rand). In functie de valoarea acestuia se va realiza una din actiunile urmatoare:
  - q - iesire din program
  - m - eliminare element minim
  - M - eliminare element maxim
  - p - printare vector

Următoarele două probleme vă vor fi date de asistent în cadrul laboratorului.

Checker laborator 5 [https://drive.google.com/file/d/16vwYGOWmvHDvj6F83lRFXXXtjbKo5ENl/view?usp=sharing] Tutorial folosire checker laborator [https://ocw.cs.pub.ro/courses/programare/checker]

Pentru utilizarea checkerului:

- Se va scrie cate un fisier sursa pentru fiecare problema;
- La finalul fiecarui printf utilizat pentru afisarea rezultatului trebuie sa existe un newline;
- Sursa nu trebuie sa contina alte printf-uri in afara de cele care scriu rezultatul asteptat la stdout.
- Se va dezarhiva arhiva specifica exercitiului;
- In directorul curent se afla checkerul, executabilul generat, folderele de input si output specifice problemei;
- Se va rula "bash checker.sh <executabil>" unde <executabil> este numele executabilului generat;

Exerciţii de Laborator

1. [4 x 1p] Se consideră un vector x cu n < 100 componente. Să se determine:
   a. valoarea componentei minime;
   b. poziţia componentei maxime;
   c. media aritmetică a componentelor;
   d. numărul componentelor mai mari ca media aritmetică.
   Dimensiunea vectorului, n, şi elementele vectorului se citesc de la tastatură (cu scanf()).
   Programul se va testa cu fişierul de intrare de mai jos (in.txt), utilizând comanda:

```
./pb1 <in.txt >out.txt
```

   Fişierul de intrare, in.txt:

```
50
95 59 50 15 53 44 91 7 86 16 73 57 27 54 97 62 59 5 98 61 99 22 22 84 17 96 13 96 5 50 62 53 61 12 68 14 8 59 74 95 27 47 52 54 53 2 68 13 7 19
```

   Fişierul de ieşire asociat, out.txt:

```
min = 2
poz_max = 20
ma = 49.22
gt_ma = 30
```

2. [2p] Să se implementeze sortarea prin selecție pe un vector citit de la tastatură.
3. [2p] Citindu-se doi vectori A și B să calculeze diferența lor (elementele care sunt în A, dar nu sunt în B).
   Exemplu:

```
a = [6, 4, 2, 1, 7]
b = [4, 6, 3, 1]

Rezultat = [2, 7]
```

4. [2p] Având un vector de N elemente să se determine subsecvența sa de suma maximă. O subsecvență a unui vector reprezintă un subșir de elemente ale acestuia aflate pe poziții consecutive.

Exemplu:

```
4
3 -4 2 3 // raspuns 5
4
-3 7 -3 4 // raspuns 8
3
-1 -2 -3 // raspuns -1
```

BONUS

- [2p] Implementați adunarea a două numere mari. Un număr mare va fi reprezentat ca un vector A, unde A[0] va fi numărul de cifre ale numărului, iar elementul A[i] va conține a i-a cea mai nesemnificativă cifră.
Exemplu:

```
9420 // -> A[] = {4, 0, 2, 4, 9}
845 // -> B[] = {3, 5, 4, 8}
// Suma
10265 // -> C[] = {5, 5, 6, 2, 0, 1}
```

- [2p] Se citește de la tastatură un vector V sortat crescator de lungime N și apoi un șir de perechi (x, y) de numere naturale terminat cu -1. Pe măsura citirii acestor numere, să se determine **eficient** numărul de elemente din V cuprinse între x și y (x < y). Folosiți o căutare binară modificată.
Exemplu:

```
5
2 10 20 50 80
5 15
1 // output
10 20
2 // output
19 91
3 // output
-1
```

Probleme laborator 14:00-16:00 [https://drive.google.com/open?id=1SZfdFDKXvJGSfs0FILosxTiBYVksDn3D]

Referinţe

- Cheatsheet vectori [https://github.com/cs-pub-ro/ComputerProgramming/blob/master/Laboratories/Lab5/Lab5.pdf]
- Wikipedia - Căutare binară [http://en.wikipedia.org/wiki/Binary_search]
- Wikipedia - Algoritmi de sortare [http://en.wikipedia.org/wiki/Sorting_algorithm]

programare/laboratoare/lab05.txt · Last modified: 2020/11/10 17:34 by marius.vintila