

Laboratorul 1

POINTERI ȘI REFERINȚE	2
Tipul de date pointer	2
Tipul de date referință	4
ALOCARE DINAMICĂ	4
STRUCTURI DE DATE ALOCATE DINAMIC	5
Liste. Implementarea Dinamică a Listelor	5
Operații pe liste	6
Probleme	7

Bibliografie

Manualele de liceu clasa 11 intensiv

POINTERI ȘI REFERINȚE

Zone de memorie în care se alocă spațiu datelor, în funcție de modul în care au fost declarate în program.

Segmentul de date	← variabile globale / locale statice
Stiva sistemului (stack)	← variabile locale
Zona de adrese libere (heap)	← date alocate dinamic
Registrii procesorului	


O variabilă se caracterizează prin:

1. **nume**
2. **tip de date** (stabilește numărul de octeți pe care se va memora variabila + modul în care este memorată <=> valoarea minimă și maximă a variabilei)
3. **adresă de memorie** (adresa de început a zonei de memorie alocată variabilei respective)
4. **valoarea** la un moment dat (conținutul zonei de memorie rezervate datei)

Exemplu:

```
int x = 1234;  
cout<<x<<" "<<sizeof(int)<<" "<<endl;  
cout<<&x<<endl;
```

	int => sizeof(int) = 4 octeți				
...	00000000	00000000	00000100	11010010	...
	x (numele variabilei)				



&x => 0x96fadffdb4 (adresa octetului în hexa)

Pentru a putea utiliza o dată, programul trebuie să poată identifica zona de memorie în care este stocată. Aceasta se poate identifica prin **numele** datei sau prin **adresa** la care este memorată. Pentru a putea **manipula datele cu ajutorul adreselor de memorie** în limbajul C++ sunt implementate tipul **pointer (adresă)** și tipul **referință**.

Tipul de date pointer

Pointer = variabilă de memorie în care se memorează o adresă de memorie

Se asociază întotdeauna unui tip de dată (numit tip de bază), pentru a putea ști cum se interpretează conținutul de la acea adresă de memorie (care este dimensiunea zonei, ce algoritm de decodificare trebuie folosit)

Declarare

```
<tip_data> *nume_variabila;
```

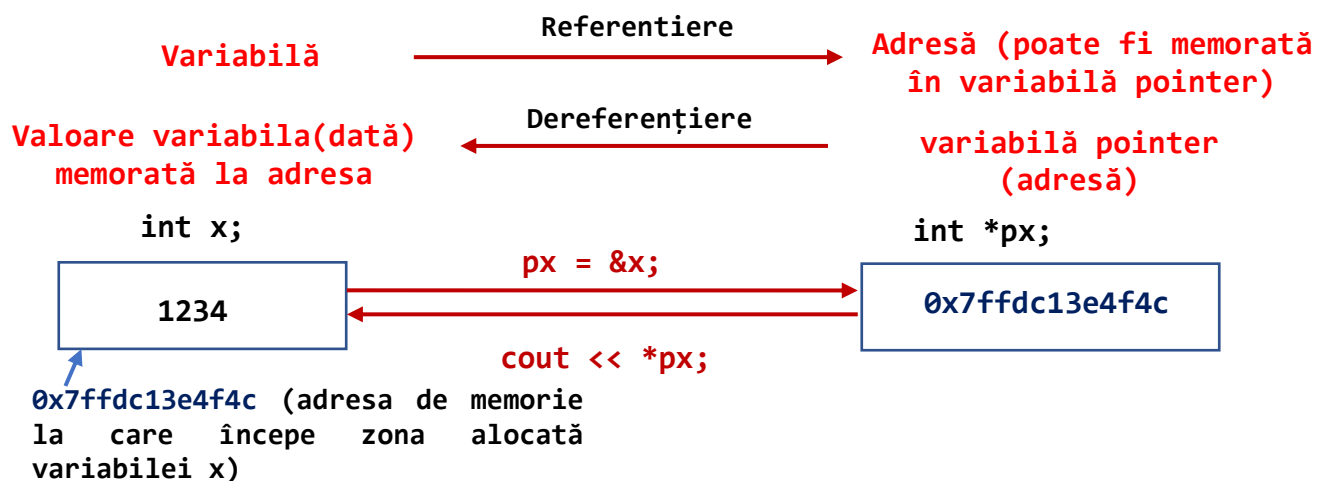
Pointer 0 / adresa 0/ NULL = adresa inexistentă - folosit ca marcaj de terminare în structurile de date care implementate cu ajutorul pointerilor

Operatori SPECIFICI

- **Operatorul de referențiere & (operatorul de adresare):**
&variabilă => adresa variabilei respective
- **Operatorul de dereferențiere * (operatorul de redirectare):**
***pointer => valoarea** aflată la adresa memorată în pointer; rezultatul obținut este de tipul datei asociate pointerului

Exemplu

```
int x = 123;  
int *px;  
px = &x;  
cout<<x<<" "<<px<<endl;  
cout<<*px<<endl;  
*px = *px + 2;  
cout<<x<<" "<<*px<<endl;  
x = x + 4;  
cout<<x<<" "<<*px<<endl;
```



Dacă tipul de bază al unui pointer **p** este **struct** (tipul înregistrare), data conține mai multe câmpuri. Pentru a accesa un câmp al structurii aflate la adresa memorată de **p** putem folosi operatorul de dereferențiere *****, apoi **.** pentru a accesa câmpul:

(*p).camp

sau putem folosi operatorul **->**

p->camp

Exemplu

```
struct interval{  
    int x,y;  
};  
interval i={4,5};  
cout<<i.x<<" "<<i.y<<endl;  
interval *pi;  
pi=&i;  
cout<<(*pi).x<<" "<<(*pi).y<<endl;  
cout<<pi->x<<" "<<pi->y<<endl;
```

Tipul de date referință

Permite **folosirea mai multor identificatori pentru aceeași variabilă**.

Declarare:

```
<tip_data> &nume_variabila = nume_variabilă_referita;
```

- Referința trebuie inițializată **în momentul declarării**, după aceea nu mai poate fi modificată.
- Tipul referință conține tot o adresă, dar pentru a avea acces la variabilă se folosește direct numele variabilei referință. Nu se poate crea un pointer la o referință
- Operatorii aritmetici, relaționali – lucrează pe valoarea memorată la adresă (nu cu adresa)

Exemplu

```
int u=8, &w=u;  
cout<<u<<" "<<w<<endl;  
u++;  
cout<<u<<" "<<w<<endl;  
w++;  
cout<<u<<" "<<w<<endl;  
cout<<(&u)<<" "<<(&w)<<endl;
```

ALOCARE DINAMICĂ

Alocarea statică

- în timpul compilării
- nu mai poate fi modificată în timpul execuției
- dezavantaj – se poate să nu fie spațiu suficient, pot ocupa mai mult spațiu decât necesar (se alocă de dimensiune maximă)

Alocarea dinamică

- alocare sau eliberarea de spațiu de memorie pentru o variabilă în timpul execuției programului
- spațiul de memorie va fi alocat în HEAP (zona de adrese libere)

Lucru cu variabile dinamice

• Declarare

```
<tip_de_baza> *nume_pointer;
```

• Alocare cu operatorul new

```
nume_pointer = new tip_de_baza;
```

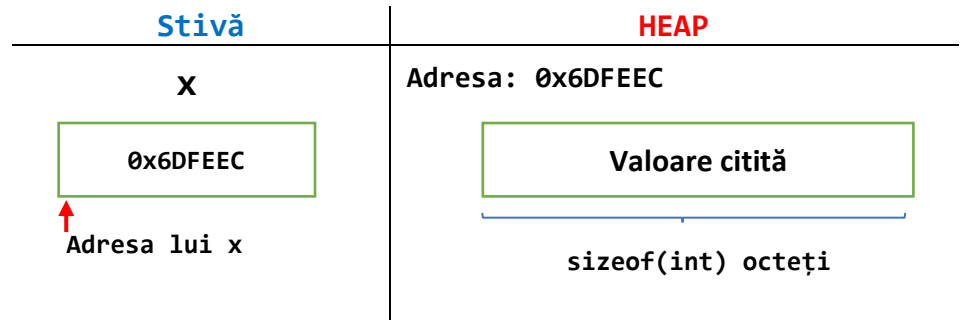
Se caută în heap o zonă liberă, de dimensiunea tipului de bază și se atribuie pointerului adresa acelei zone

• Eliberarea spațiului de memorie

```
delete nume_pointer;
```

Exemplu

```
int *x;
x = new int;
cin>>(*x);
cout<<*x<<endl;
cout<< &x<<endl;
delete x;
cout<< &x<<endl;
//cout<<*x<<endl;
```



STRUCTURI DE DATE ALOCATE DINAMIC

Liste. Implementarea Dinamică a Listelor

Lista liniară = structură de date cu date **omogene** (de același tip), în care fiecare element are un succesor și un predecesor (exceptând primul element care are doar predecesor, și ultimul care are doar succesor)

Diferențe față de vector:

Vectori- structuri deja **implementate, contigue** (elementele ocupă zone contigue de memorie, putem accesa un element folosind operații aritmetice cu adrese, deplasându-ne față de adresa de început: $v+2$), **statice**

Liste – structuri **explicite, dispersate** (elementele sunt dispuse în zone dispersate de memorie elementele sunt dispuse în zone dispersate de memorie; pentru a putea localiza elementele în structură trebuie să memorăm pentru fiecare element și adresa la care se găsește), **statice** sau **dinamice** în funcție de implementare

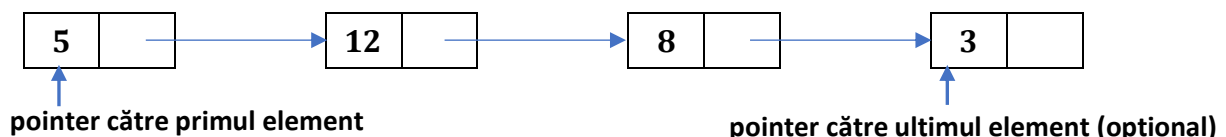
Posibile dezavantaje vector => când utilizăm liste:

- Vectorul are lungime fizică fixă - diferența între lungimea maximă și cea efectivă poate fi foarte mare
- O operație de adăugare sau ștergere a unui element necesită deplasări

Implementarea dinamică a listelor

Elementele (nodurile) listei nu ocupă zone consecutive în memorie, de aceea pentru fiecare nod din listă trebuie să știm adresa succesorului. Astfel, un nod al listei conține atât informația asociată lui, cât și adresa succesorului lui, deci va fi de tip **struct** (înregistrare).

Trebuie memorată în plus adresa primului element, iar dacă este util pentru eficiența operațiilor, putem memora și adresa ultimului element.



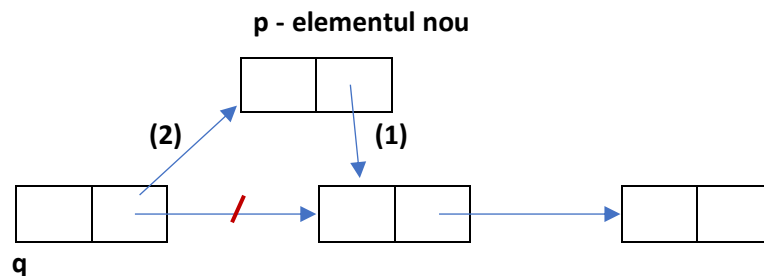
```
struct node{
    int info;
    node* next;
};
node *head, *last;
```

Inițializare: `head = NULL; last = NULL;`

Alocarea unui nod nou cu informația **x**:

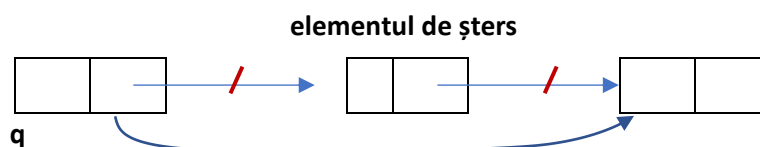
```
p = new node;  
p->info = x;  
p->next = NULL;
```

Adăugare – schema



```
p->next = q->next; //(1)  
q->next = p; //(2)  
//contează ordinea atribuirilor
```

Ștergere – schema



```
p = q-> next;  
q->next = p->next;  
delete p; // !!!! trebuie
```

Atenție - când lista are un nou prim element după actualizare, trebuie actualizat **head** (similar pentru **last**)

Operații pe liste

Creați o listă cu n numere citite de la tastatură.

Implementați următoarele operații:

- Parcurgere
- Adăugare la început
- Adăugare la final
- Adăugare pe poziție dată
- Căutarea unei valori date
- Ștergerea primului element
- Ștergerea ultimului element
- Ștergerea unui element de pe poziție dată
- Ștergerea primei apariții a unei valori date în listă
- Ștergerea unui element dat prin adresa sa
- ...

Probleme

1. Sortare prin inserție – Se citesc n numere de la tastatură. Să se insereze pe rând aceste numere într-o listă astfel încât după fiecare inserare elementele listei să fie în ordine crescătoare <https://leetcode.com/problems/insertion-sort-list/description/>
2. Adunarea a două numere mari <https://leetcode.com/problems/add-two-numbers/> - citite fiecare dintr-un fișier; un număr se va memora ca listă, primul element din listă fiind cifra unităților.
3. Inversarea legăturilor într-o listă <https://leetcode.com/problems/reverse-linked-list/description/>
4. Ștergerea duplicatelor dintr-o listă ordonată:
<https://leetcode.com/problems/remove-duplicates-from-sorted-list/description/>
5. <https://leetcode.com/problems/middle-of-the-linked-list/>
6. <https://leetcode.com/problems/delete-the-middle-node-of-a-linked-list/>