



PROIECTAREA ALGORITMILOR

Lect. univ. dr. Adrian Runceanu



Curs 3

Alocarea dinamică de memorie în C++ (partea II)

Conținutul cursului

- 1. Pointeri la funcții**
- 2. Folosirea pointerilor și tablourilor ca parametri în funcții**
- 3. Alocarea dinamică a memoriei**
 - 3.1. Operatorul `new`**
 - 3.2. Operatorul `delete`**
- 4. Structuri implementate dinamic:**
 - 4.1. `Stiva`**
 - 4.2. `Coadă`**
 - 4.3. `Lista simplu înlănțuită`**
 - 4.4. `Lista dublu înlănțuită`**

1. *Pointeri la funcții*

Numele unei funcții este un pointer spre funcția respectivă.

- ✓ El poate fi folosit ca parametru efectiv la apeluri de funcții.
- ✓ În felul acesta, o funcție poate transfera funcției apelate un pointer spre o funcție.
- ✓ Aceasta, la rândul ei, poate apela funcția care i-a fost transferată în acest fel.

1. *Pointeri la funcții*

În instrucțiunile în care se atribuie pointeri la funcții, tipurile de funcții trebuie să corespundă exact.

Sintaxa de declarare este:

```
tip functie(*pointer_functie)(lista_param);
```

Exemple:

1. Se observă importanța *folosirii parantezelor rotunde pentru pointeri la funcții*.

Fără acestea nu s-ar mai considera un *pointer la funcție* ci o funcție care întoarce un pointer de un anumit tip.

a) Fie declarația

int *s(int *s1, int *s2);

✓ În acest caz avem o funcție care *întoarce pointer la int*.

b) În declarația

```
int (*s)(int *s1, int *s2);
```

avem un *pointer la o funcție care întoarce o valoare de tip int.*

c) ***int (*f (int,int)) [10];***

✓ În acest caz se definește o variabilă de tipul unei *funcții cu doi parametri de tip int care întoarce ca valoare un pointer la un tablou de 10 întregi.*

2. Schemă de program care folosește un tablou de pointeri la funcții predefinite.

```
// tablou de pointeri la funcții predefinite
```

```
#include<iostream.h>
```

```
#include<math.h>
```

```
double(*func_mat[ ])(double) = { sin, sinh, cos, cosh, tan,  
    atan };
```

```
int main()
```

```
{  
    int i;  
    double x;  
    for (x=0.01; x<1.01; x+=0.01)  
        for(i=0; i<6; i++)  
            cout<<(*func_mat[i])(x)<<endl;  
}
```


3) În continuare, un exemplu de program care folosește un **tablou de pointeri la funcții definite**:

```
#include<iostream.h>
typedef int (*functie)(); // Pointer la funcție de tip întreg
void tiparire()
{
    cout<<“\nFuncția de tiparire”;
}

void afisare()
{
    cout<<“\nFuncția de afisare”;
}
```

```
void scriere()
{
    cout<<“\nFuncția de scriere”;
}
void verificare()
{
    cout<<“\nFuncția de verificare”;
}
/* declaratia tabloului de pointeri la funcții */
functie tf[ ] = { scriere, verificare, afisare, tiparire };
int main()
{
    int i;
    for(i=0; i<4; i++)
    /* Apelul unei funcții din tabloul de pointeri la funcții */
        (*tf[i])() ;
}
```

Conținutul cursului

1. Pointeri la funcții
2. Folosirea pointerilor și tablourilor ca parametri în funcții
3. Alocarea dinamică a memoriei
 - 3.1. Operatorul **new**
 - 3.2. Operatorul **delete**
4. Structuri implementate dinamic:
 - 4.1. **Stiva**
 - 4.2. **Coadă**
 - 4.3. **Lista simplu înlănțuită**
 - 4.4. **Lista dublu înlănțuită**

2. Folosirea pointerilor și tablourilor ca parametri în funcții

- ✓ În lista de parametri a unei funcții pot apărea atât parametri de tip tablou cât și pointeri la un tip de dată.
- ✓ **În cazul tablourilor, nu se încarcă în stivă tot conținutul tablourilor ci numai adresa primului element.**
- ✓ *Este recomandată utilizarea ca parametri a pointerilor în locul tablourilor.*
- ✓ Pentru a vedea mai clar, vom defini aceeași funcție folosind pe rând tablouri și apoi pointeri.

Exemple:

1. Funcția **lungime** calculează lungimea unui șir de caractere.

Varianta din stânga definește **funcția prin utilizarea tabloului**, varianta din dreapta definește **funcția prin utilizarea pointerilor**.

```
int lungime (char s[ ])
{
    int i;
    for(i=0; s[i]; i++);
    return i;
}
```

```
int lungime(char *s)
{
    char *p;
    for(p=s; *p; p++);
    return p-s;
}
```

2. Următoarea funcție face *transferul datelor între două zone de memorie*.

Funcția **copiere** are ca parametri:

- **dest** = adresa zonei destinație
- **sursa** = adresa zonei sursă
- **nr** = numărul de octeți care vor fi copiați

```
void copiere(void *dest, void *sursă, int nr)
{
    while (nr-- > 0)
        *((char*) dest)++ = *((char*)sursa)++ ;
}
```

- ✓ Pointerii vor fi convertiți la *char* pentru a face accesul la nivel de octet.
- ✓ La fiecare secvență de ciclare se transferă un octet și se micșorează numărul de octeți transferați cu 1.
- ✓ În momentul în care *nr* devine 0 se iese din ciclul de transfer.

3. Produsul elementelor de pe diagonala principală a unei matrici de tip *int*.

Pointerul p este inițializat cu adresa de început a matricii

```
long produs(void *a, int n)
```

```
{
```

```
int *p=(int*)a,m;
```

```
long k=1;
```

```
for(m=n; m--; k=k * (*p), p = p + n+1) ;
```

```
return k;
```

```
}
```

Variabila k este înmulțită pe rând cu elementele de la adresa lui p

Prin adunarea cu valoarea n+1 se face mereu poziționarea pe următorul element de pe diagonala principală

Conținutul cursului

1. Pointeri la funcții
2. Folosirea pointerilor și tablourilor ca parametri în funcții
3. **Alocarea dinamică a memoriei**
 - 3.1. Operatorul **new**
 - 3.2. Operatorul **delete**
4. Structuri implementate dinamic:
 - 4.1. **Stiva**
 - 4.2. **Coadă**
 - 4.3. **Lista simplu înlănțuită**
 - 4.4. **Lista dublu înlănțuită**

3. Alocarea dinamică a memoriei

- ✓ Odată cu gestiunea pointerilor apare și posibilitatea utilizării variabilelor dinamice.
- ✓ Spre deosebire de variabilele statice, așa cum sugerează și denumirea, *variabilele dinamice sunt variabile care sunt create și eliminate la cererea programatorului și a căror dimensiune se poate modifica pe parcursul execuției programului.*
- ✓ Zona de memorie în care se face alocarea dinamică a variabilelor se numește *heap*.

3. Alocarea dinamică a memoriei

- ✓ *Alocarea de zone de memorie și eliberarea lor în timpul execuției programelor* permite gestionarea optimă a memoriei de către programe.
- ✓ Un astfel de mijloc de gestionare a memoriei se numește ***alocare dinamică a memoriei***.

3. Alocarea dinamică a memoriei

Alocarea dinamica a memoriei se poate executa prin utilizarea a doi operatori ai limbajului C++:

1. Operatorul de alocare a memoriei - **new**
2. Operatorul de dezalocare(eliberare) a memoriei - **delete**

Conținutul cursului

1. Pointeri la funcții
2. Folosirea pointerilor și tablourilor ca parametri în funcții
3. Alocarea dinamică a memoriei
 - 3.1. Operatorul **new**
 - 3.2. Operatorul **delete**
4. Structuri implementate dinamic:
 - 4.1. **Stiva**
 - 4.2. **Coadă**
 - 4.3. **Lista simplu înlănțuită**
 - 4.4. **Lista dublu înlănțuită**

3.1. Operatorul new

- ✓ Limbajul C++ permite alocări în zona heap prin intermediul operatorului **new**.
- ✓ Acesta este un **operator unar** și are aceeași prioritate ca și ceilalți operatori unari.
 - Operatorul **new** are ca valoare *adresa de început a zonei de memorie alocată în memoria heap* sau *zero (pointerul nul)* în cazul în care *nu se poate face alocarea*.
 - *Operandul operatorului new în cea mai simplă formă, este numele unui tip* (predefinit sau definit de utilizator).

3.1. Operatorul new

Exemplul 1:

```
int *pint;
```

```
pint = new int;
```

- ✓ Prin intermediul acestei expresii, se alocă în memoria heap o zonă de memorie în care se pot păstra date de tip int.
- ✓ Adresa de început a zonei alocate se atribuie pointerului *pint*.

Expresia:

***pint=100;** păstrează întregul 100 în zona respectivă.

3.1. Operatorul new

Exemplul 2:

```
int& i = *new int;
```

- ✓ Prin intermediul acestei declarații (definiții) se alocă în memoria *heap* o zonă de memorie în care se pot păstra date de tip `int`.
- ✓ Numele `i` permite referirea la întregul păstrat în zona respectivă.
- ✓ Expresia de atribuire: `i=100` păstrează întregul 100 în zona respectivă.

3.1. Operatorul new

- ✓ Zonele de memorie alocate cu ajutorul operatorului **new** pot fi inițializate.
- ✓ În acest scop se utilizează o expresie de forma:

new tip(expresie)

unde:

- ***tip*** – este numele unui tip de date
- ***expresie*** – este o expresie a cărei valoare inițializează zona de memorie

3.1. Operatorul new

Exemplul 1:

```
double *pdouble;  
pdouble = new double(3.14159265);
```

Această instrucțiune realizează următoarele:

- ✓ alocă în memoria **heap** o zonă de memorie în care se pastrează valoarea 3.14159265 în format real dublă precizie
- ✓ adresa de început a acestei zone de memorie se atribuie variabilei *pdouble*

3.1. Operatorul new

Exemplul 2:

```
double pi = *new double(3.14159265);
```

Prin această declarație se rezervă, în memoria heap, o zonă de memorie în care se păstrează valoarea 3.14159265 în format real dublă precizie.

Data respectivă se poate referi cu ajutorul numelui **pi**.

De exemplu, **pi** poate fi utilizat în mod obișnuit în expresii de forma:

- **pi*r*r**
- **sin(pi/2)**
- **x*180/pi**, etc.

3.1. Operatorul new

- ✓ O altă utilizare importantă este aceea de alocare a unei zone de memorie, în memoria *heap*, **pentru tablouri**. În acest scop, utilizăm o expresie de forma:

new tip[expresie]

- unde *expresie* – expresie de tip întreg;
- ✓ Prin această construcție se rezervă, în memoria *heap*, o zonă de memorie de *expresie * sizeof(tip) octeți*.
- ✓ Valoarea expresiei de mai sus, este adresa de început a zonei de memorie rezervată prin operatorul *new*.

Exemplu:

```
double *tab;
```

```
int m,n;
```

```
.....
```

```
tab=new double[m*n];
```

- ✓ Această instrucțiune rezervă în memoria *heap* o zonă de $m*n*sizeof(double)$ octeți.
- ✓ Adresa de început a acestei zone de memorie se atribuie pointerului *tab*.
- ✓ Elementele unei astfel de zone de memorie nu pot fi inițializate decât numai prin secvențe de program corespunzătoare.

De exemplu, pentru a initializa cu 0 cele $m*n$ elemente de tip *double* rezervate ca mai sus, putem folosi instrucțiunea *for* de mai jos:

```
for(int i=0; i<m*n; i++)
```

```
tab[i]=0.0;
```

Conținutul cursului

1. Pointeri la funcții
2. Folosirea pointerilor și tablourilor ca parametri în funcții
3. Alocarea dinamică a memoriei
 - 3.1. Operatorul **new**
 - 3.2. Operatorul **delete**
4. Structuri implementate dinamic:
 - 4.1. **Stiva**
 - 4.2. **Coadă**
 - 4.3. **Lista simplu înlănțuită**
 - 4.4. **Lista dublu înlănțuită**

3.2. Operatorul delete

- ✓ O zonă de memorie alocată prin operatorul ***new*** se eliberează prin operatorul ***delete***.
- ✓ Dacă *p* este un pointer spre *tip*:

tip *p;

și

p=new tip;

- ✓ atunci zona din memoria heap alocată cu ajutorul lui ***new*** se eliberează folosind construcția:

delete p

3.2. Operatorul delete

- De asemenea, operatorul ***delete*** se utilizează pentru a dezaloca tablourile alocate prin ***new***.
- Fie alocarea:

```
tip *p=new tip[expresie];
```

- Această zonă se eliberează folosind o construcție de forma:

```
delete[expresie] p;
```


Conținutul cursului

1. Pointeri la funcții
2. Folosirea pointerilor și tablourilor ca parametri în funcții
3. Alocarea dinamică a memoriei
 - 3.1. Operatorul **new**
 - 3.2. Operatorul **delete**
4. **Structuri implementate dinamic:**
 - 4.1. **Stiva**
 - 4.2. **Coadă**
 - 4.3. **Lista simplu înlănțuită**
 - 4.4. **Lista dublu înlănțuită**

4.1. Stiva

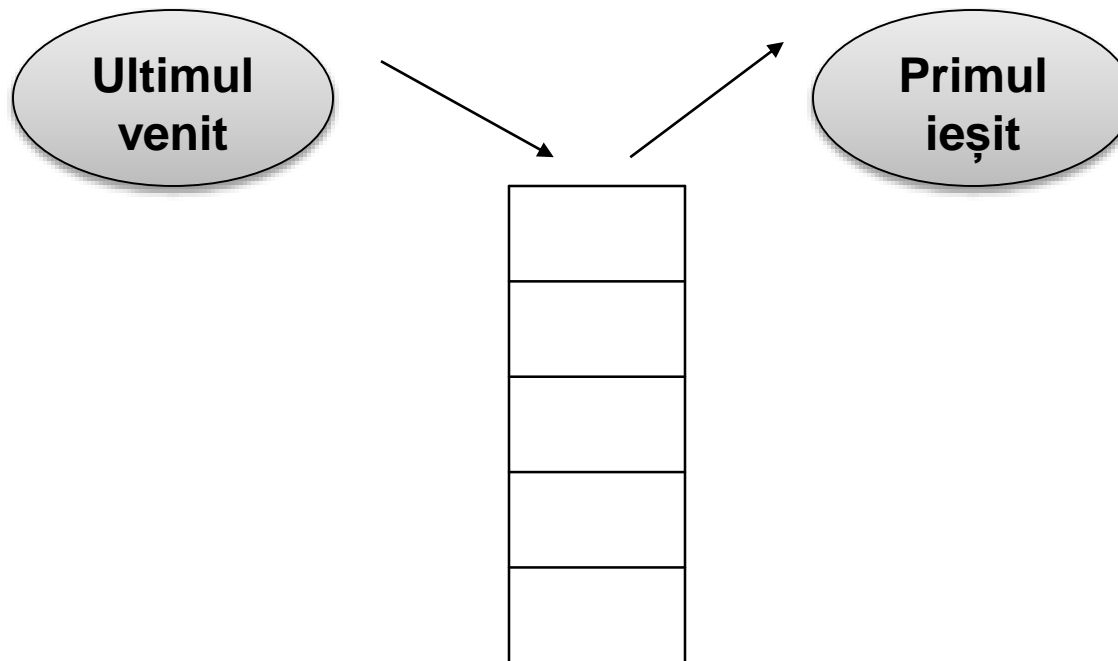
O stivă este un tip de dată ale cărei operații de inserare și de ștergere păstrează această regulă:

ultimul - venit - primul - ieșit
LIFO (Last-In-First-Out)

- ✓ Astfel, într-o stivă pot fi adăugate sau șterse elemente doar în vârful stivei.

4.1. Stiva

Grafic, o stivă se poate reprezenta astfel:



4.1. Stiva

- ✓ Pentru a putea realiza implementarea dinamică a stivei, elementele unei astfel de structuri vor fi de tip **structură**, cu două feluri de informații:
 - ✓ **informația propriu-zisă** (conținutul efectiv al elementului respectiv și care variază în funcție de problemă – notată cu *inf* care este de un anumit tip de date *tip*)
 - ✓ **informația de legătură** (care este un pointer ce conține adresa elementului precedent din stivă – notat cu *leg*).

4.1. Stiva

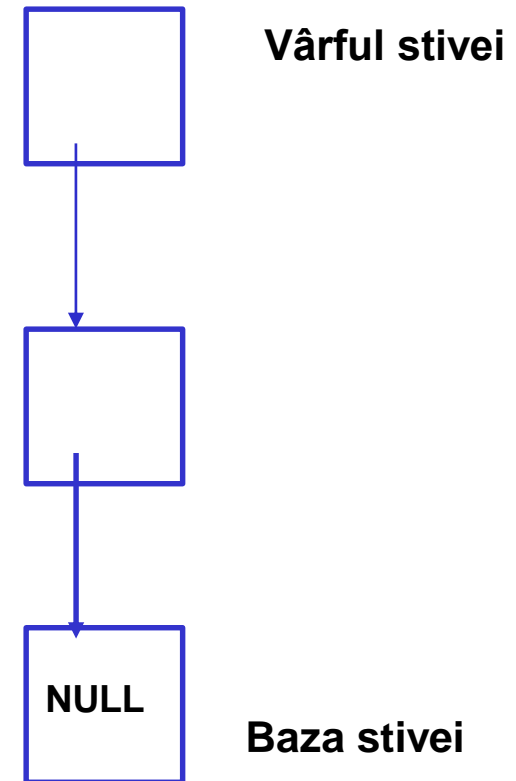
Tipul de date necesar implementării dinamice a structurii de tip stivă este:

```
typedef struct tnod
{
    tip inf;           // informatia propriu-zisa
    struct tnod *leg; // informatia de legatura
} TNOD;
```

4.1. Stiva

- ✓ Adăugarea sau extragerea unui element se face la un singur capăt numit **vârful stivei**.
- ✓ Elementul introdus primul în stivă se afla la **baza stivei**.
- ✓ Informația de legătură a fiecărui element din stivă reprezintă adresa elementului pus anterior în stivă, excepție făcând elementul de la bază, a cărui informație de legătură este **NULL**.

- ✓ Se observă că este **necesar și suficient** să fie reținută **adresa elementului din vârful stivei** într-un pointer pe care îl voi nota cu **$vf(TNOD *vf)$** , celelalte elemente putând fi accesate cu ajutorul informației de legătură de care dispune fiecare element al stivei.
- ✓ **Stiva** se poate reprezenta grafic astfel:



4.1. Stiva

Operațiile ce se pot efectua asupra stivei

1) **Inițializarea stivei:**

```
void init( TNOD* vf )  
{  
    vf=NULL;  
}
```


2) Adăugarea unui element x în vârful stivei:

Deoarece toate adăugările și ștergerile se fac la un singur capăt al stivei, singura posibilitate de a adăuga un element în stivă este în vârf.

Acest lucru se realizează astfel:

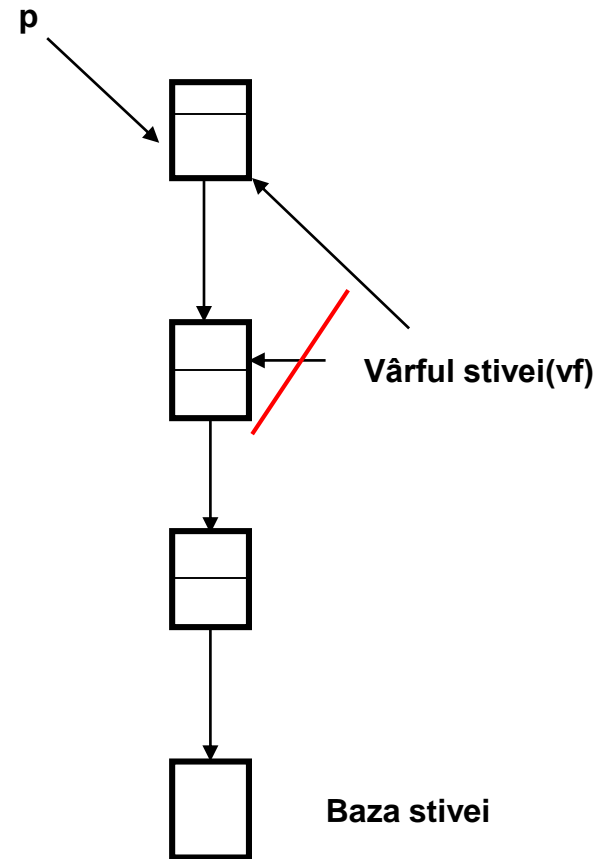
- Se alocă memorie pentru noul element (**a**);
- Se inițializează zona de informație propriu-zisă (utilă) a acestuia (**b**);
- Se inițializează informația de legătură cu adresa elementului care a fost anterior în vârful stivei, adresă păstrată în variabila **vf** (**c**);
- Se actualizează variabila **vf** cu adresa noului element alocat (**d**).
- Putem observa că acest algoritm este corect și dacă stiva era vidă înainte de adăugare(**vf=NULL**).

```

void adaug(tip x, TNOD *vf)
{
    TNOD *p;           //pointer de lucru
    p=new TNOD;       //(a)
    p->inf=x;          //(b)
    p->leg=vf;         //(c)
    vf=p;              //(d)
}

```

Această operație se poate reprezenta grafic astfel:



3) Extragerea elementului din vârful stivei

Prin extragerea unui element se înțelege eliminarea acestuia din stivă.

Această operație se poate realiza numai dacă **stiva este nevidă** și în caz afirmativ **se va șterge elementul din vârful stivei**.

Operația se efectuează astfel:

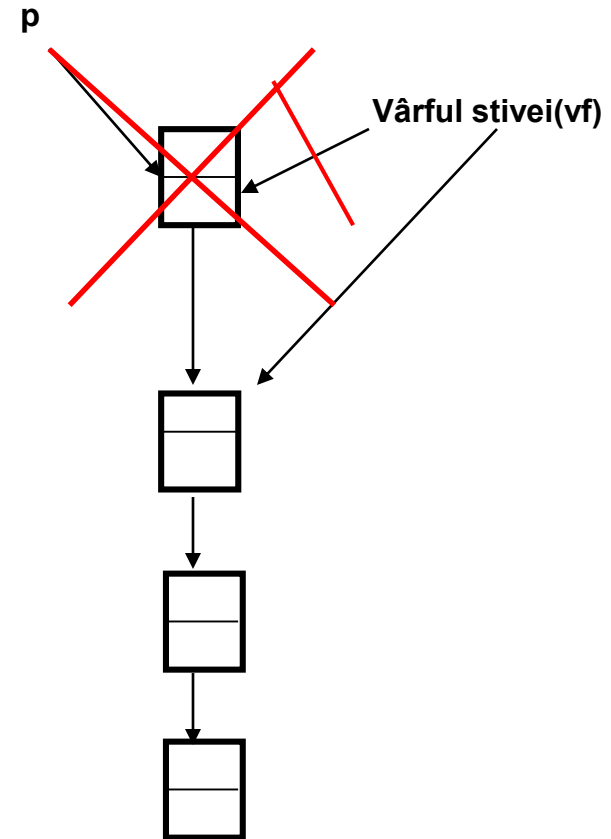
- Într-un pointer de lucru notat **p**, se reține adresa elementului din vârful stivei; **(a)**
- În variabila **q** se extrage elementul din vârful stivei;
- Adresa elementului următor celui din vârful stivei devine adresa noului vârf al stivei; **(b)**
- Se eliberează memoria ocupată de elementul care a fost anterior în vârful stivei. **(c)**

```

void extragere( TNOD *vf,
  tip *q)
{
  TNOD *p; //pointer de lucru
  p=vf;    //(a)
  *q=vf->inf; // Se extrage în
  variabila *q valoarea din
  varful stivei
  vf=vf->leg; //(b)
  delete p;  //(c)
}

```

Această operație se poate reprezenta grafic astfel:



4) Verificarea dacă stiva este vidă

```
int stvida( TNOD *vf )
{
    return vf==NULL;
}
```

5) Numărarea elementelor din stivă

```
int cardinal(TNOD *vf)
{
    int m=0; //contorizeaza
             elementele stivei
    TNOD* p; //pointer de lucru
    p=vf;
    while (p!=NULL)
    {
        m++;
        p=p->leg;
    }
    return m;
}
```

Întrebări?