

Programarea calculatoarelor

Limbajul C



CURS 8



Alocare dinamică



Clase de alocare a memoriei

- Când, cum și unde se alocă memorie pentru o variabilă.
- Orice variabilă are o clasă de memorare care rezultă fie din declarația ei, fie implicit din locul unde este definită variabila.
- Zona de memorie utilizată de un program C cuprinde 4 subzone:
 1. Zona **text**: codul programului
 2. Zona de **date**: variabilele globale
 3. Zona **stivă**: date temporare (variabilele locale)
 4. Zona **heap**: memoria dinamică

Moduri de alocare a memoriei

- Statică: variabile implementate în zona de date
- Auto: variabile implementate în stivă
- Dinamică: variabile implementate în heap
 - Memoria se alocă dinamic (la execuție) în zona “heap” atașată programului, dar numai la cererea explicită a programatorului, prin apelarea unor funcții de bibliotecă (malloc, calloc, realloc).
 - Memoria este eliberată numai la cerere, prin apelarea funcției “free”
- Register: variabile implementate într-un registru de memorie

Variabile locale / variabile globale

- Durata de viață vs. domeniu de vizibilitate

	Variabile globale	Variabile locale
Alocare	Statică; la compilare	Auto; la execuție bloc
Durata de viață	Cea a întregului program	Cea a blocului în care e declarată
Inițializare	Cu zero	Nu se face automat

Clase de alocare a memoriei: Auto

```
int doi() {
    int x = 2;
    return x;
}
void main() {
    int a;
    {
        int b = 5;
        a = b*doi();
    }
    printf("a = %d\n", a);
}
```

Clase de alocare a memoriei: Auto

Conținut stivă:

x 2

b 5

a 10

- Variabilele locale unui bloc (unei funcții) și argumentele formale sunt implicit din clasa *auto*;
- Durata locală:
 - memoria este alocată automat, la activarea unei funcții, în zona stivă alocată unui program și este eliberată automat la terminarea funcției.

Clase de alocare a memoriei: Static

- Memoria este alocată la compilare în segmentul de date din cadrul programului și nu se mai poate modifica în cursul execuției
- Variabilele globale sunt implicit statice
- Pot fi declarate *static* și variabile locale, definite în cadrul funcțiilor.
- O variabilă sau o funcție declarată static are durata de viață egală cu cea a programului
- **static** face ca o variabilă globală sau o funcție să fie *privată (proprie)* unității unde a fost definită: ea devine inaccesibilă altei unități, chiar prin folosirea lui **extern**.

Clase de alocare a memoriei: Static

```
int f(void) {
    static int nr_apeluri=0;
    nr_apeluri++;
    printf("\nfuncția f() este apelata pentru a %d-a oara.", nr_apeluri);
    // ...
    return nr_apeluri;
}
```

```
int f(void);
int main(void) {
    int i;
    for (i=0; i<10; i++)
        if (cond(i)) f();
    return 0;
}
```


Observație

- O variabilă statică declarată într-o funcție:
 - își păstrează valoarea între apeluri succesive ale funcției
 - spre deosebire de variabilele *auto* care sunt realocate pe stivă la fiecare apel al funcției și pornesc de fiecare dată cu valoarea primită la inițializarea lor (sau cu o valoare imprevizibilă, dacă nu sunt inițializate).
- Variabilele locale statice se folosesc foarte rar în practica programării (funcția de bibliotecă “strtok” este un exemplu de funcție cu o variabilă statică).

Clase de alocare a memoriei: Register

- O variabilă declarată *register* solicită sistemului alocarea ei într-un registru mașină, dacă este posibil
- De obicei se lasă compilatorului decizia de alocare a registrelor mașinii pentru anumite variabile *auto* din funcții
- Se utilizează pentru variabile “foarte solicitate”, pentru mărirea vitezei de execuție:

```
{  
    register int i;  
    for(i = 0; i < N; ++i){  
        /* ... */  
    }  
} /* se elibereaza registrul */
```

Funcții C pentru alocarea dinamică a memoriei

- `stdlib.h`

`void *malloc(int numar_de_octeti);`

- Alocă memorie de dimensiunea cerută

`void *calloc(int nitems, int size);`

- Alocă memorie de dimensiunea cerută și inițializează zona alocată cu zerouri

Funcții C pentru alocarea dinamică a memoriei

- Au ca rezultat adresa zonei de memorie alocate (de tip *void **)
- Au ca argument dimensiunea zonei de memorie alocate
- Dacă cererea de alocare nu poate fi satisfăcută, pentru că nu mai exista un bloc continuu de dimensiunea solicitată, atunci funcțiile de alocare au rezultat NULL.
- Funcțiile de alocare au rezultat *void** deoarece funcția nu știe tipul datelor ce vor fi memorate la adresa respectivă.
- Se folosesc:
 - Operatorul *sizeof* pentru a determina numărul de octeți necesar unui tip de date (variabile);
 - Operatorul de conversie *cast* pentru adaptarea adresei primite de la funcție la tipul datelor memorate la adresa respectivă (conversie necesară atribuirii între pointeri de tipuri diferite).

Funcții C pentru alocarea dinamică a memoriei

```
void *realloc(void* adr,int numar_de_octeti);
```

- Alocă o zonă de dimensiunea specificată ca al doilea argument
- Copiază la noua adresă datele de la adresa veche (primul argument)
- Eliberează memoria de la adresa veche.

Atenție!

- Realocarea repetată de memorie poate conduce la fragmentarea memoriei “heap”, adică la crearea unor blocuri de memorie libere dar neadiacente și prea mici pentru a mai fi reutilizate ulterior.
- Se va evita redimensionarea unui vector cu o valoare foarte mică de un număr mare de ori; o strategie de realocare folosită pentru vectori este dublarea capacității lor anterioare.

Eliberarea memoriei

```
void free(void* adr);
```

- Eliberează memoria de la adresa adr (dimensiunea ei este memorată la începutul zonei alocate (de către funcția de alocare)
- Eliberarea memoriei prin "free" este inutilă la terminarea unui program, deoarece înainte de încărcarea și lansarea în execuție a unui nou program se eliberează automat toată memoria "heap"!

Exemple

- `char *str;`

```
str=(char *)malloc(10*sizeof(char));
```

```
str=(char *)realloc(str,20*sizeof(char));
```

```
free(str);
```

- `int *p;`

```
p=(int *)malloc(50*sizeof(int));
```

- `int * a= (int*) calloc (n, sizeof(int));`

Exerciții – alocare dinamică

1. Program care alocă spațiu pentru o variabilă întreagă dinamică, după citire și tipărire, spațiul fiind eliberat. Modificați programul astfel încât variabila dinamică să fie de tip double.
2. Program care citește mai multe linii de text de la tastatură, alocă dinamic memorie pentru fiecare linie și pune adresa liniei într-un vector de pointeri (vector alocat și realocat dinamic).

Rezolvare 1

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

int main(){
    int *pi;
    pi=(int *)malloc(sizeof(int));
    if(pi==NULL){
        puts("*** Memorie insuficienta ***");
        return 1; // revenire din main
    }
    printf("valoare:");
    //citirea variabilei dinamice, de pe heap, de la adresa din pi!!!
    scanf("%d",pi);
}
```

Rezolvare 1

```
*pi*=2;          // dublarea valorii
printf("val=%d,pi(adresa pe heap)=%p,adr_pi=%p\n", *pi, pi, &pi);

/* sizeof aplicat unor expresii */
printf("%d %d %d\n",sizeof(*pi), sizeof(pi), sizeof(&pi));

free(pi);        //eliberare spatiu
printf("pi(dupa elib):%p\n",pi);          // nemodificat, dar invalid

system("pause");
return 0;
}
```

Rezolvare 2

```
#include <stdio.h>
#include <stdlib.h>
#define INCR 4
int main() {
    int n,n_crt,i ;
    float x, * v;
    n=INCR;    // dimensiune memorie alocata
    n_crt=0;   // numar curent elemente în vector
    v=(float *)malloc (n*sizeof(float)); //alocare initiala
```

Rezolvare 2

```
while (scanf("%f",&x) !=EOF){
    if (n_crt == n) {
        n= n+ INCR;
        v=(float *) realloc (v, n*sizeof(float) );    //realocare
    }
    v[n_crt++]=x;
}
for (i=0;i<n_crt;i++)
    printf ( "%.2f ",v[i]);
free(v);
system("pause");
return 1;
}
```

Observație

- Nu orice vector cu dimensiune constantă este un vector static!!
- Un vector definit într-o funcție (alta decât “main”) nu este static deoarece nu ocupă memorie pe toată durata de execuție a programului, deși dimensiunea sa este stabilită la scrierea programului.
- Un vector definit într-o funcție este alocat pe stivă, la activarea funcției, iar memoria ocupată de vector este eliberată automat la terminarea funcției.

Matrice alocate dinamic

- Folosește într-un mod economic memoria și evită alocări acoperitoare, estimative.
- Permite matrice cu linii de lungimi diferite.
- Reprezintă o soluție bună la problema argumentelor de funcții de tip matrice.
- O matrice alocată dinamic este de fapt un *vector de pointeri către fiecare linie din matrice*, deci un vector de pointeri la vectori alocați dinamic.
- O astfel de matrice se poate folosi la fel ca o matrice declarată cu dimensiuni constante.

Matrice alocate dinamic

- Dacă numărul de linii este cunoscut sau poate fi estimată valoarea lui maximă, atunci vectorul de pointeri are o dimensiune constantă:

```
int * a[M];
```

- Dacă nu se poate estima numărul de linii din matrice atunci și vectorul de pointeri se alocă dinamic:

```
int** a;
```

- se va alocă mai întâi memorie pentru un vector de pointeri (funcție de numărul liniilor)
- apoi se va alocă memorie pentru fiecare linie (funcție de numărul coloanelor) cu memorarea adreselor liniilor în vectorul de pointeri.

Exemplu

- Să se scrie funcții de alocare a memoriei și afișare a elementelor unei matrice de întregi alocată dinamic.

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

```
// rezultat adresa matrice sau NULL
int ** intmat ( int nl, int nc) {
    int i;
    int ** p=(int **) malloc (nl*sizeof (int*));
    if ( p != NULL)
        for (i=0; i<nl ;i++)
            p[i] =(int*) calloc (nc,sizeof (int));
    return p;
}
```


Exemplu

```
void printmat (int ** a, int nl, int nc) {
    int i,j;
    for (i=0;i<nl;i++) {
        for (j=0;j<nc;j++)
            printf ("%2d", a[i][j] );
        printf("\n");
    }
}
int main () {
    int **a, nl, nc, i, j;
    printf ("nr linii și nr coloane: \n");
    scanf ("%d%d", &nl, &nc);
    a= intmat(nl,nc);
    for (i=0;i<nl;i++)
        for (j=0;j<nc;j++)
            a[i][j]= nc*i+j+1;
    printmat (a ,nl,nc);
    return 1;
}
```

Vectori de pointeri la date alocate dinamic

- Date alocate dinamic ale căror adrese sunt reunite într-un vector de pointeri.
- Situațiile cele mai frecvente sunt:
 - vectori de pointeri la șiruri de caractere alocate dinamic
 - vectori de pointeri la structuri alocate dinamic.

Exemplu

- Program pentru citirea unor nume, alocare dinamică a memoriei pentru fiecare șir (în funcție de lungimea șirului citit) și pentru memorarea adreselor șirurilor într-un vector de pointeri. În final se vor afișa numele citite, pe baza vectorului de pointeri.
- Să se adauge programului anterior o funcție de ordonare a vectorului de pointeri la șiruri, pe baza conținutului fiecărui șir. Programul va afișa lista de nume în ordine alfabetică.
- Apoi vectorul de pointeri se va aloca dinamic, funcție de numărul de șiruri.

Exemplu

```
void printstr ( char * vp[], int n) { //afisare
    int i;
    for(i=0;i<n;i++)
        printf ("%s\n",vp[i]);
}
```

```
int readstr (char * vp[]) { // citire siruri și creare vector de pointeri
    int n=0; char * p, sir[80];
    while ( scanf ("%s", sir) == 1) {
        vp[n]= (char*) malloc (strlen(sir)+1);
        strcpy( vp[n],sir);
        //sau: vp[n]=strdup(sir);
        ++n;
    }
    return n;
}
```

Exemplu

```
/* ordonare vector de pointeri la șiruri prin Bubble Sort (metoda  
bulelor)*/
```

```
void sort ( char * vp[],int n) {  
    int i,j,schimb=1;  
    char * tmp;  
    while(schimb){  
        schimb=0;  
        for (i=0;i<n-1;i++)  
            if ( strcmp (vp[i],vp[i+1])>0) {  
                tmp=vp[i];  
                vp[i]=vp[i+1];  
                vp[i+1]=tmp;  
                schimb=1;  
            }  
    }  
}
```

Exemplu

```
int main () {
    int n;
    char * vp[1000];
    // vector de pointeri, cu dimensiune fixa

    n=readstr(vp); // citire siruri și creare vector
    sort ( vp,n); // ordonare vector
    printstr (vp,n); // afișare șiruri

    system("pause");
    return 1;
}
```

Observație

- O funcție **nu** poate avea ca rezultat un vector sub forma:
`int [] funcție(...) {...}`
- O funcție poate avea ca rezultat un pointer !!
`int *funcție(...) {...}`
- De obicei, rezultatul pointer este egal cu unul din argumente, eventual modificat în funcție.

Exemplu corect:

```
// incrementare pointer p
char * incptr ( char * p) {
    return ++p;
}
```

- *Atenție! Acest pointer nu trebuie să conțină adresa unei variabile locale!*

Atenție!

- O variabilă locală are o existență temporară, garantată numai pe durata executării funcției în care este definită (cu excepția variabilelor locale statice)
- Adresa unei astfel de variabile nu trebuie transmisă în afara funcției, pentru a fi folosită ulterior!!

Exemplu greșit:

```
// vector cu cifrele unui nr intreg de maxim cinci cifre
int * cifre (int n) {
    int k, c[5];                // vector local
    for (k=4;k>=0;k--) {
        c[k]=n%10; n=n/10;
    }
    return c;                  // aici este eroarea !
}
//warning la compilare și rezultate greșite în main!!
```


funcție cu rezultat pointer

- O funcție care trebuie să transmită ca rezultat un vector poate fi scrisă corect în mai multe feluri:
1. Primește ca argument adresa vectorului (*definit și alocat în altă funcție*) și depune rezultatele la adresa primită (este soluția recomandată!!)

```
void cifre (int n, int c[ ]) {
    int k;
    for (k=4;k>=0;k--) {
        c[k]=n%10; n=n/10;
    }
}
int main(){
    int a[10];
    ....
    cifre(n,a);
    ....
}
```

funcție cu rezultat pointer

2. Alocă dinamic memoria pentru vector (cu "malloc")
 - această alocare (pe heap) se menține și la ieșirea din funcție.
 - funcția are ca rezultat adresa vectorului alocat în cadrul funcției.
 - problema este unde și când se eliberează memoria alocată.

```
int * cifre (int n) {  
    int k, *c;           Programarea calculatoarelor  
                        // vector local
```

Exemplu: corect sau greșit?

```
// extrage un subsir de lungime n de la adresa s
char * substr (char* s, int n) {
    char aux[1000];           // o variabila locala pentru subsirul rezultat
    int m;
    m=strlen(s);             // cate caractere mai sunt la adresa s
    if (n>m) n=m;
    strncpy(aux,s,n);
    aux[n]=0;                // terminator de (sub)sir
    return aux;              // pointer la o variabila locala !
}

// verificare funcție
void main () {
    char s[]="abcdef";
    puts (substr(s,3));
    puts (substr (substr(s,4),2));
}
```

- `aux= (char*) malloc(strlen(s)+1);`
- `char * aux=strdup(s);`