Capitolul IB.10. Alocarea memoriei în limbajul C

Cuvinte cheie
Clase de memorare, alocare statică, alocare dinamică, variabile auto, variabile locale, variabile globale, variabile register, funcții standard, vectori de pointeri, sructuri alocate dinamic

IB.10.1. Clase de memorare (alocare a memoriei) în C

Clasa de memorare arată când, cum şi unde se alocă memorie pentru o variabilă.

Orice variabilă are o clasă de memorare care rezultă fie din declarația ei, fie implicit din locul unde este definită variabila.

Zona de memorie utilizată de un program C cuprinde 4 subzone:
- Zona text: în care este păstrat codul programului
- Zona de date: în care sunt alocate (păstrate) variabilele globale
- Zona stivă: în care sunt alocate datele temporare (variabilele locale)
- Zona heap: în care se fac alocările dinamice de memorie

Moduri de alocare a memoriei:

- **Statică:** variabile implementate în zona de date - globale

Memoria este alocată la compilare în segmentul de date din cadrul programului şi nu se mai poate modifica în cursul execuției. Variabilele externe, definite în afara funcțiilor, sunt implicit statice, dar pot fi declarate *static* şi variabile locale, definite în cadrul funcțiilor.

- **Auto**: variabile implementate în stivă - locale

Memoria este alocată automat, la activarea unei funcții, în zona stivă alocată unui program şi este eliberată automat la terminarea funcției. Variabilele locale unui bloc (unei funcții) şi parametrii formali sunt implicit din clasa *auto*. Memoria se alocă în stiva ataşată programului.

- **Dinamică**: variabile implementate în heap

Memoria se alocă dinamic (la execuție) în zona *heap* ataşată programului, dar numai la cererea explicită a programatorului, prin apelarea unor funcții de bibliotecă (*malloc, calloc, realloc*). Memoria este eliberată numai la cerere, prin apelarea funcției *free*

- **Register**: variabile implementate într-un registru de memorie

IB.10.2. Clase de alocare a memoriei: Auto

Variabilele locale unui bloc (unei funcții) şi parametrii formali sunt implicit din clasa *auto*.
Durata de viață a acestor variabile este temporară: memoria este alocată automat, la activarea blocului/funcției, în zona stivă alocată programului şi este eliberată automat la ieşirea din bloc/terminarea funcției. Variabilele locale NU sunt inițializate! Trebuie să le atribuim o valoare inițială!

Exemplu:

```
int doi() {
      int x = 2;
      return x;
}
int main() {
      int a;
      {
            int b = 5;
            a = b*doi();
```

```
      }
      printf("a = %d\n", a);
      return 0;

}

Conţinut stivă:
(x)  2
(b)  5
(a) 10
```

IB.10.3. Clase de alocare a memoriei: Static

Memoria este alocată la compilare în segmentul de date din cadrul programului şi nu se mai poate modifica în cursul execuției.
Variabilele globale sunt implicit *statice* (din clasa *static*).
Pot fi declarate *static* şi variabile locale, definite în cadrul funcțiilor, folosind cuvântul cheie static.
O variabilă sau o funcție declarată (sau implicit) *static* are durata de viață egală cu cea a programului. In consecință, o variabilă *statică* declarată într-o funcție îşi păstrează valoarea între apeluri succesive ale funcției, spre deosebire de variabilele *auto* care sunt realocate pe stivă la fiecare apel al funcției şi pornesc de fiecare dată cu valoarea primită la inițializarea lor (sau cu o valoare imprevizibilă, dacă nu sunt inițializate).
Exemple:

```
int f1() {
   int x = 1;          /*Variabilă locală, inițializată cu 1 la fiecare
                       apel al lui f1*/
   ......
}

int f2() {
   static int y = 99; /*Variabilă locală statică, inițializată cu 99
                       doar  la primul apel al lui f2; valoarea ei este
                       reținută pe parcursul apelurilor lui f2*/
   ......
}

int f() {
      static int nr_apeluri=0;
      nr_apeluri++;
      printf("funcția f() este apelata pentru a %d-a oara\n", nr_apeluri);
      return nr_apeluri;
}
int main() {
      int i;
      for (i=0; i<10; i++) f();   //f() apelata de 10 ori
      printf("functia f() a fost apelata de %d ori.", f()); // 11 ori!!
      return 0;
}
```

Observații:
Variabilele locale statice se folosesc foarte rar în practica programării (funcția de bibliotecă *strtok* este un exemplu de funcție cu o variabilă statică).

- Variabilele statice pot fi inițializate numai cu valori constante (pentru că inițializarea are loc la compilare), dar variabilele *auto* pot fi inițializate cu rezultatul unor expresii (pentru că inițializarea are loc la execuție).
  Exemplu de funcție care afişează un întreg pozitiv în cod binar, folosind câturile împărțirii cu puteri descrescătoare ale lui 10:

```
// afisare intreg in binar
void binar ( int x) {
   int n=digits(x);   //functie care intoarce nr-ul de cifre al lui x
   int d=pw10 (n-1);  //functie care calculeaza 10 la o putere intreaga
   while ( x >0) {
      printf("%d",x/d); //scrie catul impartirii lui x prin d
      x=x%d;
      d=d/10;           //continua cu x = x%d si d = d/10
   }
}
```

- Toate variabilele externe (şi statice) sunt automat inițializate cu valori zero (inclusiv vectorii).

Cuvântul cheie *static* face ca o variabilă globală sau o funcție să fie *privată(proprie)* unității unde a fost definită: ea devine inaccesibilă altei unități, chiar prin folosirea lui *extern*.

- Cantitatea de memorie alocată pentru variabilele cu nume rezultă din tipul variabilei şi din dimensiunea declarată pentru vectori. Memoria alocată dinamic este specificată explicit ca parametru al funcțiilor de alocare, în număr de octeți.

Memoria neocupată de datele statice şi de instrucțiunile unui program este împărțită între *stivă* şi *heap*.

Consumul de memorie *stack* (*stiva*) este mai mare în programele cu funcții recursive (număr mare de apeluri recursive).

Consumul de memorie *heap* este mare în programele cu vectori şi matrice alocate (şi realocate) dinamic.

De observat că nu orice vector cu dimensiune constantă este un vector *static*; un vector definit într-o funcție (alta decât *main*) nu este static deoarece nu ocupă memorie pe toată durata de execuție a programului, deşi dimensiunea sa este stabilită la scrierea programului. Un vector definit într-o funcție este alocat pe stivă, la activarea funcției, iar memoria ocupată de vector este eliberată automat la terminarea funcției.

Sinteză variabile locale / variabile globale

O sinteză legată de variabilele locale şi cele globale din punct de vedere al duratai de viață vs. domeniu de vizibilitate este dată în tabelul următor:

	Variabile globale	Variabile locale
Alocare	Statică; la compilare	Auto; la execuție bloc
Durata de viață	Cea a întregului program	Cea a blocului în care e declarată
Inițializare	Cu zero	Nu se face automat

IB.10.4. Clase de alocare a memoriei: Register

A treia clasă de memorare este clasa *register*, pentru variabile cărora li se alocă registre ale procesorului şi nu locații de memorie, pentru un timp de acces mai bun.

O variabilă declarată *register* solicită sistemului alocarea ei într-un registru maşină, dacă este posibil.

De obicei compilatorul ia automat decizia de alocare a registrelor maşinii pentru anumite variabile *auto* din funcții. Se utilizează pentru variabile "foarte solicitate", pentru mărirea vitezei de execuție.

Exemplu:

```
{
      register int i;
      for(i = 0; i < N; ++i){
```

```
            /*… */
        }
} /* se elibereaza registrul */
```

IB.10.5. Clase de alocare a memoriei: extern

O variabilă externă este o variabilă definită în alt fişier. Declaraţia extern îi spune compilatorului că identificatorul este definit în alt fişier sursă (extern). Ea este este alocată în funcţie de modul de declarare din fişierul sursă.

Exemplu:

```
// File1.cpp
extern int i;  // Declara aceasta variabila ca fiind definita in alt fisier

// File2.cpp
int i = 88;    // Definit aici
```

IB.10.6. Alocarea dinamică a memoriei

Reamintim că pentru variabilele alocate dinamic memoria se alocă dinamic (la execuţie) în zona *heap* ataşată programului, dar numai la cererea explicită a programatorului, prin apelarea unor funcţii de bibliotecă (*malloc, calloc, realloc*). Memoria este eliberată numai la cerere, prin apelarea funcţiei *free.*

Principalele diferenţe între alocarea statică şi cea dinamică sunt:

- La alocarea statică, compilatorul alocă şi eliberează memoria automat, ocupându-se astfel de gestiunea memoriei, în timp ce la alocarea dinamică programatorul este cel care gestionează memoria, având un control deplin asupra adreselor de memorie şi a conţinutului lor.
- Entităţile alocate static sau auto sunt manipulate prin intermediul unor variabile, în timp ce cele alocate **dinamic** sunt gestionate prin intermediul **pointerilor**!

IB.10.6. 1 Funcţii standard pentru alocarea dinamică a memoriei

Funcţiile standard pentru alocarea dinamica a memoriei sunt declarate în fişierele *stdlib.h* şi *alloc.h.*

Alocarea memoriei:

void *malloc(size_t size);

Alocă memorie de dimensiunea *size* octeţi

void *calloc(int nitems, size_t size);

Alocă memorie pentru *nitems* de dimensiune *size* octeţi şi iniţializează zona alocată cu zerouri

Cele două funcţii au ca rezultat adresa zonei de memorie alocate (de tip *void.*

Dacă cererea de alocare nu poate fi satisfăcută, pentru că nu mai exista un bloc continuu de dimensiunea solicitată, atunci funcţiile de alocare au rezultat *NULL*. Funcţiile de alocare au rezultat *void** deoarece funcţia nu ştie tipul datelor ce vor fi memorate la adresa respectivă.

La apelarea funcţiilor de alocare se folosesc:

- Operatorul *sizeof* pentru a determina numărul de octeţi necesar unui tip de date (variabile);
- Operatorul de conversie *cast* pentru adaptarea adresei primite de la funcţie la tipul datelor memorate la adresa respectivă (conversie necesară atribuirii între pointeri de tipuri diferite).

Exemple:

```
//aloca memorie pentru 30 de caractere:
char * str = (char*) malloc(30);

//aloca memorie ptr. n întregi:
int * a = (int *) malloc( n * sizeof(int));

//aloca memorie ptr. n întregi si initializeaza cu zerouri
int * a= (int*) calloc (n, sizeof(int) );
```

IB.10.6. 2 Realocarea memoriei

Realocarea unui vector care creşte (sau scade) față de dimensiunea estimată anterior se poate face cu funcția *realloc*, care primeşte adresa veche şi noua dimensiune şi întoarce noua adresă:

void *realloc(void* adr, size_t size);

Funcția *realloc* realizează următoarele operații:

- Alocă o zonă de dimensiunea specificată prin al doilea parametru.
- Copiază la noua adresă datele de la adresa veche (primul parametru).
- Eliberează memoria de la adresa veche.

Exemple:

```
// dublare dimensiune curenta a zonei de la adr. a
a = (int *)realloc (a, 2*n* sizeof(int));
```

Atenție! Se va evita redimensionarea unui vector cu o valoare foarte mică de un număr mare de ori; o strategie de realocare folosită pentru vectori este dublarea capacității lor anterioare.
Exemplu de funcție cu efectul funcției *realloc*, dar doar pentru caractere:

```
char * ralloc (char * p, int size) {     // p = adresa veche
  char *q;                               // q=adresa noua
  if (size==0) {                         // echivalent cu free
    free(p);
    return NULL;
  }
  q = (char*) malloc(size);              // aloca memorie
  if (q) {                               // daca alocare reusita
       memcpy(q,p,size);                 // copiere date de la p la q
       free(p);                          // elibereaza adresa p
  }
  return q;                              // q poate fi NULL
}
```

Observație: La mărirea blocului, conținutul zonei alocate în plus nu este precizat, iar la micşorarea blocului se pierd datele din zona la care se renunță.

IB.10.6. 3 Eliberarea memoriei

Funcția *free* are ca argument o adresă (un pointer) şi eliberează zona de la adresa respectivă (alocată dinamic). Dimensiunea zonei nu mai trebuie specificată deoarece este memorată la începutul zonei alocate (de către funcția de alocare):

void free(void* adr);

Eliberarea memoriei prin *free* este inutilă la terminarea unui program, deoarece înainte de încărcarea şi lansarea în execuție a unui nou program se eliberează automat toată memoria *heap*.

Exemple:

```
char *str;
```

```
str=(char *)malloc(10*sizeof(char));
…
str=(char *)realloc(str,20*sizeof(char));
…
free(str);
```

Observaţie:
Atenție la definirea de şiruri în mod dinamic! Şirul respectiv trebuie inițializat cu adresa unui alt şir sau a unui spațiu alocat pe heap (adică alocat dinamic)!

Exemple:

```
char *sir3;
char şir1[30];

// Varianta 1: sir3 ia adresa unui şir static
sir3 = sir1;          // Echivalent cu: sir3=&sir1; ⇔ sir3=&sir1[0];
char *sir4="test";  //sir4 este iniţializat cu adresa unui şir constant

// Varianta 2: se alocă dinamic un spațiu pe heap
sir3=(char *)malloc(100*sizeof(char));
```

Exemplu
Program care alocă spațiu pentru o variabilă întreagă dinamică, după citire şi tipărire, spațiul fiind eliberat. Modificați programul astfel încât variabila dinamică să fie de tip double.

Rezolvare

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

int main(){

      int *pi;
      pi=(int *)malloc(sizeof(int));
      if(pi==NULL){
            puts("*** Memorie insuficienta ***");
            return 1;       // revenire din main
      }

      printf("valoare:");
      //citirea variabilei dinamice, de pe heap, de la adresa din pi!!!
      scanf("%d",pi);
      *pi=*pi*2;        // dublarea valorii
      printf("val=%d,pi(adresa pe heap)=%p,adr_pi=%p\n", *pi, pi, &pi);

      // sizeof aplicat unor expresii:
      printf("%d %d %d\n",sizeof(*pi), sizeof(pi), sizeof(&pi));

      free(pi);         //eliberare spatiu
      printf("pi(dupa elib):%p\n",pi); // nemodificat, dar invalid!
      return 0;
}
```

IB.10.7. Vectori alocați dinamic

Structura de vector are avantajul simplității şi economiei de memorie față de alte structuri de date folosite pentru memorarea unei colecții de date.

Dezavantajul unui vector cu dimensiune fixă (stabilită la declararea vectorului şi care nu mai poate fi modificată la execuţie) apare în aplicaţiile cu vectori de dimensiuni foarte variabile, în care este dificil de estimat o dimensiune maximă, fără a face risipă de memorie.
De cele mai multe ori programele pot afla din datele citite dimensiunile vectorilor cu care lucrează şi deci pot face o alocare dinamică a memoriei pentru aceşti vectori. Aceasta este o soluţie mai flexibilă, care foloseşte mai bine memoria disponibilă şi nu impune limitări arbitrare asupra utilizării unor programe.

În limbajul C nu există practic nici o diferenţă între utilizarea unui vector cu dimensiune fixă şi utilizarea unui vector alocat dinamic, ceea ce încurajează si mai mult utilizarea unor vectori cu dimensiune variabilă.

Un vector alocat dinamic se declară ca variabilă pointer care se iniţializează cu rezultatul funcţiei de alocare. Tipul variabilei pointer este determinat de tipul componentelor vectorului.

Exemplu:

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

int main() {
  int n, i;
  int * a;
     // adresa vector alocat dinamic

  printf ("n=");
  scanf ("%d", &n);                      // dimensiune vector
  a=(int *) calloc (n,sizeof(int));      // aloca memorie pentru vector
        // sau: a=(int*) malloc (n*sizeof(int));

  // citire component vector:
  printf ("componente vector: \n");
  for (i=0;i<n;i++)
    scanf ("%d", &a[i]);                 // sau   scanf ("%d", a+i);
  // afisare vector:
  for (i=0;i<n;i++)
    printf ("%d ",a[i]);
  return 0;
}
```

Există şi cazuri în care datele memorate într-un vector rezultă din anumite prelucrări, iar numărul lor nu poate fi cunoscut de la începutul execuţiei. În acest caz se poate recurge la o realocare dinamică a memoriei. O strategie de realocare pentru vectori este dublarea capacităţii lor anterioare.

În exemplul următor se citeşte un număr necunoscut de valori întregi într-un vector extensibil:
Program care citeşte numere reale până la CTRL+Z, le memorează într-un vector alocat şi realocat dinamic în funcţie de necesităţi şi le afişează.

Rezolvare:

```
#include <stdio.h>
#include <stdlib.h>
#define INCR 4

int main() {
      int n,n_crt,i ;
      float x, * v;
      n = INCR;              // dimensiune memorie alocata
      n_crt = 0;             // numar curent elemente în vector
```

```
      v = (float *)malloc (n*sizeof(float));     //alocare initiala

      while (scanf("%f",&x) !=EOF){
            if (n_crt == n) {
            n = n + INCR;
            v = (float *) realloc (v, n*sizeof(float) ); //realocare
            }
            v[n_crt++] = x;
      }
      for (i=0; i<n_crt; i++)
            printf ("%.2f ", v[i]);
      return 0;
}
```

Din exemplele anterioare lipseşte eliberarea memoriei alocate pentru vectori, dar fiind vorba de un singur vector alocat în funcția *main* şi necesar pe toată durata de execuție, o eliberare finală este inutilă. Eliberarea explicită poate fi necesară pentru vectori de lucru, alocați dinamic în funcții.

IB.10.8. Matrice alocate dinamic

Alocarea dinamică pentru o matrice este importantă deoarece foloseşte economic memoria şi permite matrice cu linii de lungimi diferite. De asemenea reprezintă o soluție bună la problema parametrilor de funcții de tip matrice.

O matrice alocată dinamic este de fapt un *vector de pointeri către fiecare linie din matrice*, deci un *vector de pointeri la vectori alocați dinamic*. Dacă numărul de linii este cunoscut sau poate fi estimată valoarea lui maximă, atunci vectorul de pointeri are o dimensiune constantă. O astfel de matrice se poate folosi la fel ca o matrice declarată cu dimensiuni constante.
Exemplu de declarare matrice de întregi:

```
int * a[M];            // M este o constanta simbolica
```

Dacă nu se poate estima numărul de linii din matrice atunci şi vectorul de pointeri se alocă dinamic, iar declararea matricei se face ca pointer la pointer:

```
int** a;
```

În acest caz se va aloca mai întâi memorie pentru un vector de pointeri (funcție de numărul liniilor) şi apoi se va aloca memorie pentru fiecare linie cu memorarea adreselor liniilor în vectorul de pointeri.
Notația *a[i][j]* este interpretată astfel pentru o matrice alocată dinamic:

- *a[i]* conține un pointer (o adresă *b*)
- *b[j]* sau *b+j* conține întregul din poziția *j* a vectorului cu adresa *b*.

Exemplu
Să se scrie funcții de alocare a memoriei şi afişare a elementelor unei matrice de întregi alocată dinamic.

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

// rezultat adresa matrice sau NULL
int ** intmat ( int nl, int nc) {
      int i;
      int ** p=(int **) malloc (nl*sizeof (int*));
      if ( p != NULL)
      for (i=0; i<nl ;i++)
            p[i] =(int*) calloc (nc,sizeof (int));
```

```
      return p;
}

void printmat (int ** a, int nl, int nc) {
      int i,j;

      for (i=0;i<nl;i++) {
            for (j=0;j<nc;j++)
                  printf (“%2d”, a[i][j] );
            printf(“\n”);
      }
}


int  main () {
      int  **a, nl, nc, i, j;

      printf ("nr linii şi nr coloane: \n");
      scanf ("%d%d", &nl, &nc);
      a= intmat(nl,nc);

      for (i=0;i<nl;i++)
            for (j=0;j<nc;j++)
                  a[i][j]= nc*i+j+1;

      printmat (a ,nl,nc);
      return 0;
}
```

Funcția *printmat* dată anterior nu poate fi folosită pentru afișarea unei matrice cu dimensiuni constante. Explicația este interpretarea diferită a conținutului zonei de la adresa aflată în primul argument.
Astfel, chiar dacă exemplul următor este corect sintactic el nu se execută corect:

```
int x [2][2]={{1,2},{3,4}};        // 2 linii şi 2 coloane
printmat ( (int**)x, 2, 2);
```

IB.10.9. Funcții cu rezultat vector

O funcție **nu** poate avea ca rezultat un vector sub forma:

int [] funcție(…) {…}

O funcție poate avea ca rezultat doar un pointer !!

int *funcție(…) {…}

De obicei, rezultatul pointer este egal cu unul din argumente, eventual modificat în funcție.
Exemplu corect:

```
// incrementare pointer p

char * incptr ( char * p) {

   return ++p;

}
```

Atenție! Acest pointer nu trebuie să conțină adresa unei variabile locale, deoarece:

- O variabilă locală are o existență temporară, garantată numai pe durata executării funcției în care este definită (cu excepția variabilelor locale statice)
- Adresa unei astfel de variabile nu trebuie transmisă în afara funcției, pentru a fi folosită ulterior!!

Exemplu greşit:

```
// vector cu cifrele unui nr intreg de maxim cinci cifre
int * cifre (int n) {
  int k, c[5];          // vector local
  for (k=4;k>=0;k--) {
    c[k]=n%10; n=n/10;
  }
  return c;             // aici este eroarea !
}
//warning la compilare şi POSIBIL rezultate greşite în main!!
```

O funcție care trebuie să transmită ca rezultat un vector poate fi scrisă corect în în mai multe feluri:

1. **Primeşte ca parametru adresa vectorului (*definit şi alocat în altă funcție*)** şi depune rezultatele la adresa primită (este soluția recomandată!!)

```
void cifre (int n, int c[ ]) {
  int k;
  for (k=4;k>=0;k--) {
    c[k]=n%10; n=n/10;
  }
}
int main(){
      int a[10];
      ….
      cifre(n,a);
      ….
}
```

2. **Alocă dinamic memoria** pentru vector (cu "malloc")
   - Această alocare (pe heap) se menține şi la ieşirea din funcție.
   - Funcția are ca rezultat adresa vectorului alocat în cadrul funcției.
   - Problema este unde şi când se eliberează memoria alocată.

```
int * cifre (int n) {
  int k, *c;            // vector local
  c = (int*) malloc (5*sizeof(int));
  for (k=4;k>=0;k--) {
    c[k]=n%10; n=n/10;
  }
  return c;             // corect
}
```

3. O soluție oarecum echivalentă este **utilizarea unui vector local static**, care continuă să existe şi după terminarea funcției.

IB.10.10. Vectori de pointeri la date alocate dinamic

Ideea folosită la matrice alocate dinamic este aplicabilă şi pentru alte date alocate dinamic: adresele acestor date sunt reunite într-un vector de pointeri. Situațiile cele mai frecvente sunt:

- vectori de pointeri la şiruri de caractere alocate dinamic
- vectori de pointeri la structuri alocate dinamic.

Exemplu de utilizare a unui vector de pointeri la structuri alocate dinamic:

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

typedef  struct {
   int zi, luna, an;
} date;

// afisare date reunite în vector de pointeri
void print_vp ( date * vp[], int n) {
   int i;
   for(i=0;i<n;i++)
      printf ("%4d %4d %4d \n", vp[i]->zi, vp[i]->luna, vp[i]->an);
   printf ("\n");
}

int main () {
   date d, *dp;
   date *vp[100];
   int n=0;

   while (scanf ("%d%d%d", &d.zi, &d.luna, &d.an) {
      dp=(date*)malloc (sizeof(date));    //    alocare    dinamica    ptr
      structură
      *dp=d;
             // copiaza datele citite la dp
      vp[n++]=dp;
       // memoreaza adresa in vector
   }
   print_vp (vp,n);
}
```

De reținut că trebuie create adrese distincte pentru fiecare variabilă structură şi că ar fi greşit să punem adresa variabilei ***d*** în toate pozițiile din vector!

Este posibilă şi varianta următoare pentru ciclul principal din *main* dacă cunoaştem numărul de elemente din structură:

```
....
scanf (“%d”,&n);                    // numar de structuri ce vor fi citite
for (k=0; k<n; k++) {
   dp = (date*) malloc (sizeof(date));// alocare dinamica ptr structură
   scanf ("%d%d%d", &dp->zi, &dp->luna, &dp->an)
   vp[n++]=dp;                      // memoreaza adresa in vector
}
....
```

Exemplu
Program pentru citirea unor nume, alocare dinamică a memoriei pentru fiecare şir (în funcție de lungimea şirului citit) şi memorarea adreselor şirurilor într-un vector de pointeri. În final se vor afişa numele citite, pe baza vectorului de pointeri.

Să se adauge programului anterior o funcție de ordonare a vectorului de pointeri la şiruri, pe baza conținutului fiecărui şir. Programul va afişa lista de nume în ordine alfabetică.
a. Vectorului de pointeri i se va aloca o dimenisune fixă.
b. Vectorul de pointeri se va aloca dinamic, funcție de numărul de şiruri.

Rezolvare a:

```
void printstr ( char * vp[], int n) { //afisare
      int i;
      for(i=0;i<n;i++)
            printf ("%s\n",vp[i]);
}

int readstr (char * vp[]) {   // citire siruri şi creare vector de pointeri
      int n=0; char * p, sir[80];
      while ( scanf ("%s", sir) == 1) {
            vp[n]= (char*) malloc (strlen(sir)+1);
            strcpy( vp[n],sir);
            //sau:  vp[n]=strdup(sir);
            ++n;
      }
      return n;
}

/* ordonare vector de pointeri la şiruri prin Bubble Sort (metoda
bulelor)*/
void sort ( char * vp[],int n) {
      int i,j,schimb=1;
      char * tmp;

      while(schimb){
            schimb=0;
            for (i=0;i<n-1;i++)
                  if ( strcmp (vp[i],vp[i+1])>0) {
                        tmp = vp[i];
                        vp[i] = vp[i+1];
                        vp[i+1] = tmp;
                        schimb = 1;
                  }
      }
}


int main () {
      int n;
      char * vp[1000];   // vector de pointeri, cu dimens. fixa

      n=readstr(vp);    // citire siruri şi creare vector
      sort(vp,n);       // ordonare vector
      printstr(vp,n);   // afişare şiruri
      return 0;
}
```

IB.10.11. Anexa A: Structuri alocate dinamic

În cazul variabilelor structură alocate dinamic şi care nu au nume se va face **o indirectare printr-un pointer** pentru a ajunge la variabila structură.

Avem de ales între următoarele două notaţii echivalente:
pt->camp (*pt).camp
unde *pt* este o variabilă care conţine un pointer la o structură cu câmpul **camp**.

O colecţie de variabile structură alocate dinamic se poate memora în două moduri:

- Ca un vector de pointeri la variabilele structură alocate dinamic;
- Ca o listă înlăntuită de variabile structură, în care fiecare element al listei conţine şi un câmp de legătură către elementul următor (ca pointer la structură).

Pentru primul mod de memorare a se vedea ***Vectori de pointeri la date alocate dinamic.***

Liste înlănţuite

O listă înlănţuită ("linked list") este o colecţie de variabile alocate dinamic (de acelaşi tip), dispersate în memorie, dar legate între ele prin pointeri, ca într-un lanţ. Într-o listă liniară simplu înlănţuită fiecare element al listei conţine adresa elementului următor din listă. Ultimul element poate conţine ca adresă de legatură fie constanta NULL, fie adresa primului element din listă (lista circulară).

Adresa primului element din listă este memorată într-o variabilă cu nume şi numită cap de lista ("list head"). Pentru o listă vidă variabila cap de listă este NULL.

Structura de listă este recomandată atunci când colecţia de elemente are un conţinut foarte variabil (pe parcursul execuţiei) sau când trebuie păstrate mai multe liste cu conţinut foarte variabil.

Un element din listă (un nod de listă) este de un tip structură şi are (cel puţin) două câmpuri:

- un câmp de date (sau mai multe)
- un câmp de legătură

Definiţia unui nod de listă este o definitie recursivă, deoarece în definirea câmpului de legătură se foloseşte tipul în curs de definire.

Exemplu pentru o listă de întregi:

```
typedef struct snod {
     int val ;                //  camp de date
     struct snod * leg ;      // camp de legatura
} nod;
```

Programul următor arată cum se poate crea şi afisa o listă cu adăugare la început (o stivă realizată ca listă înlănţuită):

```
int main ( ) {
```

```
  nod *lst=NULL, *nou, * p;             // lst = adresa cap de lista
  int x;
  // creare lista cu numere citite
  while (scanf("%d",&x) > 0) {     // citire numar intreg x
    nou=(nod*)malloc(sizeof(nod));// creare nod nou
    nou->val=x;                    // completare camp de date din nod
    nou->leg=lst; lst=nou;         // legare nod nou la lista
  }
  // afisare listă (fara modificare cap de lista)
  p=lst;
  while ( p != NULL) {             // cat mai sunt noduri
    printf("%d  ", p□ val);        // afisare numar de la adr p
    p=p->leg;                      // avans la nodul urmator
  }
}
```

Câmpul de date poate fi la rândul lui o structură specifică aplicației sau poate fi un pointer la date alocate dinamic (un şir de caractere, de exemplu).
De obicei se definesc funcții pentru operațiile uzuale cu liste.
Exemple:

```
typedef struct nod {                    // un nod de lista inlantuita
     int val;                           // date din fiecare nod
     struct snod *leg;                  // legatura la nodul urmator
} nod;

// insertie la inceput lista
nod* insL( nod* lst, int x) {
     nod* nou ;                         // adresa nod nou
     if ((nou=(nod*)malloc(sizeof(nod)))==NULL)
     return NULL;
     nou->val=x;
     nou->leg=lst;
     return nou;                        // lista incepe cu nou
}

// afisare continut lista
void printL ( nod* lst) {
     while (lst != NULL) {
       printf("%d ",lst->val);          // scrie informatiile din nod
       lst=lst->leg;                    // si se trece la nodul următor
     }
}

int main () {                           // creare si afisare lista stiva
     nod* lst; int x;
     lst=NULL;                          // initial lista e vida
     while (scanf("%d",&x) > 0)
       lst=insL(lst,x);                 // introduce pe x in lista lst
     printL (lst);                      // afisare lista
}
```

Alte structuri dinamice folosesc câte doi pointeri sau chiar un vector de pointeri; într-un arbore binar fiecare nod conține adresa succesorului la stânga şi adresa succesorului la dreapta, într-un arbore multicăi fiecare nod conține un vector de pointeri către succesorii acelui nod.

IB.10.11. Anexa B: Operatori pentru alocare dinamică in C++

În C++ s-au introdus doi operatori noi:

- pentru alocarea dinamică a memoriei ***new***
- pentru eliberarea memoriei dinamice ***delete***

destinați să înlocuiască funcțiile de alocare şi eliberare.
Operatorul ***new*** are ca operand un nume de tip, urmat în general de o valoare inițială pentru variabila creată (între paranteze rotunde); rezultatul lui *new* este o adresă (un pointer de tipul specificat) sau *NULL* daca nu există suficientă memorie liberă.
Exemple:

```
nod * pnod;              // pointer la nod de lista
pnod = new nod;          // alocare fara inițializare
int * p = new int(3);    // alocare cu inițializare
```

Operatorul *new* are o formă puțin modificată la alocarea de memorie pentru vectori, pentru a specifica numărul de componente.
Exemplu:

```
int * v = new int [n];        // vector de n intregi
```

Operatorul ***delete*** are ca operand o variabilă pointer şi are ca efect eliberarea blocului de memorie adresat de pointer, a cărui mărime rezultă din tipul variabilei pointer sau este indicată explicit.
Exemple:

```
int * v;
delete v;          // elibereaza sizeof(int) octeți
delete [ ] v;
delete [n] v;      // elibereaza n*sizeof(int) octeți
```

Exemplu de utilizare new şi delete pentru un vector de întregi alocat dinamic:

```
#include <iostream>
#include <cstdlib>

int main() {
   const int SIZE = 5;
   int *pArray;

   pArray = new int[SIZE];  // alocare memorie

   // atribuie numere aleatoare intre 0 and 99
   for (int i = 0; i < SIZE; i++) {
      *(pArray + i) = rand() % 100;
   }

   // afisare
   for (int i = 0; i < SIZE; i++) {
      cout << *(pArray + i) << " ";
   }
   cout << endl;

   delete[] pArray;  // eliberare memorie
   return 0;
}
```

După alocarea de memorie cu *new* se pot folosi funcțiile *realloc* şi *free* pentru realocare sau eliberare de memorie.