Structuri cu alocare dinamică în C: Stive, Cozi, etc.

Lecție deschisă

Drăgulici Dumitru Daniel

Facultatea de matematică și informatică,
Universitatea București

septembrie 2011

Cuprins

Cuprins

Liste, stive, cozi

Conceptual, o LISTA este o multime finita, eventual vida, total ordonata, de ELEMENTE oarecare.

De obicei lista se reprezinta ca un sir finit: x1, ..., xn sau x0, ..., x(n-1) si astfel putem vorbi de primul/ultimul element, pozitia unui element, predecesorul/succesorul unui element, etc.

Asupra unei liste oarecare se pot efectua o gama larga de operatii: cautarea unui element, inserarea/extragerea unui element pe o pozitie data sau inainte/dupa un element dat, etc.

Liste, stive, cozi

Impunand restrictii asupra operatiilor permise asupra listei, obtinem clase particulare de liste, cum ar fi:

- STIVA: operatiile de inserare/extragere a unui element sunt permise doar la unul dintre capete, numit VARFUL stivei;

  Astfel, stivele se mai numesc liste LIFO (last in first out), deoarece elementul inserat ultima data este primul care poate fi extras.

- COADA: operatia de inserare a unui element este permisa doar la unul dintre capete, numit BAZA cozii, iar cea de extragere a unui element doar la celalalt capat, numit VARFULl cozii.

  Astfel, cozile se mai numesc liste FIFO (first in first out), deoarece elementul care a fost inserat prima data este primul care poate fi extras.

Cuprins

Modalități de alocare

Reprezentarea unei liste intr-un sistem informatic (implementarea) presupune specificarea:

- Unei modalitati de stocare a elementelor in memorie;

Memoria unui sistem informatic este considerata de obicei ca fiind un sir finit de locatii elementare (octeti, bytes), avand fiecare aceeasi dimensiune si o adresa unica; adresele octetilor pot fi asimilate cu numerele naturale succesive 0, 1, ... o valoare maxima. O locatie oarecare de memorie este formata dintr-un numar > 0 de octeti succesivi; ea are o dimensiune (numarul octetilor componenti) si o adresa (adresa primului sau octet).

- Unui set de mecanisme primitive pentru a opera asupra acestor elemente.

Deasupra acestor primitive se definesc operatiile permise asupra tipului de lista respectiv (inserare element, extragere element, test de lista vida, etc.).

Modalități de alocare

Din punct de vedere al metodei de implementare, distingem doua modalitati de alocare a listelor:

- alocarea secventiala;

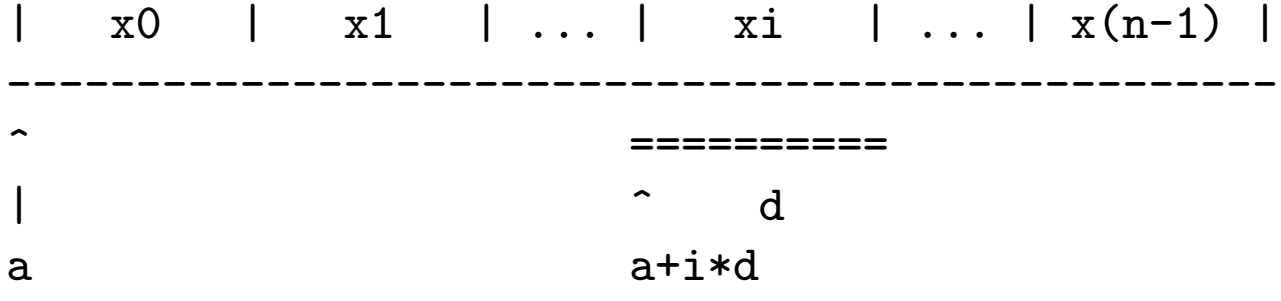
- alocarea inlantuita.

Modalități de alocare

ALOCAREA SECVENTIALA consta in urmatoarele:

- elementele listei sunt stocate in locatii de memorie de aceeasi dimensiune, adiacente, in ordinea in care apar in lista, iar aceste locatii contin doar valorile elementelor si nimic in plus;
- se retine (memoreaza) adresa primului element "a", dimensiunea locatiei unui element "d" si numarul elementelor "n".

Accesarea elementului de pe pozitia i (numarand de la 0) se face calculand adresa locatiei acestuia, dupa formula: a + i * d (trebuie implementat acest mecanism):



Modalități de alocare

Avantajele/dezavantajele alocarii secventiale:

- accesarea unui element se face in timp constant, indiferent de pozitia elementului (deoarece calculul a+i*d se face in timp constant, indiferent de valoarea lui i);
- inserarea/extragerea unui element se face in timp liniar, in functie de pozitia elementului (deoarece trebuie translatate celelalte elemente).

Exemplu: dorim inserarea lui x pe pozitia 2

```
                    v
     |   x0   |   x1   |   x2   |   x3   |
     ------------------------------------
                      ================>
                      elementele x2, x3 trebuie translatate
                      spre dreapta
rezulta:
     |   x0   |   x1   |   x    |   x2   |   x3   |
     ----------------------------------------------
```

Modalități de alocare

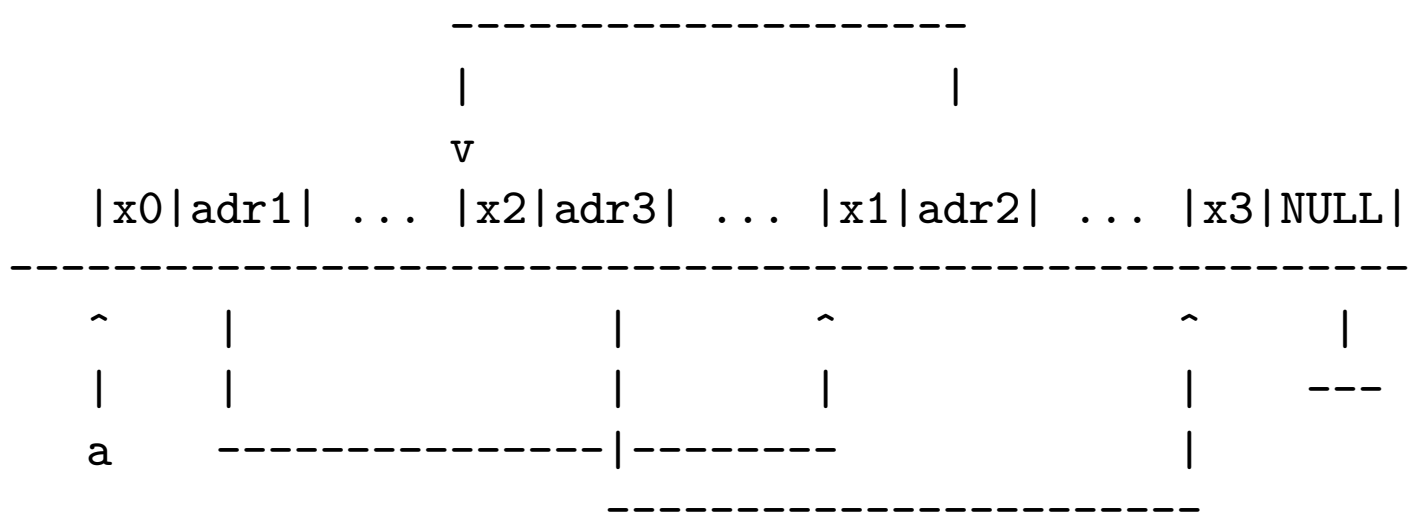
ALOCAREA (SIMPLU) INLANTUITA consta in urmatoarele:

- elementele listei sunt stocate in locatii de memorie nu neaparat de aceeasi dimensiune, nu neaparat adiacente, nu neaparat in ordinea in care apar in lista, iar fiecare locatie contine pe langa valoarea elementului si adresa locatiei urmatorului element din lista (poate fi gandit ca predecesorul sau succesorul acestui element); in cazul ultimului element, se retine o adresa invalida (NULL) pentru a marca faptul ca nu mai exista un urmator;
- se retine (memoreaza) adresa primului element "a".

Alternativ, nu mai retinem NULL la ultimul element, dar retinem numarul elementelor "n".

Modalități de alocare

Accesarea elementului de pe pozitia i (numarand de la 0) se face plecand de la locatia aflata la adresa "a" si trecand de "i" ori la "urmatorul" (trebuie implementat acest mecanism):



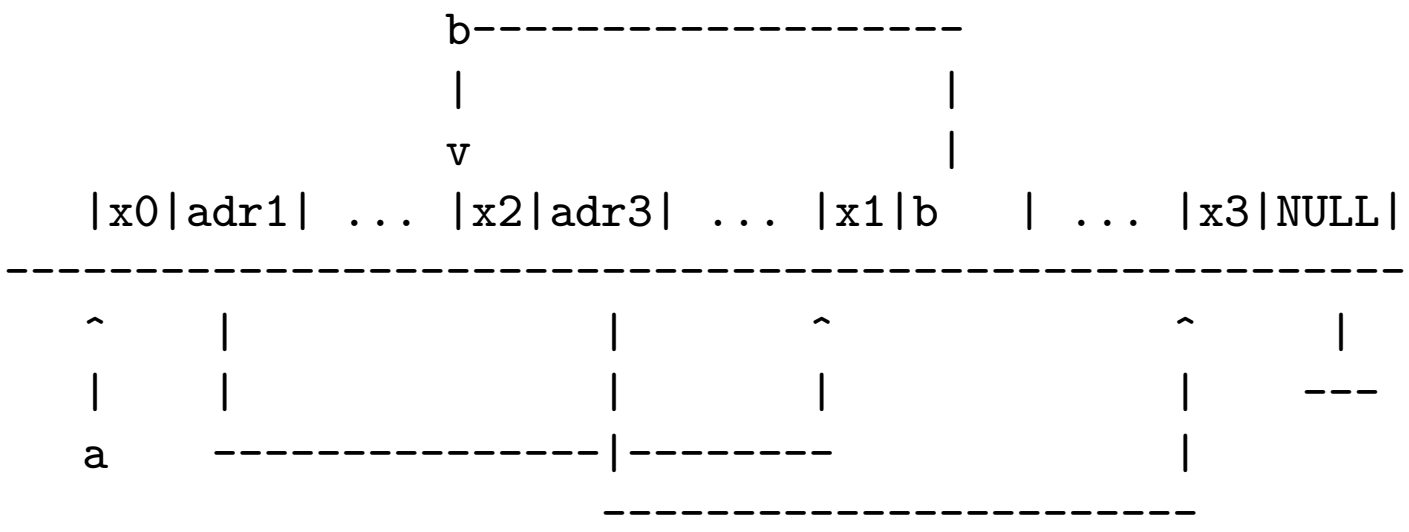
Modalități de alocare

Avantajele/dezavantajele alocarii inlantuite:

- accesarea unui element se face in timp liniar in functie de pozitia "i" a elementului (deoarece nu putem determina locatia lui decat plecand de la adresa de inceput "a" si trecand de "i" ori la "urmatorul");
- se consuma mai multa memorie, deoarece locatia fiecarui element contine pe langa valoarea elementului si adresa urmatorului);
- inserarea/extragerea unui element se face in timp constant, odata determinata pozitia sa (de exemplu locatia elementului dupa care se insereaza ca element urmator):

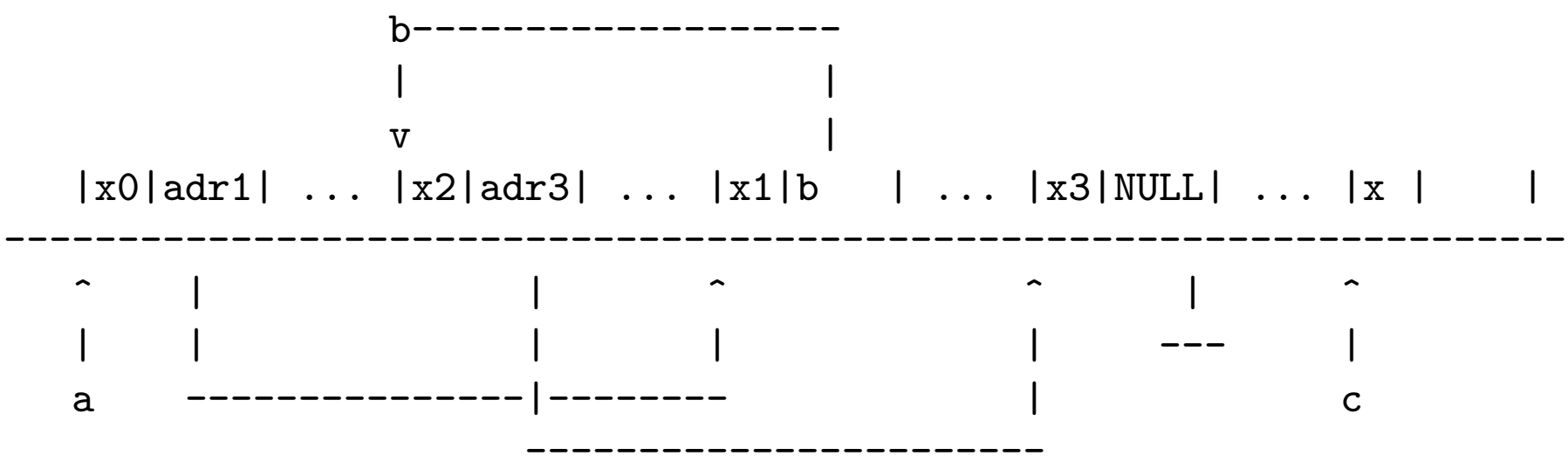
Modalități de alocare

Exemplu: dorim inserarea lui x ca urmator lui x2;
presupunem ca am determinat adresa "b" a lui x2



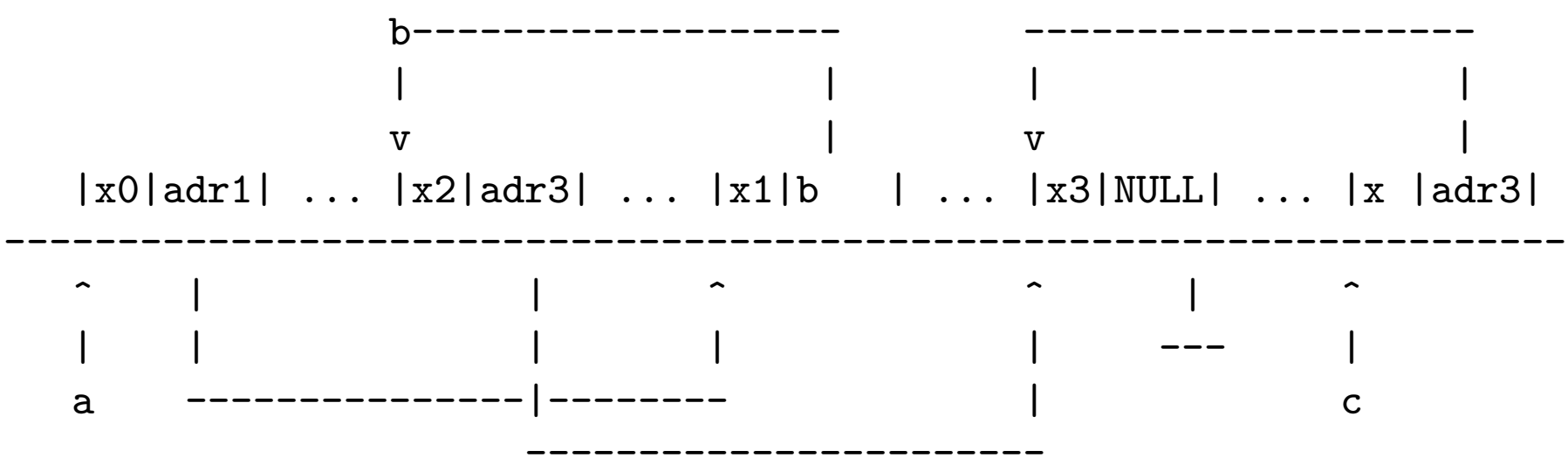
Modalități de alocare

pasul 1: alocam o noua locatie de element in zona libera a memoriei, de ex. la o adresa "c", in care stocam valoarea x:



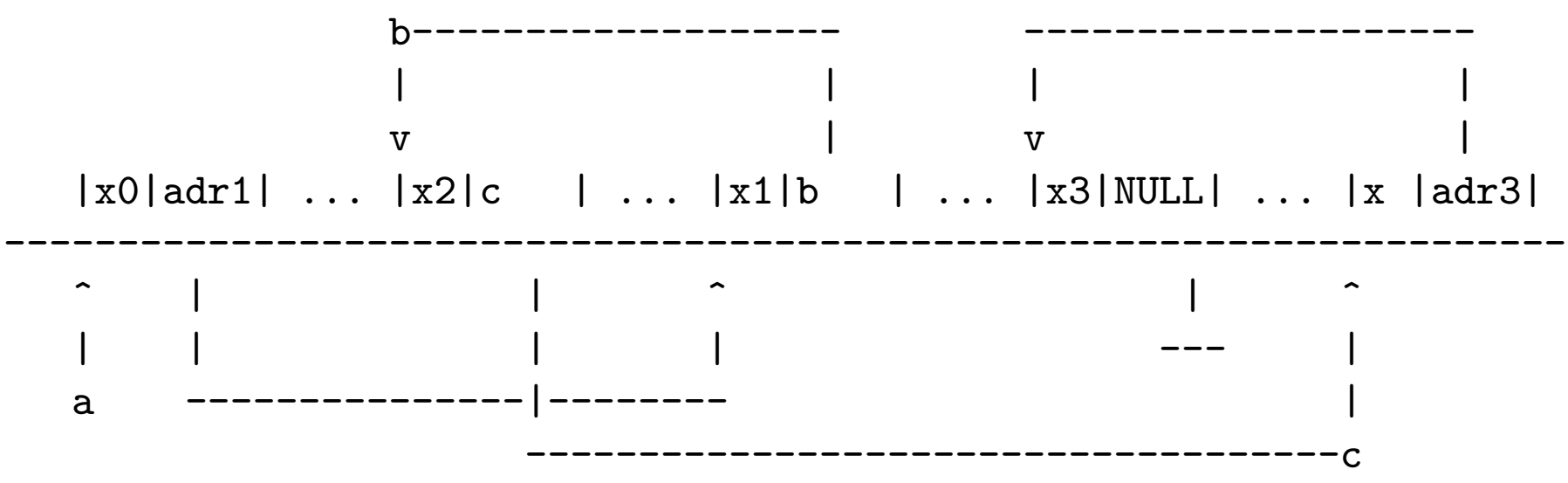
Modalități de alocare

pasul 2: in partea de "adresa a urmatorului" din noua locatie stocam informatia corespunzatoare din locatia de la adresa "b":



Modalități de alocare

pasul 3: in partea de "adresa a urmatorului" din locatia de la adresa "b" stocam "c"



Modalități de alocare

In cazul alocarii simplu inlantuite a stivelor si cozilor, cel mai comod si eficient este sa implementam:

- stivele a.i. legaturile sa mearga de la varf (unde inseram/extragem) spre celalalt capat al listei; intr-o variabila separata se retine adresa varfului;
- cozile a.i. legaturile sa mearga de la varf (de unde extragem) spre baza (unde inseram); intr-o variabila separata se retine adresa varfului, dar este eficient sa mai folosim inca o variabila, pentru a retine si adresa bazei.

Modalități de alocare

Variante de alocare inlantuita:

- alocarea dublu inlantuita (lista dublu inlantuita): pentru fiecare element se retine atat adresa elementului urmator, cat si a celui anterior; astfel, se consuma mai multa memorie, dar este posibila parcurgerea listei in ambele sensuri;

- alocarea circulara (lista circulara): la ultimul element se retine in loc de NULL adresa primului element; astfel, lista se poate parcurge circular;
  se pot aloca liste dublu inlantuite circulare - ele pot fi parcurse circular in ambele sensuri.

In cazul alocarii dublu inlantuite si/sau circulare, in loc sa retinem adresa primului element putem retine doar adresa "elementului curent".

In cele ce urmeaza vom considera doar liste simplu inlantuite ne-circulare.

Modalități de alocare

Odata ales tipul de lista (oarecare, stiva, coada) si modul de alocare (secventiala, inlantuita), se definesc operatiile permise asupra tipului respectiv de lista (inserare element la baza, extragere element din varf, testare lista vida, etc.) deasupra mecanismelor primitive de operare specifice modului de alocare (alocare/dezalocare memorie, calculul direct de adresa, respectiv trecerea repetata la urmatorul).

Modalități de alocare

Pentru implementarea listelor in limbajul C sunt folosite de obicei urmatoarele instrumente:

Pentru stocarea elementelor:
- masive: vectori, matrici, etc.
- structuri si pointeri;

Pentru accesarea elementelor:
- operatii cu adrese: referentiere/deferentiere, aritmetica pointerilor, indexare;
- accesarea membrilor unei structuri;

Pentru alocarea dinamica (i.e. la cerere, in timpul rularii) de memorie:
- functiile de biblioteca malloc, calloc, realloc, free.

Reamintim pe scurt aceste instrumente cu ajutorul unor exemple (recapitulare limbajul C):

Cuprins

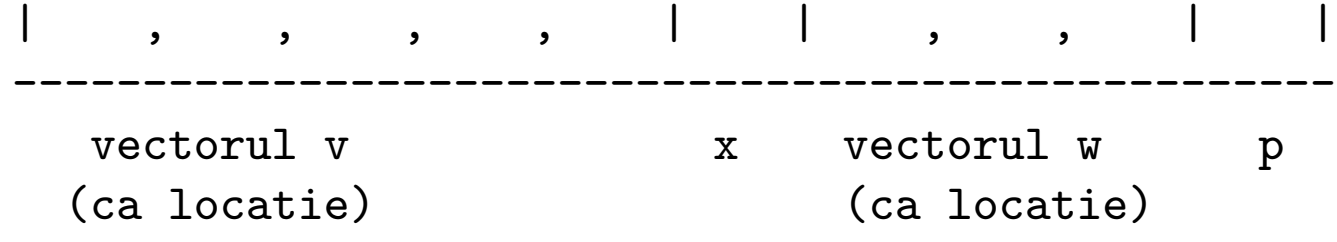
Recapitulare C - vectori

Un VECTOR este un ansamblu de date de un acelasi tip (numite elemente sau COMPONENTE), stocate in locatii de aceeasi dimensiune, adiacente. Vom numi vector si locatia de memorie ce contine aceste date.

Exemplu:

```
int v[5], x, w[3], *p;
```

Am definit vectorii "v" si "w", cu 5, resp. 3 componente intregi, variabila intreaga "x" si variabila pointer la intreg "p":



Recapitulare C - vectori

Numele unui vector este o constanta pointer, desemnand adresa sa de inceput, privita ca adresa de componenta (aceast lucru este important pentru aritmetica pointerilor); in particular nu este lvalue (nu poate aparea in stanga unei atribuiri); i se poate aplica insa "&", si va desemna aceeasi adresa, dar ca adresa de vector cu numarul declarat de componente.

In limbajul C, unui operand adresa (indiferent daca este valoarea constanta specificata de numele unui vector sau valoarea unei variabile pointer) ii putem aplica operatiile "*" (deferentiere), "+intreg", "-intreg" (aritmetica pointerilor), "[intreg]" (indexare), cu efectele cunoscute de la pointeri (a se vedea cursul respectiv - nu mai reamintim). In particular, aplicand "[numar]" unui nume de vector specificam a "numar+1" - a componenta a sa, ca variabila (lvalue).

Recapitulare C - vectori

```
Exemplu (presupunem ca sizeof(int) este 4):

int v[5], x, w[3], *p;

  |    ,    ,    ,    ,    |    |    ,    ,    |  a |
  ---------------------------------------------------
  ^    ^                     ^ x  ^                   p
  |    |                     |    |
  |    v+1                   |    w (adresa de "int")
  |    adresa fizica a+4   (&v)+1
  |                        adresa fizica a+5*4
  |
  v  (adresa de "int")
  &v (adresa de vector de 5 "int")
  adresa fizica a

v=p; /* gresit, "v" nu este lvalue */
p=v; /* corect, "p" primeste valoarea "a" */
```

Recapitulare C - vectori

```
int v[5], x, w[3], *p;

  | 10 ,    ,    ,    ,    |    |    ,    ,    |  a |
  ---------------------------------------------------
  ^    ^                    ^ x  ^                   p
  |    |                    |    |
  |    v+1                  |    w (adresa de "int")
  |    adresa fizica a+4  (&v)+1
  |                       adresa fizica a+5*4
  |
  v  (adresa de "int")
  &v (adresa de vector de 5 "int")
  adresa fizica a

In continuare sunt echivalente si dau valoarea 10 primei
   componente a lui "v":

*v=10; v[0]=10; *p=10; p[0]=10;
```

Recapitulare C - vectori

```
int v[5], x, w[3], *p;

  | 10 ,    ,    ,    ,    |    |    ,    ,    |  a |
  ---------------------------------------------------
  ^    ^                     ^ x  ^                   p
  |    |                     |    |
  |    v+1                   |    w (adresa de "int")
  |    adresa fizica a+4  (&v)+1
  |                       adresa fizica a+5*4
  |
  v  (adresa de "int")
  &v (adresa de vector de 5 "int")
  adresa fizica a

In continuare sunt echivalente si desemneaza adresa celei
   de-a doua componente a lui "v":

v+1;  &v[1];  p+1;  &p[1];
```

Recapitulare C - vectori

```
int v[5], x, w[3], *p;

  | 10 , 20 ,    ,    ,    |    |    ,    ,    |  a |
  ---------------------------------------------------
  ^    ^                     ^ x  ^                   p
  |    |                     |    |
  |    v+1                   |    w (adresa de "int")
  |    adresa fizica a+4  (&v)+1
  |                       adresa fizica a+5*4
  |
  v  (adresa de "int")
  &v (adresa de vector de 5 "int")
  adresa fizica a
```

In continuare sunt echivalente si dau valoarea 20 celei de-a doua componente a lui "v":

```
*(v+1)=20; v[1]=20; *(p+1)=20; p[1]=20;
```

Recapitulare C - vectori

Observam ca locatiile alocate componentelor unui vector declarat:

```
tip nv[nr];
```

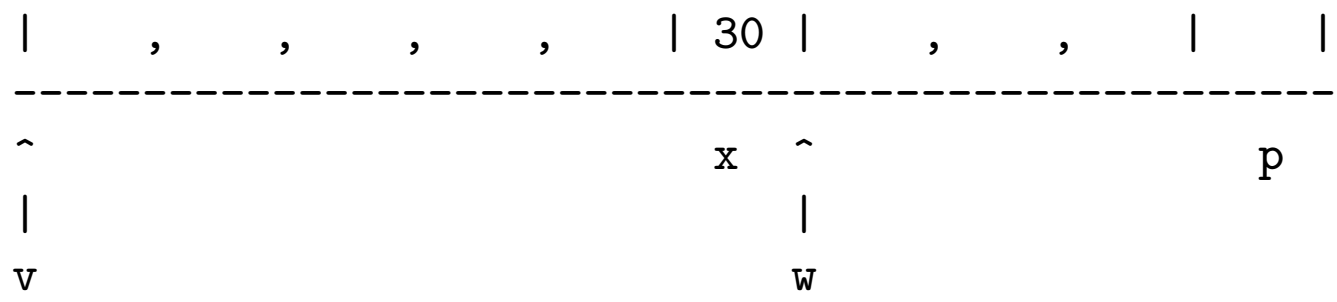
corespund indicilor 0, ..., nr-1. Daca aplicam indici in afara acestui interval, accesam locatii invecinate (a caror adresa rezulta din aritmetica pointerilor), cauzand efecte imprevizibile (modificarea altor variabile, blocarea sistemului, etc.).

Recapitulare C - vectori

Exemplu:

```
int v[5], x, w[3], *p;
```

Presupunand ca alocarea lui "v", "x", "w", "p" s-a facut adiacent si in ordinea declararii (regula de alocare poate diferi de la un compilator la altul):



atunci sunt echivalente si dau lui "x" valoarea 30:

```
v[5]=30;  w[-1]=30;
```

Recapitulare C - vectori

De regula, odata cu un vector "v" se retine intr-o variabila separata "n" si numarul de componente folosite efectiv dintre cele alocate si se asigura ca niciodata "n" nu va depasi aceasta valoare.

Recapitulare C - vectori

Pentru a transmite ca parametru actual numele unui vector, parametrul formal trebuie declarat pointer la tipul componentelor vectorului. Practic, se va transmite in apelul functiei doar o copie a adresei de inceput a vectorului, nu si a componentelor sale; pe parcursul apelului, aplicarea lui "*", "[numar]" va conduce la modificarea componentelor vectorului initial, ca si cand acestea ar fi fost transmise prin referinta.

Recapitulare C - vectori

Exemplu:

```
int v[5];
 ...
void f(int *p){p[2]=40;}
 ...
f(v);
```

Daca adresa fizica de inceput a vectorului "v" este "a", atunci functiei i se transmite doar o copie a valorii "a"; in apel, "[2]" se aplica de fapt valorii "a", instructiunea "p[2]=40;" avand acelasi efect cu "v[2]=40;", astfel ca dupa apel vectorul "v" va ramane modificat.

Recapitulare C - vectori

Notam ca puteam declara pe "f" in mod echivalent:

```
void f(int *p){p[2]=40;}
void f(int p[100]){p[2]=40;} /*nu conteaza dim.specificata la "p"*/
void f(int p[]){p[2]=40;}
```

deci la declararea parametrilor-vectori nu conteaza
dimensiunea si poate fi omisa (in toate cazurile, ei
se implementeaza ca pointeri).

Recapitulare C - vectori

Daca vrem sa transmitem prin valoare componentele vectorului, putem incapsula vectorul intr-o structura si transmite structura (la atribuirea sau transmiterea prin valoare a unei structuri se copiaza valorile tuturor membrilor ei, chiar daca sunt masive).

Recapitulare C - vectori

Exemplu:

```
struct vector{int componenta[5];} v;
 ...
void f(struct vector p){p.componenta[2]=40;}
 ...
f(v);
```

La apel se copiaza tot continutul variabilei structura "v" in variabila structura (locala functiei) "p", a.i. in cadrul apelului prin instructiunea "p.componenta[2]=40;" se opereaza de fapt pe aceasta copie, iar dupa apel "v" va ramane nealterat.

Recapitulare C - vectori

Pentru a lucra unitar, putem folosi aceasta incapsulare si atunci cand dorim transmiterea vectorului prin referinta, dar atunci vom transmite (prin valoare) adresa structurii.
De asemenea, pentru a lucra elegant, putem incapsula in structura si variabila continand numarul curent de elemente folosite.

Recapitulare C - vectori

Exemplu:

```
struct vector{int componenta[5], n;} v;
 ...
void citeste(struct vector *p){
  int i;
  scanf("%d",&p->n);
  for(i=0; i<p->n; ++i) scanf("%d",&p->componenta[i]);
}

void afisaza(struct vector p){
  int i;
  for(i=0; i<p.n; ++i) printf("%d ",p.componenta[i]);
  printf("\n");
}
 ...
citeste(&v);
afisaza(v);
```

Recapitulare C

Reamintim ca in C este implementata in mod nativ doar transmiterea parametrilor prin valoare, nu si cea prin referinta; daca dorim transmiterea prin referinta, trebuie sa o emulam explicit folosind transmiterea prin valoare.

Mai exact, vom transmite (prin valoare) adrese, pe care le vom deferentia (cu "*", "->", etc.) in cadrul functiilor pentru a ajunge la locatiile parametrilor actuali respectivi.

Cele doua functii "citeste" si "afisaza" de mai sus ilustreaza aceste tehnici.

Recapitulare C - matrici

MASIVELE multidimensionale sunt implementate prin iterarea mecansimelor de vector. Mai exact, un masiv n-dimensional este un vector de masive n-1 dimensionale.

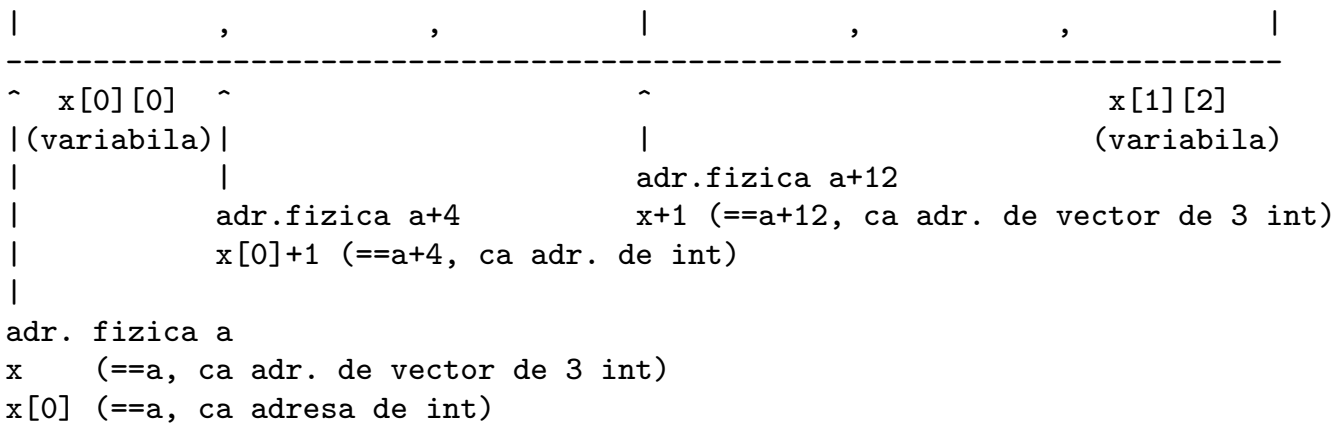
In particular, o MATRICE este un vector de vectori (componentele sale ca vector sunt liniile sale ca matrice). De aici rezulta:

- componentele matricii se aloca succesiv, pe linii (liniile sale se aloca succesiv si adiacent);
- numele unei matrici fara indici sau cu un singur indice desemneaza constante pointer si nu sunt expresii lvalue; daca punem ambii indici, este desemnata locatia unei componente de matrice, care este o variabila.

Recapitulare C - matrici

Exemplu:

```
int x[2][3];
```



Componentele alocate ale matricii corespund indicilor de la 0,0 la dimensiunile specificate la alocare - 1. Alaturi de matricea propriuzisa, putem folosi doua variabile auxiliare care sa retina numarul de linii si coloane curent utilizate.

Recapitulare C - matrici

Iterand regula de la vectori, pentru a transmite ca parametru actual numele unei matrici, parametrul formal trebuie declarat pointer la vector (linie).

Exemplu:

```
int x[2][3];
 ...
void f1(int p[100][3]){}
void f2(int p[][3]){}
void f3(int (*p)[3]){}
 /* in toate cele 3 cazuri, "p" se implementeaza
    ca pointer la vector de 3 int*/
 ...
f1(x); f2(x); f3(x);
 /* apeluri legale; se transmite doar o copie a adresei de
    inceput a matricii, nu si copii ale elementelor*/
```

Recapitulare C - matrici

Observatie: daca am fi declarat:

```
void f3(int *p[3]){}
```

atunci "p" ar fi fost pointer la pointer la int, declaratia fiind echivalenta cu oricare din urmatoarele:

```
void f3(int *p[]){}
void f3(int **p){}
```

si nu pointer la vectori de 3 int; aceasta influenteaza aritmetica pointerilor aplicata la indexarea lui "p" iar apelul "f3(x)" nu ar fi fost legal.

Recapitulare C

Reamintim ordinea prioritatilor operatorilor folositi mai sus (creste in sus):

```
.   ->
  ()
  []
++    --
 *   &
```

in particular, daca "a" si "b" sunt de tipul potrivit:

```
*a->b[3]  echivaleaza cu  *((a->b)[3])
&a.b(3)   echivaleaza cu  &((a.b)(3))
*a.b++    echivaleaza cu  *((a.b)++)
++a[3]    echivaleaza cu  ++(a[3])
```

(obs: a doua echivalenta este valabila in C++, unde putem avea membri functie iar functiile pot returna lvalue)

Recapitulare C - matrici

Pentru a transmite prin valoare matricile inclusiv ca elemente, sau/si pentru a lucra elegant, le putem incapsula in structuri.

Exercitiu: scrieti functii similare lui "citeste" si "scrie" de la vectori, care sa lucreze cu matrici de dimensiune alocata data.

Recapitulare C - structuri

STRUCTURILE sunt date complexe formate din mai multe date mai simple (numite MEMBRII structurii) incapsulate intr-un tot logic.

Exemplu:

```
struct persoana{
  int v;                        /* varsta */
  char i, np[2][100];           /* initiala, nume si prenume */
} x, *y, z[10];

struct persoana t;
```

Am definit tipul structura "persoana", variabilele "x" si "t" de tip "persoana" (notam ca putem defini variabile odata cu tipul dar si separat), variabila "y" pointer la "persoana", vectorul "z" alocat de 10 persoane.

Recapitulare C - structuri

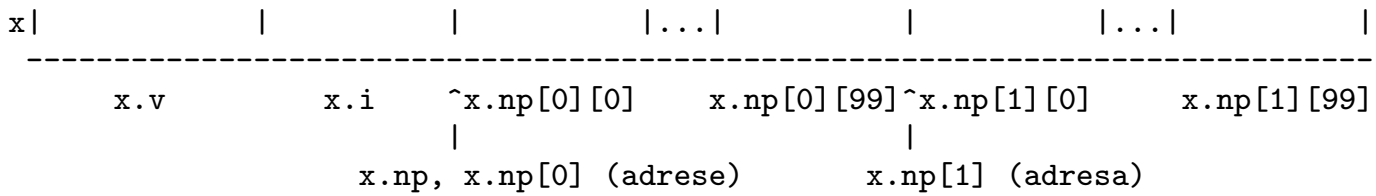
O data de tip "persoana" este formata din 3 date mai simple, de tip "int", "char", respectiv "matrice de 2 x 100 char". Locatia unei variabile "persoana" contine sub-locatii pentru stocarea acestor componente, iar acestea pot fi accesate prin constructii de forma "nume_variabila.nume_membru".

Recapitulare C - structuri

Exemplu:

```
struct persoana{
  int v;                        /* varsta */
  char i, np[2][100];           /* initiala, nume si prenume */
} x, *y, z[10];

struct persoana t;
```



Notam ca "x", "x.v", "x.i" sunt variabile (in particular expresii lvalue), dar "x.np", "x.np[0]", "x.np[1]" nu sunt - aceste expresii, asemeni numelor vectorilor, desemneaza adrese, nu locatii (in particular nu sunt expresii lvalue); totusi, elementelor matricii "x.np" li se rezerva locatii in cadrul locatiei lui "x", iar "x.np[0][0]" ... "x.np[1][99]" sunt variabile si corespund acestor elemente.

Recapitulare C - structuri

Putem atribui/pasa prin valoare structurile si ca un tot, si la nivelul membrilor lor.

Exemplu:

```
struct persoana{
  int v;                          /* varsta */
  char i, np[2][100];             /* initiala, nume si prenume */
} x, *y, z[10];

struct persoana t;

t.v = x.v;
  /* am atribuit doar membrul "v" al lui "x" membrului "v"
    al lui "t"; restul membrilor lui "t" raman cu valoarea veche */
```

Recapitulare C - structuri

```
struct persoana{
  int v;                          /* varsta */
  char i, np[2][100];             /* initiala, nume si prenume */
} x, *y, z[10];

struct persoana t;

t = x;
  /* se atribuie toti membrii lui "x" membrilor corespunzatori ai
    lui "t"; este echivalent cu:
                    t.v=x.v; t.i=x.i;
                    for(j=0; j<2; ++j)
                      for(k=0; k<100; ++k)
                        t.np[j][k]=x.np[j][k];
  */
```

Recapitulare C - structuri

```
struct persoana{
  int v;                          /* varsta */
  char i, np[2][100];             /* initiala, nume si prenume */
} x, *y, z[10];

struct persoana t;

void f(struct persoana q){q.v=10;}
  ...
f(x);
 /* se transmite o copie a intregii valori (structuri) a variabilei
    "x" in apelul functiei "f", ca valoare a variabilei (locale) "q";
    evident, pe parcursul apelului, modificarea lui "q" nu va
    afecta pe "x" (dupa apel "x.v" va avea aceeasi valoare ca
    inaintea apelului) */
```

Recapitulare C - structuri

Putem accesa membrii unei structuri pointate folosind operatorul "->".

Exemplu:

```
struct persoana{
  int v;                          /* varsta */
  char i, np[2][100];             /* initiala, nume si prenume */
} x, *y, z[10];

struct persoana t;

y=&x;
y->v=10;
 /* echivalent cu x.v=10; putem scrie si: (*y).v=10; */
```

Recapitulare C - structuri

```
struct persoana{
  int v;                          /* varsta */
  char i, np[2][100];             /* initiala, nume si prenume */
} x, *y, z[10];

struct persoana t;

void f(struct persoana *q){q->v=10;}
  ...
f(&x);
 /* se transmite o copie a adresei variabilei "x" (nu a
    continutului sau) ca valoarea a lui "q"; pe parcursul
    apelului, prin "q->" se va opera de fapt asupra locatiei
    lui "x", a.i. dupa apel "x.v" va avea valoarea 10 */
```

Recapitulare C - alocare dinamica

```
 Functiile de alocare/dezalocare dinamica a memoriei in C
sunt urmatoarele (zonele de memorie alocate/dezalocate
sunt luate din heap, iar pentru folosirea acestor functii
trebuie inclus stdlib.h):

#include<stdlib.h>

void *malloc(size_t dim);
  ==> aloca un bloc liber de "dim" octeti din heap si
        returneaza adresa lui;
      daca alocarea nu e posibila (nu exista in heap un
        bloc liber de dimensiune "dim") sau daca
        "dim" = 0, returneaza NULL;
```

Recapitulare C - alocare dinamica

```
#include<stdlib.h>

void *calloc(size_t nr, size_t dim);
  ==> aloca un bloc liber de "nr*dim" octeti din heap,
        il initializeaza cu toti octetii 0 si returneaza
        adresa lui;
      daca alocarea nu e posibila (nu exista in heap un bloc
        liber de dimensiune "nr*dim") sau daca "nr" = 0 sau
       daca "dim" = 0, returneaza  NULL;
```

Recapitulare C - alocare dinamica

```
#include<stdlib.h>

void *realloc(void *adr, size_t dim);
  ==> presupunand ca "adr" este adresa unui bloc alocat dinamic
anterior, ajusteaza dimensiunea acestui bloc  la "dim" octeti;
  daca "dim" este mai mare decat dimensiunea initiala a blocului
respectiv si nu exista suficient spatiu liber in continuarea
acestuia, se incearca alocarea unui nou bloc (de la o alta adresa)
de "dim" octeti iar in caz de succes se copiaza continutul primului
bloc la inceputul celui de al doilea bloc si se dezaloca primul bloc;
  in caz de succes (ajustarea blocului sau alocarea in alta parte a
reusit), returneaza adresa blocului final; aceasta adresa va
coincide cu "adr", in caz ca a fost posibila ajustarea, sau va fi o
alta adresa, daca s-a alocat un alt bloc;
  in caz de esec (nu a fost posibila nici ajustarea nici alocarea in
alta parte) ret. NULL, iar blocul initial nu este dezalocat/alterat;
  daca "dim" = 0 ret.NULL, iar bl.initial este dezalocat (ca la free);
  daca "adr" = NULL, realloc actioneaza ca "malloc(dim)";
```

Recapitulare C - alocare dinamica

```
#include<stdlib.h>

void free(void *adr):
  ==> presupunand ca "adr" este adresa unui bloc alocat
        dinamic anterior, dezaloca acel bloc;
```

Recapitulare C - alocare dinamica

Observatii:

- "size_t" este un tip intreg; pentru siguranta/portabilitate se recomanda conversia explicita a tipurilor intregi spre/ dinspre acest tip;

Recapitulare C - alocare dinamica

- cand se aloca dinamic un bloc (cu malloc, calloc, realloc) sistemul retine automat intr-o evidenta proprie adresa "a" a blocului si dimensiunea "n" a lui;
  cand apelam free dand ca parametru adresa "a", acesta cauta automat in evidentele respective "a" si afla de acolo dimensiunea "n" a memoriei ce trebuie dezalocate; de aceea e suficient sa retinem si sa transmitem ca informatie doar adresa respectiva, nu si dimensiunea zonei; totodata lui realloc sau free nu trebuie sa-i dam ca parametru "adr" decat o adresa returnata anterior de un malloc, calloc sau realloc, altfel ea nu va fi gasita in evidentele respective iar efectul este imprevizibil; practica arata ca apelul "free(NULL)" nu are nici un efect; exemple:

```
int *p,*q; p=(int *)malloc(10); q=p;
free(q); /* dezaloca 10 octeti */

int a; free(&a); /* efect imprevizibil */
```

Recapitulare C - alocare dinamica

- in urma dezalocarii unui bloc, acesta nu isi pierde automat continutul iar pointerii care il adresau nu devin automat NULL, insa blocul sau o parte din el vor putea fi cuprinse in alt bloc alocat ulterior (si astfel continutul sau va putea fi accesat/modificat prin alti pointeri fara stirea utilizatorului); exemple:

```
int *p,*q;
p=(int *)malloc(sizeof(int));
q=(int *)malloc(sizeof(int));
  /* p si q pointeaza locatii dinamice diferite*/
*p=10; *q=20; printf("%d",*p);/* afisaza 10 */
```

Recapitulare C - alocare dinamica

```
int *p,*q;
p=(int *)malloc(sizeof(int));
  /* p pointeaza o noua locatie dinamica */
free(p);
  /* p nu devine NULL si pointeaza aceeasi locatie */
q=(int *)malloc(sizeof(int));
  /* intamplator va aloca aceeasi locatie pe care o
     pointeaza p, deoarece acum e considerata libera */
*q=10; *p=20; printf("%d",*q);/* afisaza 20 */
```

Recapitulare C - alocare dinamica

- calloc este util pentru a aloca vectori sau pentru a aloca zone initializate cu 0 fara sa mai initializam noi explicit; exemplu:

```
int *v; v=(int *)calloc(10,sizeof(int));
```

este echivalent cu:

```
int *v,i;
v=(int *)malloc(10*sizeof(int));
for(i=0;i<10;++i)v[i]=0;
```

- realloc este util pentru a implementa vectori de dimensiune variabila; cu ajutorul lor putem implementa stive secvential si dinamic (a se vedea mai jos).

Cuprins

Alocarea secvențială în limbajul C

De regula sunt folositi vectori, deoarece mecansimul nativ al limbajului C de implementare a vectorilor implementeaza exact mecanismele generale de alocare secventiala descrise mai sus. Mai exact:

- modul de alocare al componentelor vectorului asigura ca elementele listei sunt stocate in locatii de memorie de aceeasi dimensiune, adiacente, in ordinea in care apar in lista, iar aceste locatii contin doar valorile elementelor si nimic in plus;
- numele vectorului desemneaza adresa primului element, sizeof-ul tipului componentelor desemneaza dimensiunea locatiei unui element, iar numarul de elemente se poate retine intr-o variabila separata;
- accesarea elementului de pe pozitia i dupa formula data este efectuata prin aplicarea indexarii "[i]" (formula de calcul corespunzatoare este generata automat de compilator).

Alocarea secvențială în limbajul C

Pentru eficientizare, se mai pot folosi variabile auxiliare care sa retina indicele elementului din varf, in cazul stivelor, respectiv din varf si de la baza, in cazul cozilor.

Daca se doreste ca lista sa poate creste indefinit (in limita memoriei fizice existente), vectorul se poate (re)aloca dinamic; daca se doreste ca lista sa nu poata creste peste o lungime (numar de elemente) MAX fixata, se poate folosi un vector alocat static de dimensiune MAX.

Alocarea secvențială în limbajul C

Exemplu: stiva alocata secvential si dinamic:

```
#include<stdlib.h>

struct stack{int *v, n, i;};
 /* v = adr. bufferului cu elemente (gestionat ca un vector)
    i = indicele elementului din varf
    n = dimensiunea curenta a bufferului
 */

void init(struct stack *s)
 {s->v=NULL; s->n=0; s->i=-1;}
 /* initializeaza stiva pentru prima folosire (constructor) */

void clear(struct stack *s)
 {free(s->v); s->v=NULL; s->n=0; s->i=-1;}
 /* dezaloca resursele stivei dupa ultima folosire (destructor) */
```

Alocarea secvențială în limbajul C

```
int push(struct stack *s, int x){ /* insereaza un element */
  if(s->i==s->n-1){
    int *p;
    if((p=realloc(s->v,s->n+10))==NULL)return 0;
    s->v=p; s->n+=10;
  }
  ++s->i; s->v[s->i]=x;
  return 1;
}

int pop(struct stack *s, int *x){ /* extrage un element */
  if(s->i==-1)return 0;
  *x=s->v[s->i]; --s->i;
  if(s->i==-1)clear(s);
  else if(s->i<s->n-20){s->n-=10; realloc(s->v,s->n);}
  return 1;
}

int empty(struct stack *s) /* test stiva vida */
 {return s->i==-1;}
```

Alocarea secvențială în limbajul C

Observatii:

- pentru eleganta, am incapsulat intr-o structura elementele definitorii ale stivei: adresa primei componente (primului element), numarul de componente curent alocate, indicele varfului; totodata am simulat un stil de organizare a codului specific limbajului C++, mimand constructori si destructori (care insa aici sunt functii obisnuite, ce trebuie apelate explicit); acest stil va fi folosit si in exemplele urmatoare;

- pentru a evita alocarea/dezalocarea succesiva atunci cand facem multe operatii push/pop alternate iar s->i==s->n-1, cand a trebuit sa extindem bufferul (la push) acesta a fost extins cu 10 elemente (nu doar cu 1), iar restrangerea acestuia (la pop) a fost facuta doar cand s->i<s->n-20 si atunci l-am restrans cu 10 elemente;

Alocarea secvențială în limbajul C

- functia "init" trebuie apelata o singura data pentru fiecare stiva din program, atunci cand aceasta isi incepe existenta (inaintea primei sale folosiri); ea este asemeni unui constructor din limbajul C++; functia "clear" trebuie apelata o singura data pentru fiecare stiva din program, atunci cand aceasta isi inceteaza existenta (dupa ultima sa folosire); ea este asemeni unui destructor din limbajul C++;

- functiile "push", "pop" de mai sus returneaza 1 cand operatia reuseste si 0 cand nu; am presupus ca esecul poate fi cauzat doar de imposibilitatea alocarii sau extinderii bufferului ("push") nu si la restrangerea acestuia ("pop") - de aceea nu am testat succesul lui "realloc(s->v,s->n)" din "pop".

Alocarea secvențială în limbajul C

O alta observatie interesanta este ca functiile "init", "clear", "empty", "push", "pop" formeaza o baza de operatii primitive, deasupra carora pot fi definite toate celelalte operatii care pot fi asociate conceptului de stiva.

De fapt este suficient sa consideram ca primitive doar "init", "empty", "push", "pop", deoarece "clear" se poate defini deasupra lor (facand "pop" repetat, atata timp cat not "empty" - exercitiu).

Alocarea secvențială în limbajul C

Exemplu: functie de copiere a unei stive in alta stiva
(pentru simplitate, nu am mai testat/returnat succesul/
esecul):

```
void copy(struct stack *ss, struct stack *sd){
  struct stack saux; int x;
  init(&saux);
  while(!empty(ss)){pop(ss,&x); push(&saux,x);}
  clear(sd);
  while(!empty(&saux))
     {pop(&saux,&x); push(ss,x);push(sd,x);}
  clear(&saux);
}
```

Deci, pentru a schimba modul de implementare a stivelor,
e suficient sa modificam tipul "stack" si corpul primitivelor
"init", "clear", "empty", "push", "pop"; codul scris pe nivelul
urmator (ex. "copy") ramane neschimbat si va lucra automat
cu noul concept de stiva.

Alocarea secvențială în limbajul C

Exemplu de program cu stive ce poate utiliza codul scris mai sus:

Problema: verificarea daca un sir de paranteze este corect imperecheat; in acest sens asimilam o paranteza deschisa cu un numar intreg pozitiv si o paranteza inchisa cu un numar intreg negativ; modulul numarului da tipul parantezei; de exemplu:

```
1 2 3 -3 -2 4 -4 -1 este corect imperecheat
1 2 3 -2 4 -4 -1    nu este corect imperecheat
```

Alocarea secvențială în limbajul C

Rezolvare:

```
#include<stdio.h>
 /* aici inseram codul de implementare a stivelor
     de mai sus */
int main(){
  int v[]={1, 2, 3, -3, -2, 4, -4, -1},
      n=sizeof(v)/sizeof(int), i, x;
  struct stack s;
  init(&s);
  for(i=0;i<n;++i)
    if(v[i]>0) push(&s,v[i]);
    else if(empty(&s)) break;
                  else {pop(&s,&x); if(v[i]!=-x)break;}
  if(i==n && empty(&s))printf("Sir corect.\n");
                  else printf("Sir gresit.\n");
  clear(&s);
  return 0;
}
```

Alocarea secvențială în limbajul C

```
 Observatie: "clear(&s);" din final nu este necesar (deoarece
la terminarea normala a programului memoria dinamica
alocata de acesta si nedezalocata explicit se dezaloca automat)
dar este didactic (pentru ilustrarea unui stil general de lucru).
```

Alocarea secvențială în limbajul C

```
Exemplu: coada alocata secvential si static (de dimensiune
 maxima MAXQ):

#define MAXQ 10

struct queue{int a[MAXQ], b, v;};
 /* a = bufferul cu elemente (vector alocat cu MAXQ componente)
    b = indicele bazei (al pozitiei neocupate unde se va insera
          un element)
    v = indicele varfului (al elementului care se va extrage)
    bufferul este parcurs circular
    b, v vor avansa cu 1 modulo MAXQ
    conditia de coada vida: b==v
    conditia de coada plina: (b+1)%MAXQ == v
    inserare x: a[b]=x; b=(b+1)%MAXQ
    extragere in x: x=a[v]; v=(v+1)%MAXQ
    notam ca in "a" se pot exploata efectiv doar MAXQ-1 componente
 */
```

Alocarea secvențială în limbajul C

```
void init(struct queue *q){q->b=q->v=0;}

void clear(struct queue *q){init(q);}

int full(struct queue *q){return (q->b+1)%MAXQ == q->v;}

int empty(struct queue *q){return q->b == q->v;}

int insert(struct queue *q, int x){
  if(full(q)) return 0;
  q->a[q->b]=x; q->b=(q->b+1)%MAXQ;
  return 1;
}

int extract(struct queue *q, int *x){
  if(empty(q)) return 0;
  *x=q->a[q->v]; q->v=(q->v+1)%MAXQ;
  return 1;
}
```

Alocarea secvențială în limbajul C

```
Exemplu de program cu cozi ce poate utiliza codul scris
 mai sus:

Problema: determinarea componentei conexe a unui
 graf neorientat ce contine un varf dat (dorim doar
 varfurile din componenta conexa, nu si muchiile);
 graful este dat prin matricea de adiacenta.
Rezolvare:

#include<stdio.h>

 /* aici inseram codul de implementare a cozilor de mai sus */

#define MAXG MAXQ

struct graph{int a[MAXG][MAXG], n;};
  /* a = matricea de adiacenta (simetrica)
     n =  numarul de varfuri
    matricea fiind considerata simetrica, se va initializa/consulta
     doar jumatatea superior diagonala */
```

Alocarea secvențială în limbajul C

```
/* pune in vectorul "v" varfurile din acceasi componenta
  conexa a grafului "*g" ca si "k", iar in "*nv" numarul
  acestor varfuri */
void connex(struct graph *g, int k, int *v, int *nv){
  int vizitat[MAXG], i, x;  struct queue q;
  init(&q);
  for(i=0;i<MAXG;++i)vizitat[i]=0;
  insert(&q, k); vizitat[k]=1;
  do{
    extract(&q,&x);
    for(i=0;i<g->n;++i)
      if((i<x && g->a[i][x]==1)||(x<i && g->a[x][i]==1))
        if(!vizitat[i])
          {insert(&q,i); vizitat[i]=1;}
  }while(!empty(&q));
  *nv=0;
  for(i=0;i<g->n;++i) if(vizitat[i]) {v[*nv]=i; ++*nv;}
  clear(&q);
}
```

Alocarea secvențială în limbajul C

```
int main(){
  struct graph g; int i,j,k,nr; int r[MAXG];
  for(i=0;i<MAXG;++i) for(j=0;j<MAXG;++j) g.a[i][j]=0;
  g.n=8;
  g.a[1][2]=1;  g.a[2][3]=1;  g.a[1][5]=1; g.a[4][6]=1;
  k=2;
  connex(&g,k,r,&nr);
  printf("Varfurile conexe cu %d sunt: ",k);
  for(i=0;i<nr;++i) printf("%d ",r[i]);
  printf("\n");
  return 0;
}
```

Alocarea secvențială în limbajul C

Se poate implementa coada alocata secvential si dinamic, dar este ineficient deoarece redimensionarea periodica a buffer-ului cu elemente poate conduce uneori la translatarea unora dintre ele.

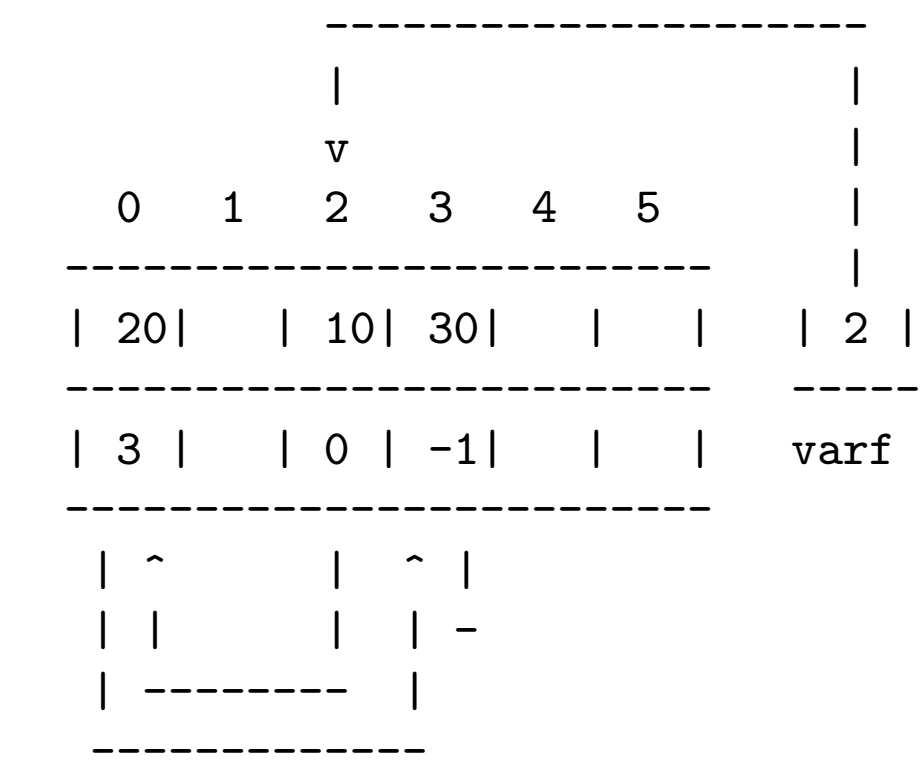
Cuprins

Alocarea înlănțuită în limbajul C

Se poate face in mai multe feluri. O modalitate de implementare in C a listelor alocate inlantuit este:
- folosim o matrice cu doua linii si un numar oarecare de coloane; fiecare coloana va fi nefolosita sau va stoca un element al listei;
  pentru fiecare element, in coloana sa se va stoca pe prima linie valoarea elementului, iar pe a doua linie indicele coloanei corespunzatoare elementului "urmator" (se pot face artificii de programare a.i. tipul componentelor matricii sa fie compatibil atat cu tipul elementelor listei cat si cu tipul "int" al indicilor, sau se poate inlocui matricea cu un vector de structuri cu doi membri);
  in a doua linie a coloanei ultimului element se retine un indice invalid (de ex. -1);
- intr-o variabila separata se retine indicele coloanei primului element; daca lista este vida, se retine indicele invalid (-1);
- accesarea unui element se face aplicand iterat mecanisme de operare cu matrici (indexare).

Alocarea înlănțuită în limbajul C

Exemplu: stiva formata din elementele (de la varf spre celalalt capat) 10, 20, 30 se poate stoca:



Alocarea înlănțuită în limbajul C

Pentru a eficientiza cautarea de coloane libere se mai poate folosi un vector caracteristic (cu 0/1) al multimii coloanelor ocupate si o variabila continand cardinalul acestei multimi.
De asemenea, in cazul cozilor se poate folosi si o variabila care retine indicele coloanei ultimului element.

Matricea poate fi alocata static (de dimensiune fixata) sau dinamic (mai dificil de implementat, deoarece matricea poate avea coloane intermediare nefolosite - am putea translata elementele (cu recalcularea indicilor stocati in a doua linie) a.i. coloanele nefolosite sa ajunga la sfarsit iar realocarea sa se poata face mai usor, dar este neperformant).

Alocarea înlănțuită în limbajul C

```
Exemplu: stiva (de elemente "int") alocata inlantuit,
         folosind o matrice alocata static:

#define MAXS 10
struct stack{int a[2][MAXS], b[MAXS], v, n;};
 /* a = bufferului cu elemente (matrice)
    b = vectorul caracteristic al multimii
          coloanelor ocupate
    v = indicele coloanei elementului din varf
    n = numarul curent de coloane ocupate
          (i.e. de elemente ale stivei)
 */
void init(struct stack *s){
  int i;
  s->v=-1; s->n=0;
  for(i=0;i<MAXS;++i) s->b[i]=0;
}
void clear(struct stack *s){init(s);}
```

Alocarea înlănțuită în limbajul C

```
int full(struct stack *s)
  {return s->n == MAXS;}
int empty(struct stack *s)
  {return s->n == 0;}
int push(struct stack *s, int x){
  int i;
  if(full(s)) return 0;
  for(i=0;i<MAXS;++i) if(s->b[i]==0) break;
  s->a[0][i]=x; s->a[1][i]=s->v; s->v=i;
  s->b[i]=1; ++s->n;
  return 1;
}
int pop(struct stack *s, int *x){
  if(empty(s)) return 0;
  *x=s->a[0][s->v];
  s->b[s->v]=0;  s->v=s->a[1][s->v]; --s->n;
  if(s->v==-1) clear(s);
  return 1;
}
```

Alocarea înlănțuită în limbajul C

Exercitii:

1. Adaugati codul programului care testeaza imperecherea parantezelor.
2. Implementati asemanator coada (de elemente "int") alocata inlantuit, folosind o matrice alocata static si adaugati codul programului pentru determinarea componentei conexe.

Alocarea înlănțuită în limbajul C

Modalitatea cea mai naturala (si mai des folosita) de implementare in C a listelor alocate inlantuit este:

- fiecare element al listei se stocheaza intr-o structura alocata dinamic, avand doi membri: unul retine valoarea elementului, celalalt adresa structurii corespunzatoare elementului "urmator"; in cazul ultimului element, se retine adresa NULL;
- intr-o variabila pointer separata se retine adresa structurii corespunzatoare primului element;
- accesarea unui element se face aplicand iterat mecanisme de operare cu structuri si pointeri (accesarea cu "." a membrilor, deferentierea cu "*" si "->", etc).

Pentru eficientizare, in cazul cozilor se poate folosi si o variabila pointer care retine adresa structurii corespunzatoare ultimului element.

Acest gen de implementare este prin excelenta dinamic.

Alocarea înlănțuită în limbajul C

```
Exemplu: stiva alocata inlantuit, folosind structuri
 si pointeri:

#include<stdlib.h>

struct element{int info; struct element *next;};

struct stack{struct element *v;};
 /* v = adresa structurii ce contine elementul din varful stivei;
        legaturile "next" vor merge de la varf spre celalalt capat
 */

void init(struct stack *s){s->v=NULL;}

int empty(struct stack *s){return s->v == NULL;}
```

Alocarea înlănțuită în limbajul C

```
int push(struct stack *s, int x){
  struct element *p;
  if((p=(struct element *)malloc(sizeof(struct element)))==NULL)
     return 0;
  p->info=x; p->next=s->v; s->v=p;
  return 1;
}

int pop(struct stack *s, int *x){
  struct element *p;
  if(empty(s))return 0;
  *x=s->v->info;
  p=s->v; s->v=s->v->next; free(p);
  return 1;
}

void clear(struct stack *s){
  int x;  while(!empty(s)) pop(s,&x);
}
```

Alocarea înlănțuită în limbajul C

```
Observatie: putem asimila conceptul de stiva cu
 acel obiect al programului care este pointerul
 spre elementul sau din varf si sa scriem:

struct element{int info; struct element *next;};

 /* functia primeste (prin referinta) stiva sub forma pointerului
    spre elementul sau din varf */
void init(struct element **s){*s=NULL;}

 ...

struct element *t; /* declaram o stiva */
init(&t);          /* initializam stiva */
```

Alocarea înlănțuită în limbajul C

dar incapsularea intr-o structura are mai multe avantaje:

- se respecta acelasi stil ca in programele anterioare;
 in particular putem refolosi fara modificari functia "copy"
 si programul ce verifica imperecherea parantezelor;

- putem imbogati usor conceptul de stiva, adaugand de
 exemplu un membru care retine numarul curent de
 elemente, pastrand incapsularea conceptului de stiva
 intr-un singur obiect al programului (o stiva = o
 structura, nu un ansamblu de date independente ce
 trebuie gestionate unitar).

Exercitiu: adaugati codul programului care testeaza
 imperecherea parantezelor.

Alocarea înlănțuită în limbajul C

Exemplu: coada alocata inlantuit, folosind structuri si pointeri:

```
#include<stdlib.h>

struct element{int info; struct element *next;};

struct queue{struct element *b,*v;};
 /* b = adresa structurii ce contine elementul de la baza cozii;
    v = adresa structurii ce contine elementul din varful cozii;
    legaturile "next" vor merge de la varf spre baza
 */

void init(struct queue *q){q->b=q->v=NULL;}

int empty(struct queue *q){return q->b == NULL;}
```

Alocarea înlănțuită în limbajul C

```
int insert(struct queue *q, int x){
  struct element *p;
  if((p=(struct element *)malloc(sizeof(struct element)))==NULL)
    return 0;
  p->info=x; p->next=NULL;
  if(empty(q)) q->b=q->v=p;
  else {q->b->next=p;  q->b=p;}
  return 1;
}

int extract(struct queue *q, int *x){
  struct element *p;
  if(empty(q))return 0;
  *x=q->v->info;
  p=q->v; q->v=q->v->next; free(p);
  if(q->v==NULL) q->b=NULL;
  return 1;
}
```

Alocarea înlănțuită în limbajul C

```
void clear(struct queue *q){
  int x;  while(!empty(q)) extract(q,&x);
}
```

Alocarea înlănțuită în limbajul C

Exercitiu: adaugati codul programului pentru determinarea componentei conexe.

Considerații finale

Notam ca investigarea structurilor de date a fost facuta la mai multe niveluri de abstractizare:

1 - nivelul matematic (lista ca multime finita, evntual vida, total ordonata);
2 - nivelul implementarii intr-un sistem informatic abstract;
3 - nivelul implementarii pe o arhitectura data (in cazul nostru am presupus arhitectura unui PC uzual), folosind un limbaj de programare dat (in cazul nostru C).

Diversele concepte ale structurilor de date au fost definite la diverse niveluri de abstractizare:

- la nivelul 1 s-au definit notiunile de lista, stiva, coada;
- la nivelul 2 s-au definit modalitatile de alocare secventiala si inlantuita;
- la nivelul 3 s-au definit tehnici de implementare a listelor folosind masive, structuri, pointeri.

Considerații finale

In general un concept de pe un nivel inalt se poate implementa prin concepte de pe nivelul imediat inferior in mai multe moduri, de exemplu:

- o stiva se poate aloca secvential sau inlantuit;
- o stiva alocata inlantuit se poate implementa in C folosind matrici cu doua linii sau structuri si pointeri.

Considerații finale

Exercitii: incercati implementarea dupa modelele de mai sus a cator mai multe combinatii de concepte (ex: lista dublu inlantuita circulara alocata folosind o matrice cu trei linii) si imaginati probleme rezolvate cu ajutorul acestora (programe complete).