

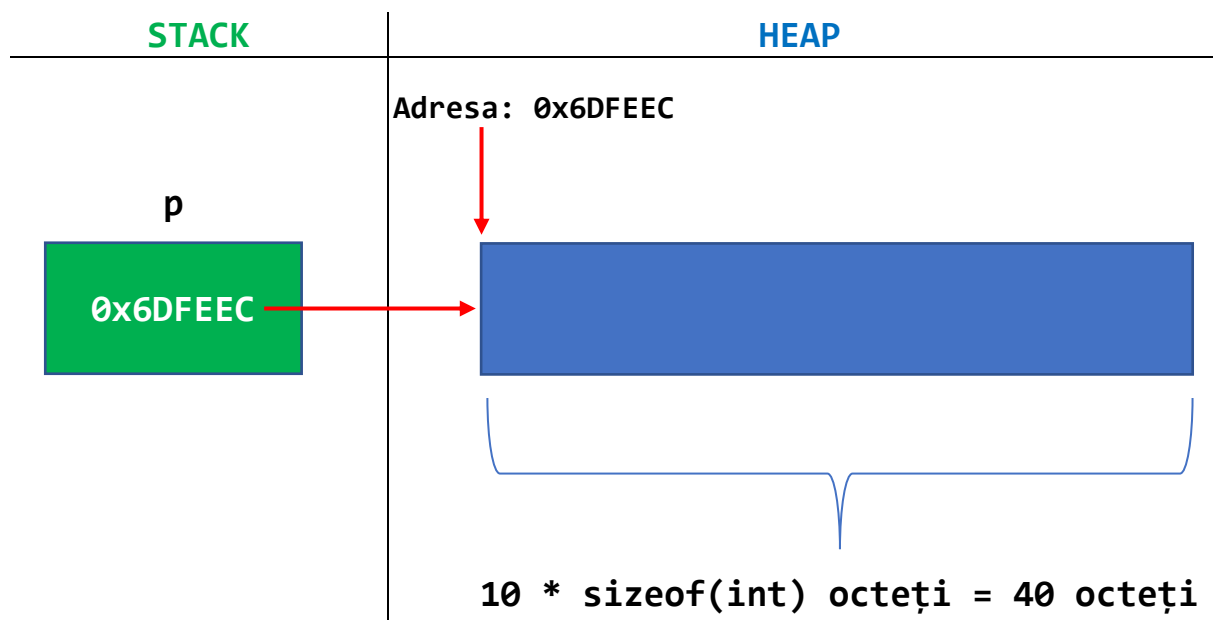
# CURS 01 – PP

## ALOCAREA DINAMICĂ A MEMORIEI

Zona STACK (stivă)	Zona HEAP
<ul style="list-style-type: none"><li>• se memorează variabilele locale (management automat)</li><li>• se execută apelurile funcțiilor</li></ul>	<ul style="list-style-type: none"><li>• se memorează variabilele alocate dinamic (management manual)</li></ul>

**Exemplu:**

```
int *p, n = 10;  
p = (int*)malloc(n * sizeof(int));
```



## Funcții pentru alocarea dinamică a memoriei (stdlib.h):

### 1) void\* malloc(int dimensiune\_zona\_de\_memorie\_în\_octeți)

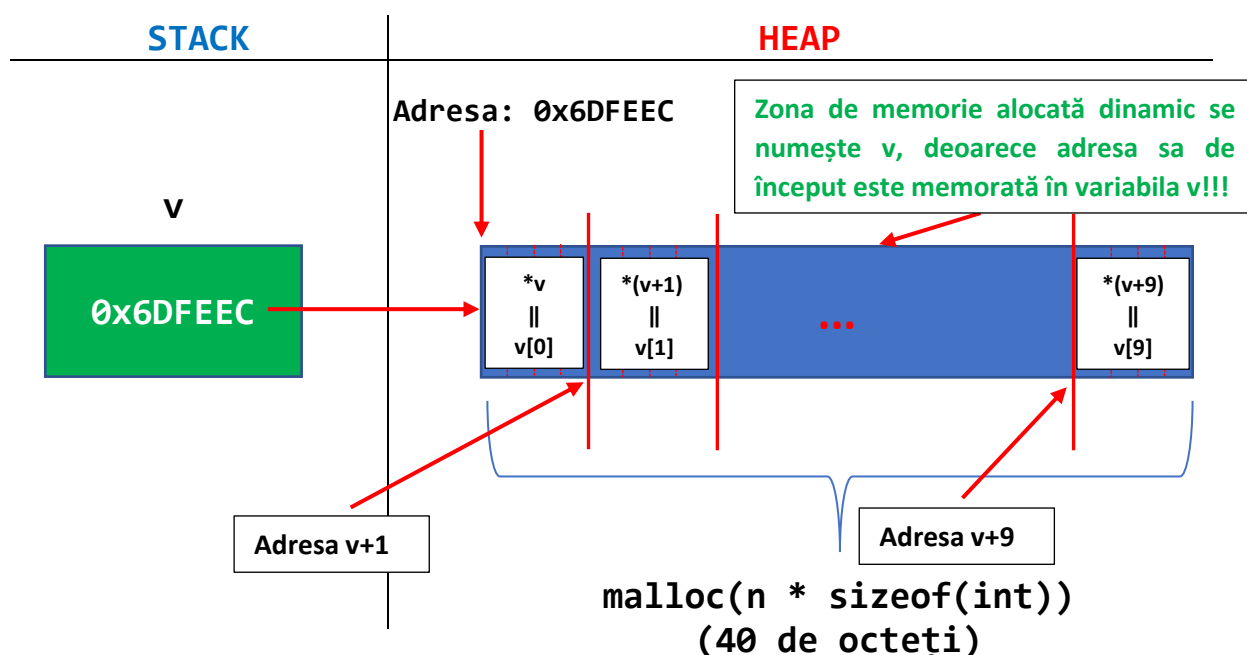
**malloc** = *memory allocation*

Funcția malloc furnizează un *pointer generic* (de tip void\*) care conține adresa de început a zonei de memorie alocată dinamic sau pointerul NULL dacă nu există suficientă memorie disponibilă.

Un *pointer generic* poate să conțină adrese de orice tip fără conversii explicite la atribuire! Acest lucru este posibil deoarece dimensiunea unui pointer este constantă (de obicei, 4 octeți), indiferent de tipul de bază al pointerului respectiv.

#### Exemplu:

```
int *v, n = 10;  
v = (int*)malloc(n * sizeof(int));
```



Datorită faptului că tipul de bază al pointerului `v` este `int` => aritmetica pointerilor se va realiza cu pasul de 1 `int` = 4 octeți => zona de memorie continuă poate fi accesată ca un tablou unidimensional folosind aritmetica pointerilor!

$$v[i] == *(v+i) \Leftrightarrow *(v+i) == v[i]$$

Numele unui tablou este adresa primului său element:

$v == \&v[0]$

Adresa primului element al unui tablou este chiar numele său:

$\&v[0] == v$

## 2) void\* calloc(int nr\_blocuri\_de\_memorie, int dimensiune\_bloc\_în\_octeți)

Funcția calloc furnizează un *pointer generic* (de tip void\*) care conține adresa de început a unei zone de memorie continuă alocată dinamic, formată din numărul de blocuri indicate și **având toți octeții inițializați cu valoarea 0**, sau pointerul NULL dacă nu există suficientă memorie disponibilă.

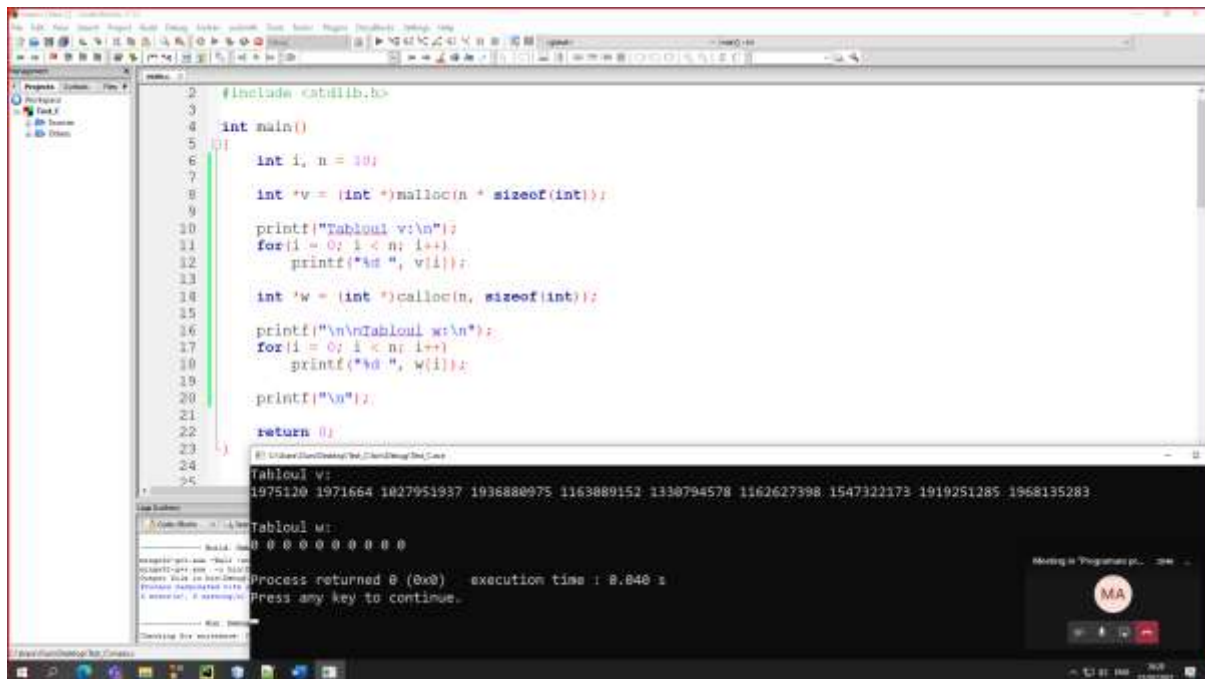
**calloc** = *clear (memory) allocation*

```
int dim = sizeof(int);
```

```
int *v = (int *)calloc(n, dim);
```

**SAU**

```
int *v = (int *)malloc(n*dim)
for(i = 0; i < n; i++)
    v[i] = 0;
```



```
#include <stdlib.h>

int main()
{
    int i, n = 10;

    int *v = (int *)malloc(n * sizeof(int));

    printf("Tabloul v:\n");
    for(i = 0; i < n; i++)
        printf("%d ", v[i]);

    int *w = (int *)calloc(n, sizeof(int));

    printf("\n\nTabloul w:\n");
    for(i = 0; i < n; i++)
        printf("%d ", w[i]);

    printf("\n\n");

    return 0;
}
```

Tabloul v:  
1975120 1971664 1027951937 1936880975 1163889152 1330794578 1162627398 1547322173 1919251285 1968135283

Tabloul w:  
0 0 0 0 0 0 0 0 0 0

Process returned 0 (0x0) execution time : 0.848 s  
Press any key to continue.

### 3) void\* realloc(void\* *adresă\_zonă\_memorie*, int *dimensiune\_nouă\_octeți*)

Funcția `realloc` încearcă să redimensioneze o zonă de memorie alocată dinamic astfel:

- dacă noua dimensiune a zonei de memorie este mai mică sau egală decât dimensiunea sa curentă, atunci se micșorează pur și simplu dimensiunea zonei respective (în tabelul de alocare al sistemului de operare) și adresa de început a zonei de memorie rămâne nemodificată;
- dacă noua dimensiune a zonei de memorie este strict mai mare decât dimensiunea sa curentă, atunci:
  - 1) încearcă să extindă zona curentă de memorie spre dreapta și, dacă acest lucru este posibil, atunci pur și simplu se mărește dimensiunea zonei respective (în tabelul de alocare al sistemului de operare) și adresa de început a zonei de memorie (întoarsă de funcție) va rămâne nemodificată;
  - 2) caută o zonă de memorie continuă de noua dimensiune dorită și, dacă o găsește, copiază conținutul zonei de memorie inițiale în noua zonă de memorie și apoi o eliberează (evident, adresa de început a zonei de memorie se va modifica), după care returnează adresa noii zone de memorie;
  - 3) dacă nu reușește în niciuna dintre variantele 1) și 2), atunci funcția va întoarce pointerul `NULL` (memorie insuficientă), **FĂRĂ A ELIBERA ZONA DE MEMORIE CURENTĂ**.

Dacă primul parametru al funcției este pointerul `NULL`, atunci funcția `realloc` va alocă o zonă de memorie cu dimensiunea indicată, la fel ca funcția `malloc`.

Dacă al doilea parametru al funcției este 0, atunci funcția `realloc` **S-AR PUTEA** să elibereze zona de memorie respectivă, la fel ca funcția `free`.

**Zona orfană de memorie (memory leak)** = zonă de memorie, de obicei alocată dinamic, a cărei adresă de început nu mai este cunoscută, deci ea nu mai poate fi eliberată și va rămâne alocată până în momentul repornirii calculatorului!!!

**Exemplu:** Se citește un șir de numere întregi terminat cu valoarea 0 (care se consideră că NU face parte din șir). Să se memoreze numerele citite într-un tablou unidimensional alocat dinamic.

```

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

int main()
{
    //x = valoarea citita
    //n = numarul de valori citite
    int i, x, n;
    //v = tabloul curent
    //aux = un pointer auxiliar
    int *v, *aux;

    //pentru ca realloc sa aloce o zona de memorie pentru prima valoare
    v = NULL;
    //initial, numarul valorilor citite este 0
    n = 0;

    do
    {
        //se citeste o noua valoare
        printf("Valoare: ");
        scanf("%d", &x);

        //daca valoarea citita nu este marcajul de final al sirului
        if(x != 0)
        {
            //crestem numarul valorilor citite cu 1
            n++;
            //incercam sa realocam tabloul v
            aux = (int *)realloc(v, n * sizeof(int));

            //daca nu exista suficienta memorie libera
            //pentru a realoca tabloul v
            if(aux == NULL)
            {
                printf("\nMemorie insuficienta!!!\n");

                //eliberam zona de memorie a tabloului v
                free(v);

                //intrerupem executarea programului (eroare fatala)
                exit(0);
            }

            //tabloul v a putut fi realocat folosind pointerul aux,
            //deci "mutam" adresa aux in v
            v = aux;

            //NU are sens sa eliberam zona de memorie a carei

```

```

        //adresa de inceput se gaseste in aux,
        //deoarece am "sterge" si tabloul v
        //(vezi desenul de mai jos)!!!
        //free(aux);

        //adaugam valoarea x citita la sfarsitul tabloului v
        v[n-1] = x;
    }
}
while(x != 0);

printf("\nValorile citite:\n");

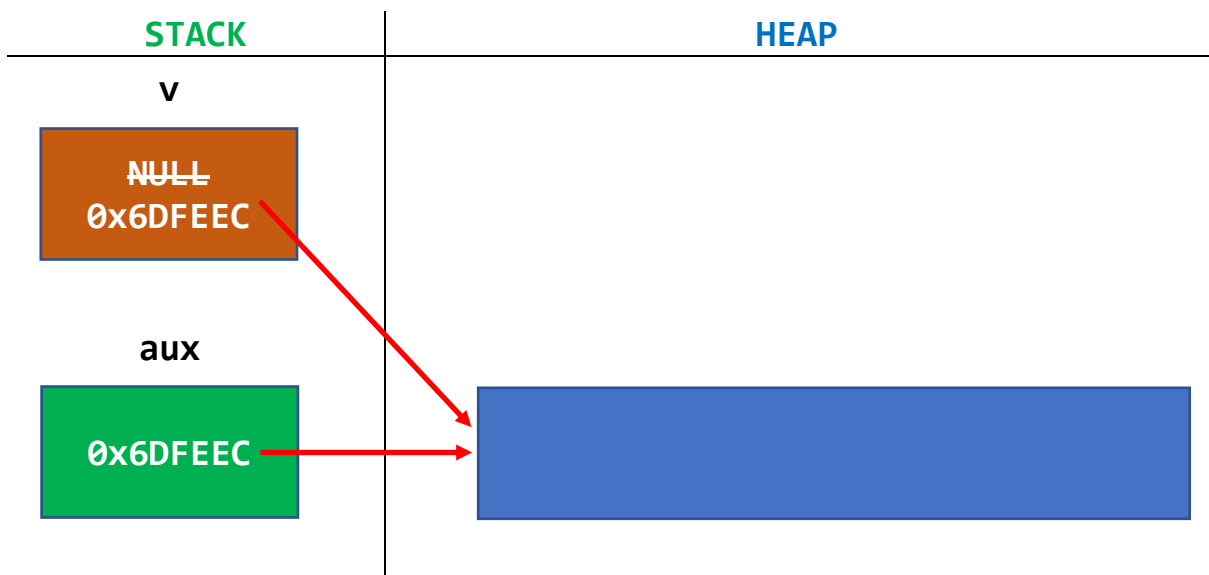
//prelucrarea tabloului v (o simpla afisare, in acest caz)
for(i = 0; i < n; i++)
    printf("%d ", v[i]);

printf("\n");

//eliberam zona de memorie alocata tabloului v
free(v);

return 0;
}

```



**Observație:** În mod normal, nu se realocă tabloul v pentru fiecare nouă valoare citită, ci se folosește un buffer (de obicei, un tablou alocat static de dimensiune în jur de 10-25% din dimensiunea totală a datelor care vor fi citite), iar realocarea tabloului se realizează după umplerea buffer-ului!!!

#### 4) void free(void\* adresa\_de\_început\_a\_zonei\_de\_memorie)

Funcția free eliberează zona de memorie alocată dinamic a cărei adresă de început este dată ca parametru.

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

int main()
{
    int i, n, *v;

    printf("n = ");
    scanf("%d", &n);

    v = (int *)malloc(n * sizeof(int));
    //SAU
    //v = malloc(n * sizeof(int));

    if(v == NULL)
    {
        printf("Memorie insuficienta!");
        exit(0);
    }

    for(i = 0; i < n; i++)
        v[i] = i+1;

    printf("Tabloul v:\n");
    for(i = 0; i < n; i++)
        printf("%d ", v[i]);

    free(v);

    return 0;
}
```

**Observație:** Funcția free NU modifică adresa din pointerul dat ca parametru (de exemplu, nu îi atribuie pointerul NULL), ci doar semnalează sistemului de operare faptul că zona respectivă de memorie poate fi eliberată!

**Exemplu:** Funcție care furnizează un tablou unidimensional format din numerele  $1, 2, \dots, n$ .

**Observație:** O funcție NU poate să întoarcă efectiv un tablou, dar poate să întoarcă adresa de început a tabloului (pointerii sunt tipuri de date simple)!

### Varianta 1 (incorectă):

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

int* tablou(int n)
{
    int aux[100], i;

    for(i = 0; i < n; i++)
        aux[i] = i + 1;

    return aux;
}

void afisare(int v[], int n)
{
    int i;

    printf("\nTabloul:\n");
    for(i = 0; i < n; i++)
        printf("%d ", v[i]);
    printf("\n");
}

int main()
{
    int *t;
    int n = 10;

    t = tablou(n);
    afisare(t, n);

    return 0;
}
```

Tabloul **aux** este local funcției **tablou**, deci va fi **creat automat** în zona de memorie STACK când se va începe executarea funcției și va fi **distrus automat** când se va încheia executarea sa.

Tabloul **t** va conține o **adresă de memorie invalidă**, adică o adresă unde nu mai există tabloul **aux** creat în funcția **tablou**!!!

### Varianta 2 (corectă):

```
int* tablou(int n)
{
    static int aux[100];
    int i;

    for(i = 0; i < n; i++)
        aux[i] = i + 1;

    return aux;
}
```

Tabloul **aux** este **static**, deci își va păstra adresa de început de-a lungul executării întregului program!!!



### Varianta 3 (corectă):

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

```
int* tablou(int n)
{
    int *aux;
    int i;

    aux = (int*)malloc(n * sizeof(int));

    for(i = 0; i < n; i++)
        aux[i] = i + 1;

    return aux;
}
```

```
void afisare(int v[], int n)
{
    int i;

    printf("\nTabloul:\n");
    for(i = 0; i < n; i++)
        printf("%d ", v[i]);
    printf("\n");
}
```

```
int main()
{
    int *t;
    int n = 10;

    t = tablou(n);
    afisare(t, n);

    free(t);

    return 0;
}
```

Variabila **aux** este locală funcției **tablou**, deci va fi **creată automat** în zona de memorie STACK când se va începe executarea funcției și va fi **distrusă automat** când aceasta se va încheia.

**Adresa memorată în variabila aux** este adresa unei zone de memorie din zona HEAP, deci zona respectivă NU va fi distrusă când se termină executarea funcției!

Variabila **t** va conține o **adresă de memorie validă** din zona HEAP, adică o adresă unde există tabloul **aux** creat în funcția **tablou**!!!

Eliberăm explicit zona de memorie alocată pentru tabloul a cărui adresă de început este memorată în variabila **t**!

### Varianta 4 (incorectă):

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

```
void tablou(int n, int *t)
{
    int i;

    t = (int*)malloc(n * sizeof(int));

    for(i = 0; i < n; i++)
        t[i] = i + 1;
}
```

```
void afisare(int v[], int n)
{
    int i;

    printf("\nTabloul:\n");
    for(i = 0; i < n; i++)
        printf("%d ", v[i]);
    printf("\n");
}
```

```
int main()
{
    int *t;
    int n = 10;

    tablou(n, t);
    afisare(t, n);

    free(t);

    return 0;
}
```

Pointerul **t** este transmis implicit prin valoare, adică se transmite o copie a adresei memorate în **t**.

Adresa de început a zonei de memorie alocată dinamic în zona HEAP va fi memorată în COPIA adresei din **t**, ci nu direct în variabila **t**!

Valoarea din pointerul **t** va rămâne neschimbată după apelarea funcției **tablou**!

### Varianta 5 (corectă):

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

void tablou(int n, int **t)
{
    int i;

    *t = (int*)malloc(n * sizeof(int));

    for(i = 0; i < n; i++)
        (*t)[i] = i + 1;
}

void afisare(int v[], int n)
{
    int i;

    printf("\nTabloul:\n");
    for(i = 0; i < n; i++)
        printf("%d ", v[i]);
    printf("\n");
}

int main()
{
    int *t;
    int n = 10;

    tablou(n, &t);
    afisare(t, n);

    free(t);

    return 0;
}
```

Pointerul **t** este transmis prin adresă.

Adresa de început a zonei de memorie alocată dinamic în zona HEAP va fi memorată direct în variabila **t**!

Pointerul **t** este transmis prin adresă.