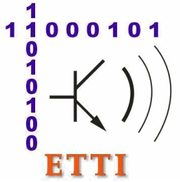
UNIVERSITATEA POLITEHNICA DIN BUCUREȘTI



Laborator 1

Pointeri și alocarea memoriei în C

Liviu-Daniel ȘTEFAN, Mihai DOGARIU, Bogdan IONESCU

Cuprins

Partea I – Noțiuni teoretice

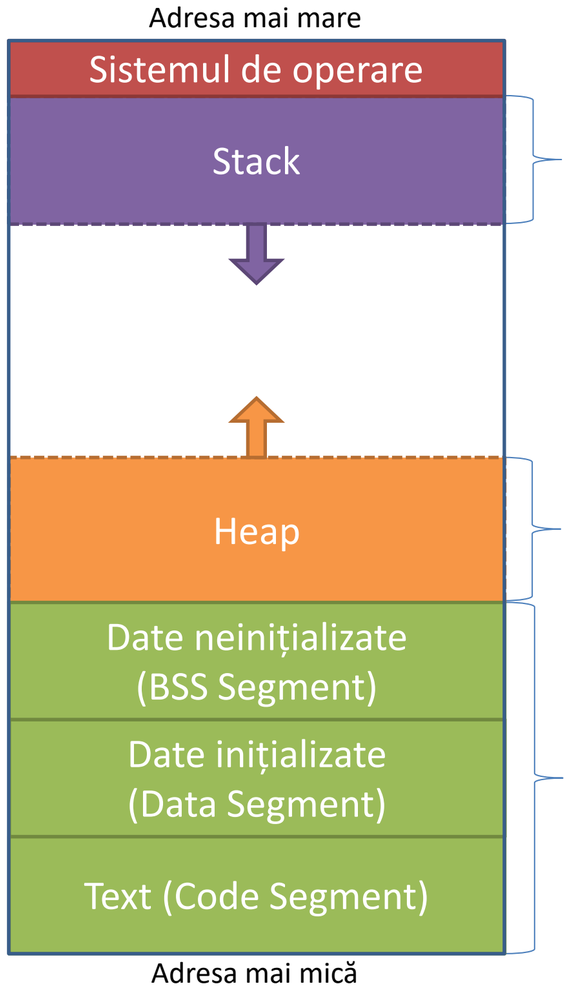
L1.1. Pointeri
L1.2. Alocarea dinamică de memorie
L1.3. Tablouri N-dimensionale

Partea II – Aplicații Laborator

Rezolvare aplicații Moodle

L1.1. Pointeri

Limbajul C folosește 3 tipuri de memorie:
1. statică/globală
2. dinamică (heap)
3. automatică (stack)



– variabilele locale (parametri formali)

– alocare dinamică de memorie

– variabile statice și globale

L1.1. Pointeri

Sfera de influență și durata de viață a variabilelor folosite în diferite regiuni de memorie:

Tip	Sfera de influență	Durata de viață
Globală	Întregul fișier	Durata de viață a aplicației
Statică	Funcția în care este declarată	Durata de viață a aplicației
Automatică	Funcția în care este declarată	Perioada de execuție a funcției
Dinamică	Este limitat la pointerul sau pointerii care fac referire la memoria respectivă	Până când memoria este eliberată

Definiție:

Pointerii sunt variabile ale căror valori sunt adresele altor variabile.

Sintaxă:

```
int *p; // declarăm o variabilă p de tip pointer către un întreg
const int *pci; // pci este o variabilă de tip pointer către un întreg constant
```

L1.1. Pointeri

Limbajul C pune la dispoziție o serie de operatori disponibili pentru lucrul cu pointeri:

Operator	Nume	Descriere
*	Asterisc	Folosit pentru declararea unui pointer
*	Indirectare	Folosit pentru dereferențierea unui pointer
&	Adresare	Folosit pentru a obține adresa unei variabile
->	La adresa lui	Folosit pentru a accesa un câmp al unei structuri printr-un pointer
+	Adunare	Folosit pentru a incrementa un pointer
-	Scădere	Folosit pentru a decrementa un pointer
==, !=	Egalitate, Inegalitate	Compară doi pointeri
>, >=, <, <=	Mai mare decât, mai mare sau egal cu, mai mic decât, mai mic sau egal cu	Compară doi pointeri
(<specificator_de_tip>)	Cast	Realizează conversia tipului pointerului la tipul specificatorului de tip

L1.1. Pointeri

Operatorul de adresare “&”

Operatorul “&” returnează adresa din memorie a operandului său.

```
int num = 0;
int *p = NULL;
```

Identificator	Adresă	Valoare
num	100	0
p	104	NULL
	108	…

L1.1. Pointeri

Operatorul de adresare “&”

Operatorul “&” returnează adresa din memorie a operandului său.

```
int num = 0;
int *p = NULL;
p = &num; // atribuim pointerului p adresa lui num
```

Identificator	Adresă	Valoare
num	100	0
p	104	NULL
	108	…

L1.1. Pointeri

Operatorul de adresare “&”

Operatorul “&” returnează adresa din memorie a operandului său.

```
int num = 0;
int *p = NULL;
p = &num; // atribuim pointerului p adresa lui num
```

Identificator	Adresă	Valoare
num	100	0
p	104	100
	108	…

L1.1. Pointeri

Operatorul de adresare “&”

Operatorul “&” returnează adresa din memorie a operandului său.

```
int num = 0;
int *p = NULL;
p = &num; // atribuim pointerului p adresa lui num
```

Identificator	Adresă	Valoare
num	100	0
p	104	100
	108	…

Operatorul de indirectare “*”

Operatorul “*” returnează valoarea de la adresa stocată în operandul său.

```
int num = 5;
int *p = NULL;
```

Identificator	Adresă	Valoare
num	100	5
p	104	NULL
	108	…

L1.1. Pointeri

Operatorul de adresare “&”

Operatorul “&” returnează adresa din memorie a operandului său.

```
int num = 0;
int *p = NULL;
p = &num; // atribuim pointerului p adresa lui num
```

Identificator	Adresă	Valoare
num	100	0
p	104	100
	108	…

Operatorul de indirectare “*”

Operatorul “*” returnează valoarea de la adresa stocată în operandul său.

```
int num = 5;
int *p = NULL;
p = &num;
```

Identificator	Adresă	Valoare
num	100	5
p	104	NULL
	108	…

L1.1. Pointeri

Operatorul de adresare “&”

Operatorul “&” returnează adresa din memorie a operandului său.

```
int num = 0;
int *p = NULL;
p = &num; // atribuim pointerului p adresa lui num
```

Identificator	Adresă	Valoare
num	100	0
p	104	100
	108	…

Operatorul de indirectare “*”

Operatorul “*” returnează valoarea de la adresa stocată în operandul său.

```
int num = 5;
int *p = NULL;
p = &num;
```

Identificator	Adresă	Valoare
num	100	5
p	104	100
	108	…

L1.1. Pointeri

Operatorul de adresare “&”

Operatorul “&” returnează adresa din memorie a operandului său.

```
int num = 0;
int *p = NULL;
p = &num; // atribuim pointerului p adresa lui num
```

Identificator	Adresă	Valoare
num	100	0
p	104	100
	108	…

Operatorul de indirectare “*”

Operatorul “*” returnează valoarea de la adresa stocată în operandul său.

```
int num = 5;
int *p = NULL;
p = &num;
printf(”%d\n”, *p); // afișează valoarea 5
```

Identificator	Adresă	Valoare
num	100	5
p	104	100
	108	…

L1.1. Pointeri

Operatorul de adresare “&”

Operatorul “&” returnează adresa din memorie a operandului său.

```
int num = 0;
int *p = NULL;
p = &num; // atribuim pointerului p adresa lui num
```

Identificator	Adresă	Valoare
num	100	0
p	104	100
	108	…

Operatorul de indirectare “*”

Operatorul “*” returnează valoarea de la adresa stocată în operandul său.

```
int num = 5;
int *p = NULL;
p = &num;
printf(”%d\n”, *p); // afișează valoarea 5
*p = 200;
```

Identificator	Adresă	Valoare
num	100	5
p	104	100
	108	…

L1.1. Pointeri

Operatorul de adresare “&”

Operatorul “&” returnează adresa din memorie a operandului său.

```
int num = 0;
int *p = NULL;
p = &num; // atribuim pointerului p adresa lui num
```

Identificator	Adresă	Valoare
num	100	0
p	104	100
	108	…

Operatorul de indirectare “*”

Operatorul “*” returnează valoarea de la adresa stocată în operandul său.

```
int num = 5;
int *p = NULL;
p = &num;
printf(”%d\n”, *p); // afișează valoarea 5
*p = 200;
```

Identificator	Adresă	Valoare
num	100	200
p	104	100
	108	…

L1.1. Pointeri

Operatorul de adresare “&”

Operatorul “&” returnează adresa din memorie a operandului său.

```
int num = 0;
int *p = NULL;
p = &num; // atribuim pointerului p adresa lui num
```

Identificator	Adresă	Valoare
num	100	0
p	104	100
	108	…

Operatorul de indirectare “*”

Operatorul “*” returnează valoarea de la adresa stocată în operandul său.

```
int num = 5;
int *p = NULL;
p = &num;
printf(”%d\n”, *p); // afișează valoarea 5
*p = 200;
printf("%d\n", num); // afișează valoarea 200
```

Identificator	Adresă	Valoare
num	100	200
p	104	100
	108	…

L1.2. Alocare dinamică

Tipul size_t

Tipul size_t reprezintă dimensiunea maximă pe care o poate avea orice obiect in C.

```
typedef unsigned int size_t;
```

Funcția malloc

Alocă un singur bloc de memorie de dimensiune precizată, populat cu valori nedeterminate, și returnează un pointer de tip void către începutul blocului.

```
void* malloc (size_t size); // prototipul funcției
int *p = malloc (n*sizeof(int)); // alocare memorie pentru un vector de n întregi
```

Funcția calloc

Alocă un bloc de memorie pentru un vector de n elemente de dimensiune precizată, populat cu valori de 0, și returnează un pointer de tip void către începutul blocului.

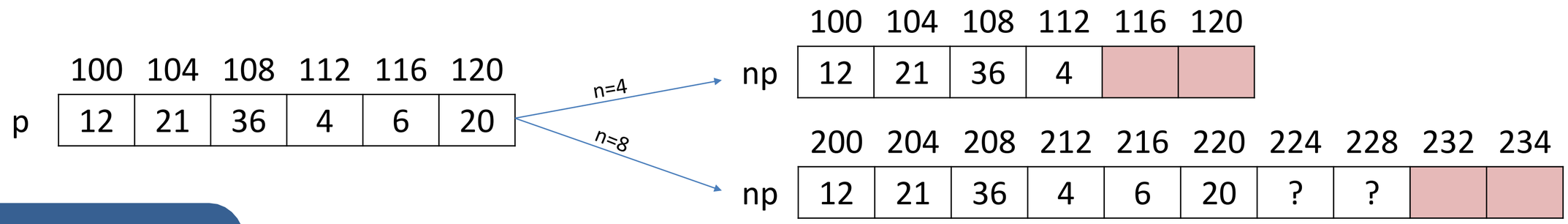
```
void* calloc (size_t num, size_t size); // prototipul funcției
int *p = calloc (n, sizeof(int)); // alocare memorie pentru un vector de n întregi
```

L1.2. Alocare dinamică

Funcția realloc

Schimbă dimensiunea blocului de memorie către care indică un pointer și returnează un pointer de tip void către începutul noului bloc. Această funcție mută complet blocul de memorie către o nouă adresă (returnată de funcție).

```
void* realloc (void* ptr, size_t size); // prototipul funcției
int *np = realloc (p, n*sizeof(int)); // realocă pointerul p pentru un vector de n întregi
```



Funcția free

Eliberează blocul de memorie către care indică pointerul transmis ca argument, alocat anterior cu una dintre funcțiile malloc, calloc sau realloc. Se folosește deoarece aceste zone de memorie nu sunt eliberate în mod automat.

```
void free (void* ptr); // prototipul funcției
free(p); // eliberează memoria către care indică pointerul p
```

L1.3. Tablouri N-dimensionale

Toate tablourile N-dimensionale (vectori, matrice, tensori - N≥3) vor fi alocate dinamic. Motivație:

- prevenirea alocării mai multor resurse decât necesar;
- reutilizarea memoriei;
- gestiune mai facilă prin intermediul pointerilor.

Tablouri unidimensionale

Static

```
int i, n, v[10];
scanf(“%d”, &n);
for (i = 0; i < n; i++){
  scanf("%d", &v[i]);
}
for (i = 0; i < n; i++){
  printf(“%d\t", v[i]);
}
```

L1.3. Tablouri N-dimensionale

Toate tablourile N-dimensionale (vectori, matrice, tensori - N≥3) vor fi alocate dinamic. Motivație:

- prevenirea alocării mai multor resurse decât necesar;
- reutilizarea memoriei;
- gestiune mai facilă prin intermediul pointerilor.

Tablouri unidimensionale

Static

```
int i, n, v[10];
scanf(“%d”, &n);
for (i = 0; i < n; i++){
  scanf("%d", &v[i]);
}
for (i = 0; i < n; i++){
  printf(“%d\t", v[i]);
}

n=4
```

Dinamic

```
int i, n, *v=NULL;
scanf(“%d”, &n);
v = malloc(n * sizeof(int));
for (i = 0; i < n; i++){
  scanf("%d", v+i);
}
for (i = 0; i < n; i++){
  printf(“%d\t", *(v+i));
}
```

L1.3. Tablouri N-dimensionale

Toate tablourile N-dimensionale (vectori, matrice, tensori - N≥3) vor fi alocate dinamic. Motivație:

- prevenirea alocării mai multor resurse decât necesar;
- reutilizarea memoriei;
- gestiune mai facilă prin intermediul pointerilor.

Tablouri unidimensionale

Static

```
int i, n, v[10];
scanf(“%d”, &n);
for (i = 0; i < n; i++){
  scanf("%d", &v[i]);
}
for (i = 0; i < n; i++){
  printf(“%d\t", v[i]);
}
```

n=4

12	3	8	4						

Dinamic

```
int i, n, *v=NULL;
scanf(“%d”, &n);
v = malloc(n * sizeof(int));
for (i = 0; i < n; i++){
  scanf("%d", v+i);
}
for (i = 0; i < n; i++){
  printf(“%d\t", *(v+i));
}
```

L1.3. Tablouri N-dimensionale

Toate tablourile N-dimensionale (vectori, matrice, tensori - N≥3) vor fi alocate dinamic. Motivație:

- prevenirea alocării mai multor resurse decât necesar;
- reutilizarea memoriei;
- gestiune mai facilă prin intermediul pointerilor.

Tablouri unidimensionale

Static

```
int i, n, v[10];
scanf(“%d”, &n);
for (i = 0; i < n; i++){
  scanf("%d", &v[i]);
}
for (i = 0; i < n; i++){
  printf(“%d\t", v[i]);
}
```

n=4

12	3	8	4						

Dinamic

```
int i, n, *v=NULL;
scanf(“%d”, &n);
v = malloc(n * sizeof(int));
for (i = 0; i < n; i++){
  scanf("%d", v+i);
}
for (i = 0; i < n; i++){
  printf(“%d\t", *(v+i));
}
```

12	3	8	4

L1.3. Tablouri N-dimensionale

Toate tablourile N-dimensionale (vectori, matrice, tensori - N≥3) vor fi alocate dinamic. Motivație:

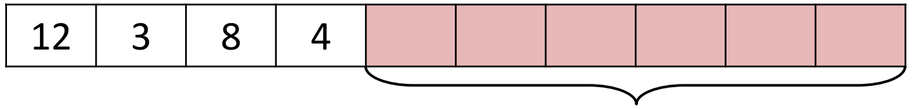
- prevenirea alocării mai multor resurse decât necesar;
- reutilizarea memoriei;
- gestiune mai facilă prin intermediul pointerilor.

Tablouri unidimensionale

Static

```
int i, n, v[10];
scanf(“%d”, &n);
for (i = 0; i < n; i++){
  scanf("%d", &v[i]);
}
for (i = 0; i < n; i++){
  printf(“%d\t", v[i]);
}
```

n=4



Dinamic

```
int i, n, *v=NULL;
scanf(“%d”, &n);
v = malloc(n * sizeof(int));
for (i = 0; i < n; i++){
  scanf("%d", v+i);
}
for (i = 0; i < n; i++){
  printf(“%d\t", *(v+i));
}
```

L1.3. Tablouri N-dimensionale

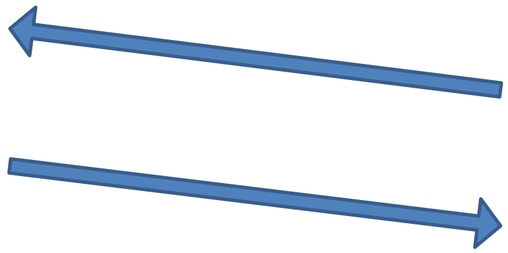
Toate tablourile N-dimensionale (vectori, matrice, tensori - N≥3) vor fi alocate dinamic. Motivație:

- prevenirea alocării mai multor resurse decât necesar;
- reutilizarea memoriei;
- gestiune mai facilă prin intermediul pointerilor.

Tablouri unidimensionale

Static

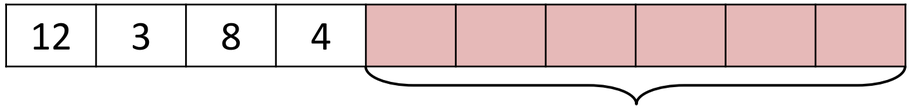
```
int i, n, v[10];
scanf(“%d”, &n);
for (i = 0; i < n; i++){
  scanf("%d", &v[i]);
}
for (i = 0; i < n; i++){
  printf(“%d\t", v[i]);
}
```



Dinamic

```
int i, n, *v=NULL;
scanf(“%d”, &n);
v = malloc(n * sizeof(int));
for (i = 0; i < n; i++){
  scanf("%d", v+i);
}
for (i = 0; i < n; i++){
  printf(“%d\t", *(v+i));
}
```

n=4



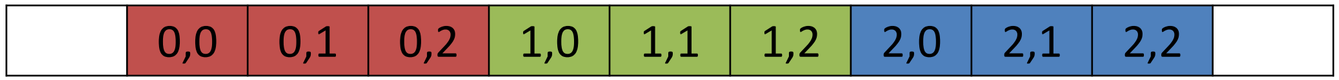
L1.3. Tablouri N-dimensionale

Tablouri bidimensionale

Memoria procesorului este o secvență lungă de locații de memorie consecutive, așezate sub forma unui vector foarte lung. Tabloruile bidimensionale (matricele) sunt scrise în memorie sub forma unui vector după regula “rând-major” (row-major).

rând, coloană

0,0	0,1	0,2
1,0	1,1	1,2
2,0	2,1	2,2



Orice altă linie suplimentară a matricei ar fi scrisă în continuarea șirului de valori de mai sus, fără să afecteze informația ce îi precede. Din acest motiv, din prototipurile funcțiilor ce conțin ca argument o matrice poate să lipsească numărul de linii al matrice, e.g. “`int max_matrice (int A[][10])`”.

L1.3. Tablouri N-dimensionale

Tablouri bidimensionale

Static

```
int i, j, M[3][3];
int nr_randuri, nr_coloane;
scanf(“%d”, &nr_randuri);
scanf(“%d”, &nr_coloane);
for (i = 0; i < nr_randuri; i++){
  for (j = 0; j < nr_coloane; j++){
    scanf("%d", &M[i][j]);
}
for (i = 0; i < nr_randuri; i++){
  for (j = 0; j < nr_coloane; j++){
    printf("%d\t", M[i][j]);
}
```

```
nr_randuri = 2
nr_coloane = 2
```

12	21	
36	4	

12	21		36	4				

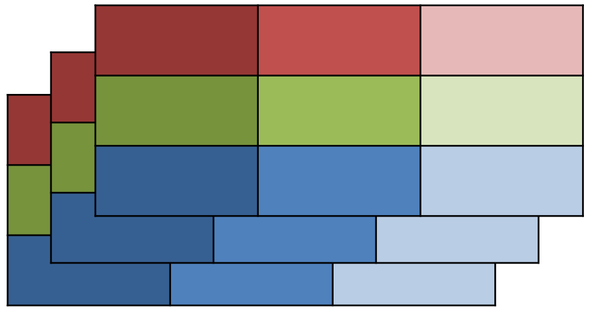
Dinamic

```
int i, j, *M=NULL;
int nr_randuri, nr_coloane;
scanf(“%d”, &nr_randuri);
scanf(“%d”, &nr_coloane);
M = malloc(nr_randuri * nr_coloane * sizeof(int));
for (i = 0; i < nr_randuri; i++){
  for (j = 0; j < nr_coloane; j++){
    scanf("%d", M + i * nr_coloane + j);
}
for (i = 0; i < nr_randuri; i++){
  for (j = 0; j < nr_coloane; j++){
    printf("%d\t", *(M + i * nr_coloane + j));
}
```

12	21	36	4

Tablouri N-dimensionale

Tablourile N-dimensionale respectă aceeași convenție “rând-major” la stocarea în memorie. Exemplu pentru N=3:



0,0,0	0,1,0	0,2,0
1,0,0	1,1,0	1,2,0
2,0,0	2,1,0	2,2,0

0,0,1	0,1,1	0,2,1
1,0,1	1,1,1	1,2,1
2,0,1	2,1,1	2,2,1

0,0,2	0,1,2	0,2,2
1,0,2	1,1,2	1,2,2
2,0,2	2,1,2	2,2,2

0,0,0	0,0,1	0,0,2	0,1,0	0,1,1	0,1,2	0,2,0	0,2,1	0,2,2	1,0,0	1,0,1	1,0,2	1,1,0	1,1,1	1,1,2	1,2,0	1,2,1	1,2,2

2,0,0	2,0,1	2,0,2	2,1,0	2,1,1	2,1,2	2,2,0	2,2,1	2,2,2

Tablouri N-dimensionale

Considerăm un pointer p ce indică un tablou N-dimensional, dim_i mărimea dimensiunii i, cu $i = 1, \dots, N$, și n_i indexul elementului curent din dimensiunea i, atunci adresa elementului a cărui indexare statică este $p[n_1][n_2] \dots [n_N]$ este $p + offset$, unde:

$$offset = n_N + dim_N \cdot \left(n_{N-1} + dim_{N-1} \cdot (n_{N-2} + \cdots)\right) = \sum_{i=1}^{N} \left(\prod_{j=i+1}^{N} dim_j \right) n_i$$