

**Introducere**

Pentru toate tipurile de date descrise până acum, memoria necesară la execuția programului este determinată de declarațiile făcute pentru variabile, deci ea este stabilită înainte de execuția programului. Variabilele sunt alocate în memorie în momentul declarării lor și rămân alocate tot timpul execuției programului în cazul variabilelor globale sau statice sau, în cazul variabilelor automate, până la încheierea blocului în care au fost declarate.

Sunt situații în care, pentru o gestionare eficientă a memoriei, este de dorit ca în momentul în care unele variabile nu mai sunt necesare să fie eliminate din memorie. De asemenea, la scrierea programelor se dorește să se dea un grad de generalitate cât mai mare, astfel încât să poată fi prelucrate, după același algoritm, pachete de date de diferite dimensiuni. De exemplu, dacă într-un program este declarat un tablou de date, pentru el este stabilită dimensiunea printr-o constantă. Este posibil ca la execuția programului să fie necesar un număr mult mai mic de elemente, ceea ce înseamnă că o parte din memorie este alocată inutil. Dacă însă numărul de elemente necesar este mai mare decât dimensiunea stabilită, este necesară modificarea codului sursă și recompilarea lui. Astfel de situații pot fi evitate prin manevrarea în mod dinamic a variabilelor, adică, stabilirea dimensiunii memoriei necesare în momentul execuției programului, alocarea ei în momentul în care variabilele devin necesare și eliminarea lor când utilitatea lor a încetat. Memoria astfel eliberată va putea fi realocată în alt scop.

**9.1. Alocarea dinamică de memorie**

Pentru gestionarea memoriei, aceasta este împărțită în segmente specializate. Microprocesoarele dispun de regiștri în care se memorează adresele segmentelor. Se folosesc patru regiștri de segment specializați (CS, DS, ES, SS), ceea ce permite adresarea simultană a patru segmente folosindu-se adresa de segment din registrul asociat:

- segment de cod (instrucțiuni) - Code Segment (CS);
- segment de date – Data Segment (DS);
- segment de date suplimentar – ExtraSegment (ES) ;
- segment de stivă – Stack Segment (SS).

Având în vedere acest mod de gestionare a memoriei, programele C/C++, la execuție, încarcă în memorie 4 tipuri de informație:

- codul executabil – memorat în segmente de cod;
- date statice – memorate în segmente de date;
- date automate – alocate în pe stivă sau regiștri;
- date dinamice – alocate în zona heap.

*Se alocă static memorie* pentru constante, variabilele globale și variabilele locale declarate în mod explicit static.

*Se alocă memorie pe stivă* pentru variabilele locale.

```

int a, b;           // a, b, x - variabile declarate global, alocate în segmentul de date
double x;

double f1(int c, double v) // parametrii c, v – alocați pe stivă
{
    int b;           // b – variabilă locală funcției f1(), alocată pe stivă
    static double z;  // z – variabilă locală funcției declarată static, alocată în
    ....             // segmentul de date
}
void main()
{
    int k;           // k – variabilă locală funcției main(), alocată pe stivă
    printf("Sir constant"); // constanta "Sir constant" este memorată în segmentul
    ....             // de date
}

```

Este de dorit ca în cazul datelor a căror dimensiune nu este cunoscută a priori sau variază în limite largi, să se utilizeze o altă abordare: alocarea memoriei în mod dinamic. În mod dinamic, memoria nu mai este alocată în momentul compilării, ci în momentul execuției.

Alocarea dinamică elimină necesitatea definirii complete a tuturor cerințelor de memorie în momentul compilării.

## 9.2. Alocarea dinamică de memorie folosind funcții specifice

Alocarea de memorie dinamică se realizează în mod explicit, cu ajutorul funcțiilor de alocare dinamica în limbajul C. C++ completează posibilitățile de alocare dinamică prin operatorii new și delete care sunt adaptați programării orientate pe obiecte.

În limbajul C, alocarea memoriei în mod dinamic se face cu ajutorul funcțiilor **malloc**, **calloc**, **realloc**; eliberarea zonei de memorie se face cu ajutorul funcției free. Funcțiile de alocare/deallocare a memoriei au prototipurile în fișierele header **<stdlib.h>** și **<alloc.h>**:

Alocarea de memorie se poate face cu funcțiile:

**void \*malloc(size\_t nr\_octei\_de\_alocat);**

- Funcția malloc necesită un singur argument (numărul de octeți care vor fi alocați) și returnează un pointer generic către zona de memorie alocată (pointerul conține adresa primului octet al zonei de memorie rezervate).

**void \*calloc(size\_t nr\_elemente, size\_t mărimea\_în\_octeți\_a\_unui\_elem);**

- Funcția calloc lucrează în mod similar cu malloc; alocă memorie pentru un tablou de nr\_elemente, numărul de octeți pe care este memorat un element este mărimea\_în\_octeți\_a\_unui\_elem și returnează un pointer către zona de memorie alocată.

**void \*realloc(void \*ptr, size\_t mărime);**

- Funcția realloc permite modificarea zonei de memorie alocată dinamic cu ajutorul funcțiilor malloc sau calloc.

Funcțiile malloc, calloc, realloc returnează adresa de memorie la care s-a făcut alocarea. Valoarea întoarsă este un pointer generic (void\*). Aceasta, de regulă, se memorează într-o variabilă pointer de un tip oarecare, astfel că la atribuire este necesară conversia explicită către tipul

variabilei situată în stânga operatorului de atribuire.

În cazul în care nu se reușește alocarea dinamică a memoriei (memorie insuficientă), funcțiile malloc, calloc și realloc returnează un pointer **NULL** (valoare 0).

Eliberarea memoriei (alocate dinamic cu una dintre funcțiile malloc, calloc sau realloc) se realizează cu ajutorul funcției **free**.

**void free(void \*ptr);**

Exemple de folosire a funcțiilor de alocare dinamică:

```
int *p;
p=(int*)malloc(20*sizeof(int));           // se rezervă 20*sizeof(int) octeți, adresa fiind
                                           // memorată în variabila p

sau
p=(int*)calloc(20, sizeof(int));
```

Eliberarea de memorie se face:

```
free(p);
```



### Observații

- Alocarea făcută în exemplele anterioare poate fi gestionată similar cu un tablou alocat prin declarația

`int p[20];`

- Avantajul constă în faptul că, în momentul în care alocarea nu mai este necesară, se poate folosi funcția free(), memoria respectivă putând fi apoi folosită în alt scop.

Pentru a se evita generarea unor erori, se recomandă să se verifice dacă operațiile de alocare au reușit:

```
int *p;
p=(int*)malloc(20*sizeof(int));
if ( p== NULL)                               // dacă valoarea preluată de p este NULL
{                                              // (0), operația de alocare a eșuat, ceea ce
    printf("\nAlocare imposibila !");        // poate genera erori fatale
    exit(1);
}
....
```

Dimensiunea, în octeți, pentru memoria alocată poate fi exprimată nu doar printr-o constantă, ci și prin variabile sau chiar expresii diverse care returnează valoare întreagă pozitivă.

```
int dimensiune;
float*pr;
printf("\nIntroduceți dimensiunea :");
scanf("%d", &dimensiune);
pr = (float*)malloc( dimensiune * sizeof(float)) ;
....
free(pr) ;
```

```
printf("\nIntroduceți alta valoare pentru dimensiune.");
scanf("%d", &dimensiune);
pr = (float*)malloc( dimensiune * sizeof(float)) ;
...
free(pr) ;
```



### Observații

- Dimensiunea alocării e stabilită în momentul execuției programului prin citirea variabilei *dimensiune*.
- Alocarea de memorie poate fi repetată.
- Eliberarea memoriei alocată cu malloc, calloc se face numai cu funcția free; dacă nu este apelată funcția free și pointerul pr își schimbă valoarea, zona alocată nu va mai putea fi accesată, blocând memorie în mod inutil;
- Dacă operațiile de alocare se repetă fără eliberare de memorie făcută între timp, se poate ajunge la ocuparea întregii memorii.

### 9.3. Alocarea dinamică de memorie folosind operatorii new și delete

C++ introduce o nouă metodă pentru alocarea dinamică a memoriei adaptată programării orientate către obiecte.

Alocarea memoriei se face cu operatorul unar new, folosind următoarea sintaxă:

```
var_ptr = new tip_var;           // 1
var_ptr = new tip_var(val_init); // 2
var_ptr = new tip_var[dim];      // 3
```

unde: - var\_ptr = o variabilă pointer de un tip oarecare;

- tip\_var = tipul variabilei dinamice var\_ptr;

- val\_init = expresie cu a cărei valoare se inițializează variabila dinamică;

Varianta 1 alocă spațiu de memorie corespunzător unei date de tip tip\_var, fără inițializare.

Varianta 2 alocă spațiu de memorie corespunzător unei date de tip tip\_var, cu inițializare.

Varianta 3 alocă spațiu de memorie corespunzător unui tablou cu elemente de tip tip\_var de dimensiune dim. Inițializarea tabloului nu este posibilă.

Dacă alocarea de memorie a reușit, operatorul new returnează adresa zonei de memorie alocate. În caz contrar, returnează valoarea NULL (=0) (în cazul în care memoria este insuficientă sau fragmentată).

```
int n=3, *p1, *p2, *p3;
p1=new int;           // variabilă întreagă neinițializată
p2=new int(15);        // variabilă întreagă inițializată cu valoarea 15
p3=new int[n];         // tablou unidimensional int de dimensiune n
```

Eliminarea variabilei dinamice și eliberarea zonei de memorie se face cu operatorul **delete**, folosind sintaxa:

```
delete var_ptr;           // 1
```

sau

```
delete [ ]var_ptr;        // 2
```

unde var\_ptr conține adresa obținută în urma unei alocări cu operatorul new. Utilizarea altei valori

este ilegală, putând determina o comportare a programului necontrolată.

Varianta de dealocare (1) este folosită dacă la alocare s-au folosit variantele de alocare (1) sau (2), iar varianta (2) dacă la alocare s-au folosit varianta de alocare (3).

```
int *p;
p=new int(10);           // se alocă spațiul de memorie necesar unui int și se face
                        // inițializarea cu valoarea 10
delete p;                // se elimină variabila p din memorie

int *p,*q;
p=new int(7);            // se alocă spațiul de memorie necesar unui int și se face
                        // inițializarea cu valoarea 7
q=p;                    // variabila pointer q primește ca valoare adresa conținută
                        // în variabila p
delete q;                // se eliberează zona de memorie de la adresa conținută
                        // în variabila q
*p=17;                  // incorect, folosește o zona de memorie care a fost deja dealocată
int x=7;
int *y=&x;
delete y;                // incorect, se cere dealocarea unei zone de memorie
                        // care nu a fost alocată cu operatorul new

double * pr;
pr= new double[10];      // se alocă 10 elemente de tip double
...
delete [ ] pr;           // eliberarea spațiului pentru cele 10 elemente

double * pr;
pr= new double[10];      // se alocă 10 elemente de tip double
...
delete pr;                // se eliberează doar spațiul ocupat de primul element,
                        // celelalte 9 elemente rămânând alocate în continuare în memorie
```

#### 9.4. Alocarea dinamică de memorie pentru tablouri multidimensionale

Funcțiile, respectiv operatorul new, destinate alocării de memorie permit alocarea dinamică de tablouri, dar sintaxa permite stabilirea unei singure dimensiuni. Se pune problema construirii de tablouri multidimensionale.

Dacă se dorește alocarea unui tablou cu n dimensiuni (dim1, dim2,...,dimn), se poate alocă un tablou unidimensional cu un număr de elemente calculat prin produsul dim1\*dim2\*...\*dimn, urmând ca apoi să se gestioneze logic aceste elemente prin expresii care calculează indecșii elementelor care să descrie parcurgerea tablourilor.

De exemplu, pentru a construi o matrice (tablou bidimensional), cu n linii și m coloane, se alocă un tablou unidimensional de dimensiune n\*m. Indecșii elementelor se calculează cu expresia : (i \* m) + j, unde i reprezintă număr de linie, j reprezintă număr de coloană. Acest mod de abordare se regăsește în exemplul următor.

```
int n, m;                // n - nr. de linii ; m – nr. coloane
int * mat;               // pointer folosit pentru alocarea de memorie
int i, j;                // variabile folosite pentru parcurgerea elementelor matricei
```

```

// citirea dimensiunilor matricei
printf("\nNumarul de linii:");
scanf("%d", &n);
printf("\nNumarul de coloane:");
scanf("%d", &m);
// alocarea de memorie
mat=new int[n*m.];
//...
// dealocarea de memorie
delete [ ]mat;

```

O altă posibilitate de construire a matricei este folosirea unei duble indirectări (`int**mat`). Alocarea se face în două etape. O primă etapă în care se alocă un tablou unidimensional cu elemente de tip `int*` de dimensiune egală cu numărul de linii. Elementele acestuia vor memora adresele obținute în a doua etapă, când, într-o secvență repetitivă se vor alocă tablouri unidimensionale cu elemente de tip `int`. Se alocă un număr egal cu numărul de linii de tablouri, fiecare având un număr de elemente egal cu numărul de coloane.

```

int n, m;
int ** mat;
int i, j;
//....
//alocarea tabloului de adrese
mat=new int*[n];
// alocarea celor n linii ale matricei; alocarea se face distinct pentru fiecare linie
for(i=0; i<n ; i++)
    mat[i]=new int[m];
//....

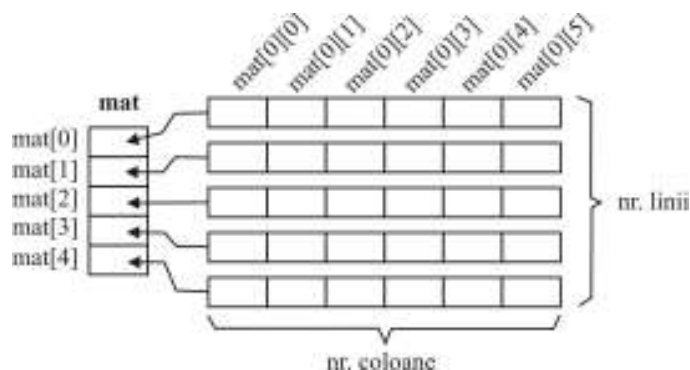
```

Eliberarea de memorie se va face tot în două etape, ordinea dealocării fiind inversă celei de alocare.

```

// dealocarea celor n linii ale matricei; se face distinct pentru fiecare linie
for(i=0; i<n ; i++)
    delete [ ]mat[i];
// dealocarea tabloului de adrese
delete [ ]mat;

```



**Fig. 9.1. Reprezentarea alocării dinamice de memorie a unei matrice folosind dubla indirectare (nr.linii = 5, nr.coloane = 6)**

Această metodă de lucru are două avantaje față de anterioara. Un prim avantaj, unul formal, îl constituie faptul că referirea elementelor se face similar declarației statice a tabloului (`mat[i][j]`). Cel de al doilea avantaj este faptul că, spre deosebire de alocarea ca tablou unidimensional, când toate elementele sunt alocate într-o zonă contiguă de memorie, în cazul alocării prin dubla indirectare alocarea se face dispart, fiecare linie putând fi alocată în altă zonă a segmentului heap. Aceasta duce la o mai bună utilizare a memoriei.

În Fig. 9.1. este reprezentat modul de alocare a memoriei pentru situația în care numărul de linii este 5, iar numărul de coloane este 6.



## Exemple

/\* \*\*\*\*\*

**Exemplu1 9.1.** - Se scrie un program în care se exemplifică folosirea alocării dinamice de memorie.

Se realizează alocare de blocuri de memorie în mod repetat, până la ocuparea întregii memorii.

/\* \*\*\*\*\*/

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

```
#include<conio.h>
```

```
#include<malloc.h>
```

```
void main()
```

```
{int i, dim;
```

```
// variabila dim determină dimensiunea blocurilor de
```

```
// memorie care se vor aloca
```

```
double *p;
```

```
printf("\nIntroduceți dimensiunea blocurilor: ");
```

```
scanf("%d", &dim);
```

```
// se folosește instrucțiunea for fără specificarea condiției de ieșire din ciclu;
```

```
// ieșirea se va face cu instrucțiunea "break" în momentul în care memoria a fost ocupată și
```

```
// alocarea nu se mai poate face, deci p ia valoarea NULL
```

```
for(i=1; ;i++)
```

```
{ p=(double *)malloc(dim*sizeof(double));
```

```
if( p != NULL)
```

```
printf("\nS-a alocat blocul nr. %d la adresa %u", i, p);
```

```
else
```

```
{ printf("\nAlocare imposibila - memorie insuficienta ");
```

```
break;
```

```
}
```

```
}
```

```
}
```

/\* \*\*\*\*\*

**Exemplu1 9.2.** - Se scrie un program în care se exemplifică folosirea alocării dinamice de memorie.

Sunt folosite, comparativ, secvențe în care se folosesc tablouri unidimensionale de date alocate static și tablouri alocate dinamic.

/\* \*\*\*\*\*/

```
#include<conio.h>
```

```

#include<stdio.h>
#include<malloc.h>

void main()
{ float tab[10];          // tablou alocat static
  float *p, aux;
  int i, dim;
//se citesc valori pentru elementele tabloului static
  for(i=0;i<10;i++)
    { printf("\n tab[%d]=", i);
      scanf("%f", &tab[i]);
    }
// se afiseaza valorile elementelor tabloului static
  for(i=0;i<10;i++)
    printf("\n tab[%d]=%f", i, tab[i]);
// alocare dinamica de memorie pentru un tablou de dimensiune stabilita de la tastatura
  printf("\nIntroduceti dimensiunea tabloului:");
  scanf("%d", &dim);
  p=(float*)malloc(dim*sizeof(float));
// se citesc valori pentru elementele tabloului alocat dinamic
  for(i=0 ; i<dim ; i++)
    { printf("\n p[%d] = ", i);
      scanf("%f", &aux);
      *(p+i)=aux;
    }
// se afiseaza valorile elementelor tabloului alocat dinamic
  for(i=0;i<dim;i++)
    printf("\n tab[%d] = %f", i, *(p+i));
// se elimina tabloul dinamic din memorie
  free(p);
// se face o realocare de memorie
  printf("\nIntroduceti dimensiunea tabloului care se realocă:");
  scanf("%d", &dim);
  p=(float*)malloc(dim*sizeof(float));
// se citesc valori pentru elementele tabloului alocat dinamic
  for(i=0;i<dim;i++)
    { printf("\n p[%d]=", i);
      scanf("%f", p+i);
    }
// se afiseaza valorile elementelor tabloului alocat dinamic
  for(i=0;i<dim;i++)
    printf("\n tab[%d]=%f", i, *(p+i));
// se elimina tabloul dinamic din memorie
  free(p);
  getch();
}

```

/\*\*\*\*\*

**Exemplu1 9.3.** - Se scrie un program în care se exemplifică folosirea alocării dinamice de memorie. Se alocă un tablou unidimensional de date, elementele sale fiind interpretate din punct de vedere logic ca aparținând unei matrice (se folosește simpla indirectare: int\*mat).

\*\*\*\*\*/

```

#include <conio.h>

```



```

#include <stdio.h>

void main()
{
    int n, m;        // n - nr. de linii ; m – nr. coloane
    int * mat;       // pointer folosit pentru alocarea de memorie
    int i, j;        // variabile folosite pentru parcurgerea elementelor matricei
    // citirea dimensiunilor matricei
    printf("\nNumarul de linii:");
    scanf("%d", &n);
    printf("\nNumarul de coloane:");
    scanf("%d", &m);
    // alocarea de memorie
    mat=new int[n*m];
    //citirea elementelor matricei
    for(i=0;i<n;i++)
        for(j=0;j<m;j++)
            { printf("mat[%d][%d]=",i,j);
              scanf("%d", &mat[i*m+j]);
            }
    // afişarea elementelor tabloului
    for(i=0 ; i<n ; i++)
    {   printf("\n");
        for(j=0;j<m;j++)
            printf("%5d",mat[i*m+j]);
    }
    //afişarea de informaţii despre matrice
    printf("\nsizeof(*mat)=%d", sizeof(*mat));           // se afişează numărul de
                                                         // octeţi ocupaţi de matrice
    printf("\n*(mat+0*m)=%d", *(mat+0*m));              // se afişează primul
                                                         // element de pe prima linie
    if(n>1)
        printf("\n*(mat+1*m)=%d", *(mat+1*m));         // se afişează primul
                                                         // element de pe linia a doua
    if(n>2)
        printf("\n*(mat+2*m)=%d", *(mat+2*m));         // se afişează primul element de pe
                                                         // linia a treia
    delete [ ]mat;                                       // dealocarea memoriei
}

```

/\*\*\*\*\*

**Exemplul 9.4.** - Se scrie un program în care se exemplifică folosirea alocării dinamice de memorie.

Se foloseşte dubla indirectare: int\*\*mat pentru a construi o matrice.

\*\*\*\*\*/

```

#include <conio.h>

```

```

#include <stdio.h>

```

```

void main()
{int n, m;
  int ** mat;
  int i, j;
  printf("\nNumarul de linii:");
  scanf("%d", &n);
  printf("\nNumarul de coloane:");

```

```

scanf("%d", &m);
mat=new int*[n];
for(i=0 ; i<n ; i++)
    mat[i]=new int[m];
//citirea elementelor matricei
for(i=0 ; i<n ; i++)
    for(j=0 ; j<m ; j++)
    {   printf("mat[%d][%d] = ", i, j);
        scanf("%d", &mat[i][j]);
    }
//afișarea elementelor matricei
for(i=0 ; i<n ; i++)
{
    printf("\n");
    for(j=0 ; j<m ; j++)
        printf("%5d", mat[i][j]);
}
//afișarea de informații despre matrice
printf("\nsizeof(*mat) = %d", sizeof(*mat)); // se afișează numărul de octeți ocupați de matrice
printf("\n**mat = %d", **mat);              // se afișează primul element de pe prima linie
if(n>1)
printf("\n**(mat+1) = %d", **(mat+1));        // se afișează primul element de pe linia a doua
if(n>2)
printf("\n**(mat+2)=%d", **(mat+2));          // se afișează primul element de pe linia a treia
for(i=0; i<n ; i++)                          // dealocarea memoriei
    delete [ ] mat[i];
delete [ ] mat;
}

```



### Întrebări. Exerciții. Probleme.

1. Se citesc de la tastatură elementele unui tablou unidimensional. Să se înlocuiască toate valorile negative cu valoarea 0. Se afișează elementele tabloului. Tabloul este alocat dinamic cu o dimensiune citită de la tastatură.
2. Se citesc de la tastatură elementele a două tablouri unidimensionale de aceeași dimensiune. Se calculează elementele unui al treilea tablou ca sumă a elementelor de același index ale primelor două și se afișează. Dimensiunea este citită de la tastatură, iar tablourile sunt alocate dinamic.
3. Se citesc de la tastatură două șiruri de caractere. Se determină un al treilea șir prin concatenarea primelor două și se afișează. Pentru referirea elementelor se vor folosi operații cu pointeri.
4. Se alocă o matrice pătratică a cărei dimensiune este citită de la tastatură. Se citesc de la tastatură valori pentru elementele matricei. Să se afișeze diagonala principală și cea secundară.
5. Sa se întocmească un program în care se alocă dinamic 3 matrice cu dimensiuni citite de la tastatură. Pentru două dintre acestea se citesc valorile elementelor de la tastatură. Se calculează a treia matrice ca sumă a primelor două și se afișează.