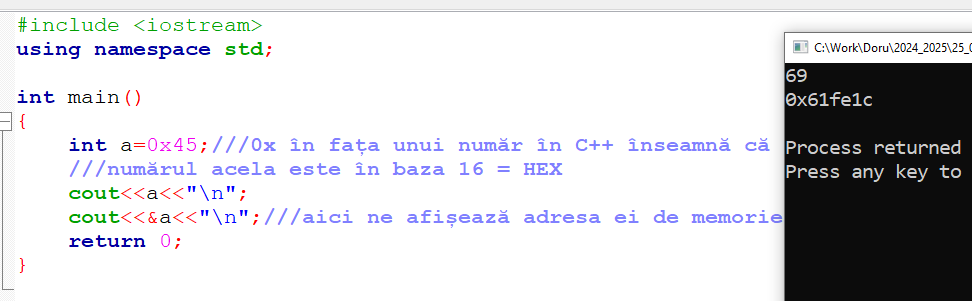
**Variabile statice și variabile dinamice (pointeri)**

Toate variabilele cu care am lucrat până acum sunt variabile statice.

Ele au o adresă de memorie FIXĂ de la lansarea în execuție a programului până la terminarea sa.

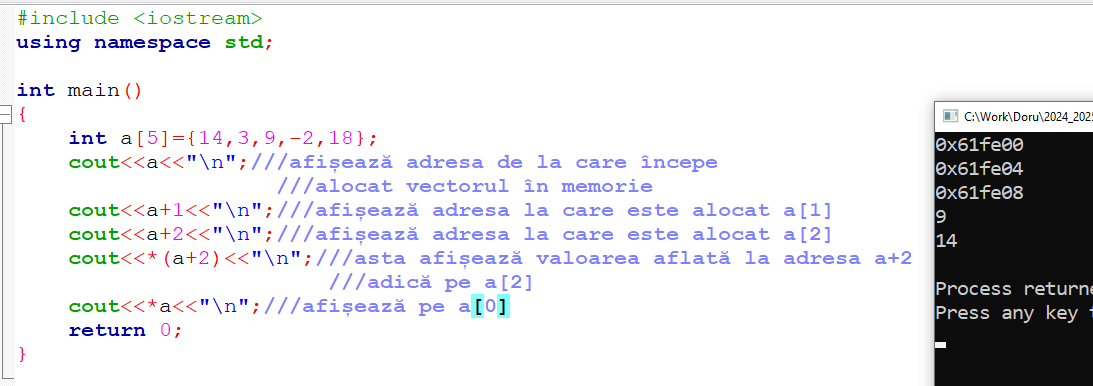
Această adresă este neinteresantă d.p.d.v. al utilizatorului (ea fiind un număr de ordine al octetului care găzduiește începutul reprezentării variabilei) dar totuși poate fi obținută cu operatorul &.

Chiar și vectorii sunt de asemenea statici.

În cazul lor, numele variabilei FĂRĂ paranteze drept reprezintă ADRESA de memorie de la care începe să fie alocat vectorul.

Adunările cu o constantă **x** la această adresă înseamnă adresa de memorie a elementului de la indicele **x**.

În cazul vectorilor - dacă lucrăm cu adrese de memorie, operatorul \* ne dă de fapt valoarea care se află la adresa respectivă.



De reținut așadar:

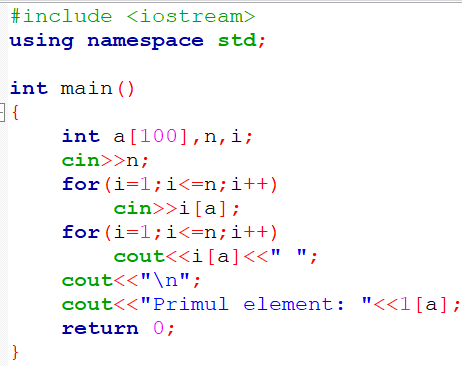
Dacă a este un vector, expresia **\*a** <=> **a[0]**

expresia **\*(a+x)** <=> **a[x]**

Funny fact:

As we saw: **a[x]=\*(a+x)**

adunarea este comutativă deci putem continua **=\*(x+a)=x[a]**



**Pointeri (sau variabile dinamice)**

Sunt variabilele care se declară cu \*.

Ex:

**int \*a;**

Imediat după declarare, aceste variabile sunt homeless - NU au alocată NICIO adresă de memorie.

Putem face două lucruri cu un pointer nealocat:

1) îi alocăm memorie prin:

**a=new int;** ///exemplul este pentru o simplă variabilă de tip int

sau

**a=new int[nr\_elemente];**///așa se alocă un vector cu **nr\_elemente**

În primul caz (variabila simplă), adresarea variabilei a se poate face prin:

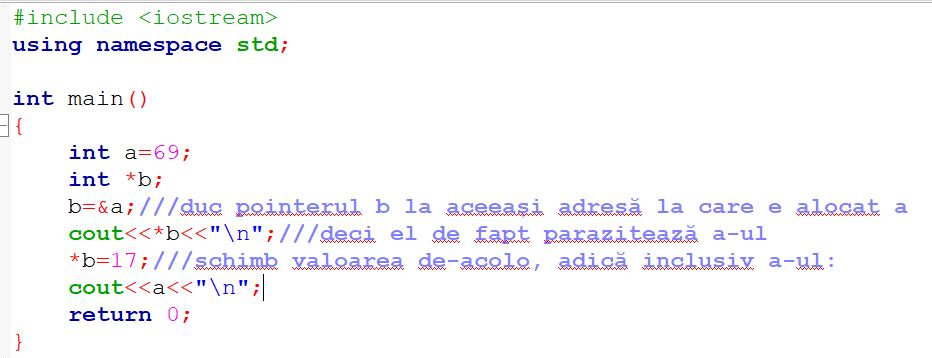
a = adresa de memorie

\*a = valoarea aflată la acea adresă de memorie

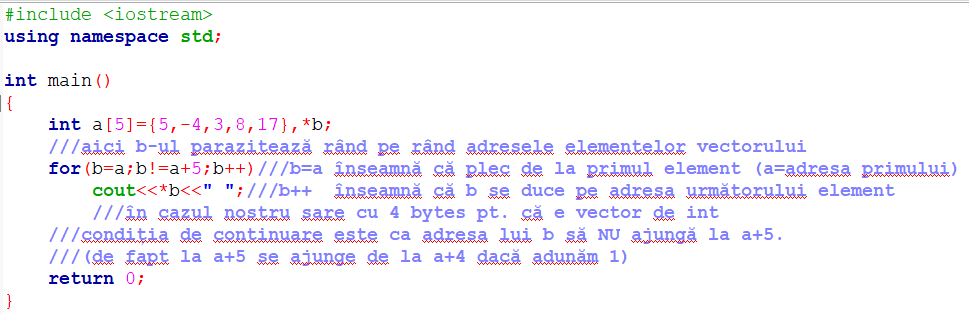
În al doilea caz (vectorul) lucrăm cu a exact așa cum știam deja de la vectori, adică putem folosi a[indice]

2) Îl ducem să "paraziteze" o altă variabilă (o adresă de memorie):

- în exemplul de mai jos parazitează o singură variabilă de tip int



- în exemplul următor mă folosesc de un pointer parazit ca să afișez elementele unui vector:



Există o constantă specială numită NULL, care poate fi atribuită unui pointer.

Rolul lui NULL este în special de a ajuta programatorul în a identifica diverse situații speciale fără a mai folosi un flag.

**Pointeri către șiruri de caractere**

În cazul șirurilor de caractere pointerii lucrează ceva mai special.

Așa cum am văzut, dacă declarăm un vector

**char s[20]="televizor";**

Prin **s** se înțelege adresa de memorie la care este alocat șirul de caractere (adresa primului caracter din acesta).

Șirurile de caractere au proprietate că, atunci când dăm

**cout<<adresă\_șir\_de\_char;** (gen **cout<<s;**) nu se mai afișează, ca la vectorii de int, adresa de memorie ci caracterele din șir. Mai precis, ia caracter cu caracter de la adresa respectivă până dă de marca de final de string, adică acel caracter de cod ASCII 0 (caracterul nul).

De fapt, **cout<<s;** este perfect echivalent cu următorul cod (de fapt asta și face)

(char \*p este un pointer de tip char, pe for este inițializat cu adresa lui s. Merg cu p până ajung la caracterul de la adresa respectivă care are codul ASCII 0, )

**for(char \*p=s;\*p;p++)**

**cout<<\*p;**

Așa cum am văzut, adunând la un pointer o constantă este ca și cum am obține elementul de la adresa dată de constantă.

Astfel,

dacă avem

char s[20]="televizor";

s+1 = adresa lui s[1] adică a lui 'e'

am văzut că, în cazul stringurilor, accesând o adresă de memorie, de fapt obține stringul care începe de la acea adresă.

cout<<s+1; //afișează elevizor

cout<<s+5; //afișează vizor

Mai mult, această regulă funcționează chiar și în cazul constantelor de tip șir de caractere:

cout<<"BOGDAN"+3; ->afișează DAN