


Universitatea "Politehnica" din București
Facultatea de Electronică, Telecomunicații și
Tehnologia Informației



Structuri de Date și Algoritmi (limbajul C)

Curs 1 – Lucrul cu pointeri

Prof. Bogdan IONESCU

2015-2016

Bibliografie

[1] Curs* (<http://imag.pub.ro/~bionescu/>),
*mulțumiri Dl. Prof. Iulian NĂSTAC pentru partajarea materialelor sale de curs.

[2] Dumitru Iulian Năstac, "Structuri de date și algoritmi – Aplicații", Editura Printech, București, 2008;

[3] Îndrumarul de laborator (disponibil la laborator);

[4] alte surse, exemplu Internet ...

Curs Structuri de Date și Algoritmi, Prof. Bogdan IONESCU, 2015-2016

Cuprins

- 1.1. Introducere
- 1.2. Lucrul cu pointeri
- 1.3. Alocarea dinamică a memoriei

Curs Structuri de Date și Algoritmi, Prof. Bogdan IONESCU, 2015-2016

1/65

1.1. Introducere

Curs Structuri de Date și Algoritmi, Prof. Bogdan IONESCU, 2015-2016

2/65

Ce am studiat până acum? (la Programarea Calculatoarelor)

- Introducere sisteme de calcul și limbaje de programare;
- Bazele programării în limbajul C;
- Operatori și expresii în C;
- Structuri condiționale în C;
- Structuri repetitive în C;
- Tipuri de date compuse în C;
- Definirea funcțiilor și conceptul de recursivitate în C;
- Definirea și lucrul cu pointeri în C (introducere).

Curs Structuri de Date și Algoritmi, Prof. Bogdan IONESCU, 2015-2016

3/65

Programarea
Calculatoarelor

- familiarizare concept programare;
- familiarizare limbaj C (sintaxă și operații de bază).

```

int main()
{
    float v1[100], v2[100], suma;
    int i, dim;

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

    for (i=0; i<dim; i++)
    {
        printf("v1[%d], v2[%d]:", i, i);
        scanf("%f %f", &v1[i], &v2[i]);
    }
    suma=v1[0]+v2[0];

    for (i=1; i<dim; i++)
        suma+=v1[i]+v2[i];
    printf("suma este: %.2f", suma);
}
    
```

Curs Structuri de Date și Algoritmi, Prof. Bogdan IONESCU, 2015-2016

4/65

Structuri de Date și Algoritmi

ilustrare algoritm Divide et Impera

- folosirea limbajului C la rezolvarea unor probleme complexe;
- implementarea de structuri de date complexe (liste, stive, etc);
- implementarea de algoritmi (sortare, Divide et Impera, etc).

Curs Structuri de Date și Algoritmi, Prof. Bogdan IONESCU, 2015-2016 5/65

1.2. Lucrul cu pointeri

Curs Structuri de Date și Algoritmi, Prof. Bogdan IONESCU, 2015-2016 6/65

Ce reprezintă o variabilă ?

> **Variabilă** = un "container" sau o zonă de memorie în care sunt stocate anumite valori.

VarA

↓

nume

2009

→ valoarea variabilei

→ adresa de memorie

> Zona de memorie nu este oarecare, ci aceasta are o anumită adresă unică (un număr) pe baza căreia se poate face referință la ea.

> Variabila este identificată printr-un **nume** alfanumeric unic.

Curs Structuri de Date și Algoritmi, Prof. Bogdan IONESCU, 2015-2016 7/65

Ce reprezintă o variabilă ? (continuare)

VarA

↓

nume

2009

→ valoarea variabilei

→ adresa de memorie

Ce se întâmplă dacă scriem:

```
printf("%d", VarA);
```

>2009

VarA indică valoarea

Dar dacă scriem:

```
printf("%d", &VarA);
```

>2567843

&VarA indică adresa

Curs Structuri de Date și Algoritmi, Prof. Bogdan IONESCU, 2015-2016 8/65

Ce reprezintă un pointer ?

> **Un pointer este tot o variabilă** (o locație de memorie).

> Acesta diferă de variabilele clasice prin faptul că, în loc să stocheze valori, **acesta stochează doar adrese** ale locațiilor din memorie.

> Pointer = indicator, prin intermediul adresei pe care o stochează, putem spune că acesta indică spre conținutul unei alte locații de memorie.

PointerA

↓

2567843

2009

→ valoarea pointerului

→ adresa variabilei pointer

Curs Structuri de Date și Algoritmi, Prof. Bogdan IONESCU, 2015-2016 9/65

Avantaje lucru cu pointeri

> Pointerii constituie fundamentul limbajului C, aceștia oferind o multitudine de avantaje:

- **rapiditate în manipularea datelor**, exemplu: vrem să copiem valorile vectorului A în vectorul B, folosind pointerii este suficient să stocăm adresa primului element din A în adresa primului element din B,
- **eficientizare a memoriei**, variabilele definite ca pointeri pot fi alocate dinamic: se alocă **când** vrem, **cât** vrem și **cât timp** vrem (se poate elibera memoria),
- anumite calcule nu pot fi exprimate decât pe baza pointerilor,

Curs Structuri de Date și Algoritmi, Prof. Bogdan IONESCU, 2015-2016 10/65

Avantaje lucru cu pointeri (continuare)

> Avantaje (continuare):

- o funcție poate returna mai multe valori cu ajutorul pointerilor, returnând astfel adresa la care se găsesc acestea (de exemplu, adresa de început a unui vector)
- transmiterea valorilor prin adresă (pointer) în funcții permite modificarea valorilor acestora, ceea ce nu este valabil când parametrul funcției este transmis prin valoare (cazul clasic),
- etc.

> Principalul dezavantaj este dat manipularea acestora, ce **necesită anumite precauții**, anumite greșeli de manipulare nefiind sesizate de compilator!

Definirea și lucrul cu pointeri

> În limbajul C studiat până în acest punct, ați lucrat deja cu pointeri, și anume la:

- lucrul cu *vectori și matrice*,
- lucrul cu *șiruri de caractere*,
- lucrul cu *structuri de date și uniuni*,
- lucrul cu *funcții*.

> Modul de definire al unui pointer:

```
<tip de bază> *<nume pointer>;
```

> **Efect:** variabila <nume pointer> va fi un pointer ce indică o locație de memorie ce stochează valori de tip <tip de bază>.

Definirea și lucrul cu pointeri (continuare)

> Exemple:

```
int *PointerInt;  
double *PointerReal;
```

PointerInt va conține adresa unei locații de memorie ce stochează întregi (32 biți),
PointerReal va conține adresa unei locații de memorie ce stochează double (64 biți).

> **Atenție:** tipul datelor indicate este specificat tocmai pentru că acestea sunt stocate diferit, astfel nu putem suprapune un double peste int.

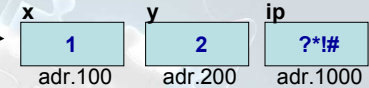
> Care sunt "ustensilele" de lucru cu pointerii:

- **operatorul unar &** = adresa unei variabile,
- **operatorul de indirectare *** = conținutul obiectului indicat de pointer (conținutul de la adresa stocată de pointer).

Definirea și lucrul cu pointeri (continuare)

> Exemplu utilizare pointeri:

```
int x=1, y=2;  
int *ip;
```



> Efect:

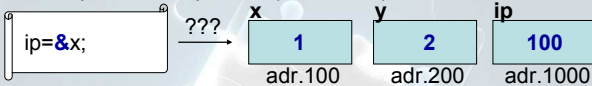
- la adresa variabilei **x** (ex. 100) am pus valoarea 1,
- la adresa lui **y** (ex. 200) am pus valoarea 2,
- la adresa pointerului **ip** (ex. 1000) am pus valoarea ???

> Când un pointer este declarat, **nu conține o adresă validă**, de regulă indică către o zonă de memorie care, de cele mai multe ori nici nu aparține programului.

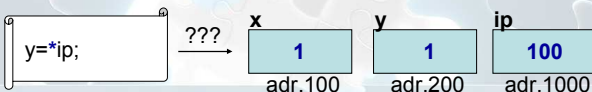
> După declarare, precum variabilele, **pointerul trebuie inițializat**.

Definirea și lucrul cu pointeri (continuare)

> Exemplu utilizare pointeri (continuare):



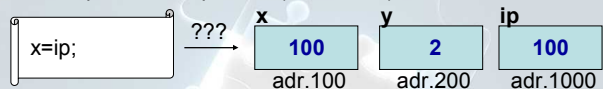
> **Traducere:** la adresa de memorie a variabilei pointer **ip** se va stoca adresa (dată de operatorul &) variabilei **x**.
→ pointerul **ip** indică acum către variabila **x**.



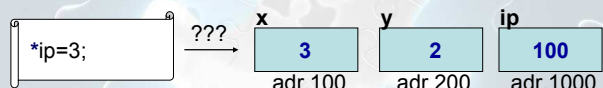
> **Traducere:** variabila **y** preia valoarea de la adresa stocată în pointerul **ip** sau cu alte cuvinte, conținutul obiectului indicat de **ip** (returnat de operatorul *).

Definirea și lucrul cu pointeri (continuare)

> Exemplu utilizare pointeri (continuare):



> **Traducere:** variabila **x** va lua valoarea variabilei pointer **ip**, și anume adresa conținută de aceasta, care este adresa lui **x**.



> **Traducere:** conținutul obiectului indicat de pointerul **ip**, și anume locația de memorie de la adresa 100 va primi valoarea 3.
→ variabila **x** are valoarea 3 (echivalent **x=3**).

Definirea și lucrul cu pointeri (continuare)

> **Pointeri nuli:** în anumite situații este necesară inițializarea cu "0" (~ adresă neutră), adică cu o adresă care nu indică niciunde (atenție != adresă invalidă):

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

> **Compararea pointerilor:** se pot compara doi pointeri printr-o expresie relațională dar comparația are sens doar dacă cei doi pointeri indică către elementele aceluiași tablou (matrice).

```
int *p = 0, *q = 0;  
if (p==q) printf("pointeri egal");
```

> pointerii nuli sunt întotdeauna egali.

Definirea și lucrul cu pointeri (continuare)

> Recapitulare:

ip înseamnă:
conținutul variabilei pointer ip și anume o adresă la o variabilă de tip întreg.

```
int *ip, x;  
ip=&x;
```

&ip înseamnă:
adresa din memorie a variabilei pointer ip și anume locația unde stochează adresele

***ip** înseamnă:
conținutul obiectului indicat de pointer și anume conținutul locației de memorie de la adresa memorată de pointer (adr. ip)

Definirea și lucrul cu pointeri (continuare)

P **Enunț:** să se realizeze un program ce permite interschimbarea valorilor a două variabile reale, a și b, prin intermediul a doi pointeri.

Variabile de intrare/lucru:	Structură program:
<code>float a,b;</code>	- se citesc a și b,
<code>float *pa,*pb;</code>	- se inițializează cei doi pointeri,
<code>float tmp;</code>	- se transferă conținutul indicat de primul pointer în al doilea.
Variabile de ieșire:	-

Definirea și lucrul cu pointeri (continuare)

```
float a,b,tmp;  
float *pa,*pb;  
  
pa=&a; //initializare pointer pa cu adresa lui a  
pb=&b; //initializare pointer pb cu adresa lui b  
  
printf("a="); scanf("%f",&a);  
printf("b="); scanf("%f",&b);  
  
tmp=*pa; //interschimbare valori  
*pa=*pb; //prin conținutul  
*pb=tmp; //indicat de pa si pb  
  
printf("a=%0.2f,b=%0.2f\n",a,b);
```

Definirea și lucrul cu pointeri (continuare)

> Un pointer este definit specificând tipul datelor pe care le indică – putem defini un pointer generic? (care să indice orice tip de date)

> Modul de definire al unui pointer generic:

```
void *<nume pointer>;
```

> **Efect:** variabila <nume pointer> va fi un pointer ce indică o locație de memorie a cărui tip de date ce va fi stocat nu este specificat (incă).

> **Avantaj:** flexibilitate în modul de definire, nefiind restricționat la un anumit tip de date, putând fi folosit practic cu orice tip.

Definirea și lucrul cu pointeri (continuare)

> Exemplu utilizare pointeri generici:

```
int x;  
double y;  
char c;  
void *p;  
  
p=&x;  
*p=10;  
  
p=&y;  
*p=4.5;  
  
p=&c;  
*p='x';
```

x este o variabilă întregă,
y o variabilă reală,
c o variabilă caracter.

p este un pointer generic.

p este inițializat cu adresa lui **x**,
x este inițializat cu **10**,
similar pentru **y** și **c** ...

Este o greșeală în program!

Definirea și lucrul cu pointeri (continuare)

> Un pointer definit ca void nu poate fi indirectat direct (ceea ce este oarecum evident, acesta neavând un tip definit) ci prin conversia la un pointer de un anumit tip specificat.

> *casting*: conversia temporară a unei date de un anumit tip, într-un tip de bază ce este specificat de utilizator.

Formă generală: `<tip_nou> <expresie>` sau C++ `<tip_nou>(<expresie>)`
 ex.: `x=float(y)/4;`

Corect ar fi:

```
int x;
...
p=&x;
*(int *)p=10;
```

p este inițializat cu adresa lui x.

(int *)p → p este convertit temporar la un pointer de tip int, și apoi indirectat cu *.

Definirea și lucrul cu pointeri (continuare)

> Exemplu utilizare pointeri generici (continuare): corect este

```
int x;
double y;
char c;
void *p;

p=&x;
*(int *)p=10;

p=&y;
*(double *)p=4.5;

p=&c;
*(char *)p='x';
```

x este o variabilă întreagă, y o variabilă reală, c o variabilă caracter.

p este un pointer generic.

p este convertit la int * și inițializat cu adresa lui x, x este inițializat cu 10 prin indirectarea lui p, similar pentru y și c ...

Aritmetica pointerilor

> Fiind variabile, pointerii permit efectuarea a o serie de operații specifice variabilelor.

Care este efectul acestor instrucțiuni ?

```
int *ip;
*ip=100;
```

ip adr. **?*!#**

100 adr. **?*!#**

> **Greșeală tipică**: un pointer trebuie **inițializat**, deoarece altfel este posibil să indice o locație inaccesibilă.

Corect ar fi:

```
int *ip, x;
ip=&x;
*ip=100;
```

ip adr. **B**

x 100 adr. **B**

Aritmetica pointerilor (continuare)

> Exemplu 1*:

```
char *mychar;
short *myshort;
long *mylong;
mychar=1000;
myshort=2000;
mylong=3000;
mychar++;
myshort++;
mylong++;
```

- mychar indică o locație de memorie pe 8 biți (1 byte),
- myshort indică o locație de memorie pe 16 biți (2 bytes),
- mylong indică o locație de memorie pe 32 biți (4 bytes).

[*http://www.cplusplus.com/doc/tutorial/pointers.html](http://www.cplusplus.com/doc/tutorial/pointers.html)

Aritmetica pointerilor (continuare)

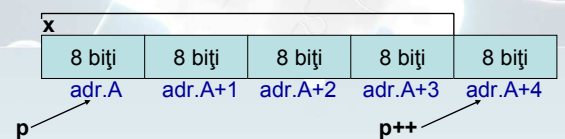
> Exemplu 2:

```
int *p, x;
p=&x;
*p++=10;
```

- p este un pointer care indică variabila întreagă x.

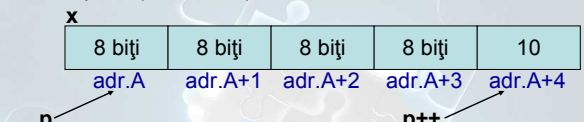
> **Atenție**: operatorii ++ și -- au prioritate mai mare decât operatorul * (vezi Cursul 5).

> Astfel, *p++ este echivalent cu *(p++).



Aritmetica pointerilor (continuare)

> Exemplu 2 (continuare)



> **Teoretic**, *(p++)=10 se traduce prin:

- **p** nu mai are ca valoare adresa lui **x** (**adr. A**) ci **p++**, și anume adresa **p+4bytes** (**adr.A+4**).

- la adresa stocată la **adr.A+4** se va pune valoarea 10.

> **Practic** am omis faptul că **p++ este o incrementare post**:

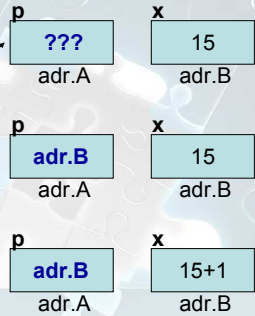
- la adresa stocată în **p** se va pune valoarea 10.

- după aceasta, **p** este incrementat și indică **adr.A+4**.

Aritmetica pointerilor (continuare)

> Exemplu 3:

```
int *p, x=15;
p=&x;
(*p)++;
```



> `(*p)++` conținutul locației de memorie indicate de `p` este mărit cu 1, echivalent cu `x++`.

Pointerii și matrice

> Exemplu 4:

```
int numere[5], n;
int *p;
p=numere;
*p=10;
p++;
*p=20;
p=&numere[2];
*p=30;
p=numere + 3;
*p=40;
p=numere;
*(p+4)=50;
for (n=0; n<5; n++)
    printf("%d, ", numere[n]);
```

32biți	32biți	32biți	32biți	32biți
1000	1004	1008	1012	1016

`p=1000` (`p` indică primul element al vectorului)

10	32biți	32biți	32biți	32biți
1000	1004	1008	1012	1016

`p=1000+4` (adresa indicată de `p` este incrementată cu o unitate, în acest caz un int de 32 de biți)

Pointerii și matrice (continuare)

> Exemplu 4 (cont.)

```
int numere[5], n;
int *p;
p=numere;
*p=10;
p++;
*p=20;
p=&numere[2];
*p=30;
p=numere + 3;
*p=40;
p=numere;
*(p+4)=50;
for (n=0; n<5; n++)
    printf("%d, ", numere[n]);
```

10	20	32biți	32biți	32biți
1000	1004	1008	1012	1016

`p=1004` (incrementat anterior)

`p`=adresa elementului de indice 2 din vector, `1008`.
`p` indică acum spre variabila `numere[2]`.

10	20	30	32biți	32biți
1000	1004	1008	1012	1016

Pointerii și matrice (continuare)

> Exemplu 4 (cont.)

```
int numere[5], n;
int *p;
p=numere;
*p=10;
p++;
*p=20;
p=&numere[2];
*p=30;
p=numere + 3;
*p=40;
p=numere;
*(p+4)=50;
for (n=0; n<5; n++)
    printf("%d, ", numere[n]);
```

`p` va primi ca valoare adresa primului element din vectorul `numere`, `1000`, la care se adaugă 3 unități de memorie, în acest caz 3×32 de biți. `p=1012`

10	20	30	40	32biți
1000	1004	1008	1012	1016

`p=1000`, operația `+` este prioritară față de `*`, astfel `p=1016`

10	20	30	40	50
1000	1004	1008	1012	1016

Pointerii și matrice (continuare)

> Exemplu 5

```
char i,j,a[2][4];
char *p;

p=a[0];

for (i=0;i<2;i++)
    for (j=0;j<4;j++)
        scanf("%d",&p[i*4+j]);
```

- am declarat o matrice `a` cu 2 linii și 3 coloane cu valori int (`i` și `j` sunt contori);
- am declarat un pointer `p` la int (char);

- `p` este inițializat cu adresa primului element din `a`;

- `i=0, j=0` → adresa $(p+0 \times 4+0)$

8 biți	8 biți	8 biți	8 biți
adr. N	adr. N+1	adr. N+2	adr. N+3
8 biți	8 biți	8 biți	8 biți
adr. N+4	adr. N+5	adr. N+6	adr. N+7

Pointerii și matrice (continuare)

> Exemplu 5 (cont.)

```
char i,j,a[2][4];
char *p;

p=a[0];

for (i=0;i<2;i++)
    for (j=0;j<4;j++)
        scanf("%d",&p[i*4+j]);
```

- am declarat o matrice `a` cu 2 linii și 3 coloane cu valori int (`i` și `j` sunt contori);
- am declarat un pointer `p` la int (char);

- `p` este inițializat cu adresa primului element din `a`;

- `i=0, j=1` → adresa $(p+0 \times 4+1)$...

8 biți	8 biți	8 biți	8 biți
adr. N	adr. N+1	adr. N+2	adr. N+3
8 biți	8 biți	8 biți	8 biți
adr. N+4	adr. N+5	adr. N+6	adr. N+7

Pointeri și matrice (continuare)

> Exemplu 5 (cont.)

```
char i,j,a[2][4];  
char *p;
```

- am declarat o matrice **a** cu 2 linii și 3 coloane cu valori int (i și j sunt contori);
- am declarat un pointer **p** la int (char);

```
p=a[0];
```

- **p** este inițializat cu adresa primului element din **a**;

```
for (i=0;i<2;i++)  
for (j=0;j<4;j++)  
scanf("%d",&a[i][j]);
```

- i=1, j=0 → adresa **(p+1*4+0)**

8 biți	8 biți	8 biți	8 biți
adr. N	adr. N+1	adr. N+2	adr. N+3
8 biți	8 biți	8 biți	8 biți
adr. N+4	adr. N+5	adr. N+6	adr. N+7

Pointeri și matrice (continuare)

> Exemplu 5 (cont.)

```
char i,j,a[2][4];  
char *p;
```

- am declarat o matrice **a** cu 2 linii și 3 coloane cu valori int (i și j sunt contori);
- am declarat un pointer **p** la int (char);

```
p=a[0];
```

- **p** este inițializat cu adresa primului element din **a**;

```
for (i=0;i<2;i++)  
for (j=0;j<4;j++)  
scanf("%d",&a[i][j]);
```

- i=1, j=1 → adresa **(p+1*4+1)**

8 biți	8 biți	8 biți	8 biți
adr. N	adr. N+1	adr. N+2	adr. N+3
8 biți	8 biți	8 biți	8 biți
adr. N+4	adr. N+5	adr. N+6	adr. N+7

Pointeri și matrice (continuare)

> Exemplu 5 (cont.): generalizare

```
<tip date> a[dim1];           a[i] ↔ *(p+i)  
<tip date> *p=&a[0];
```

```
<tip date> a[dim1][dim2];     a[i][j] ↔ *(p+i*dim2+j)  
<tip date> *p=&a[0][0];
```

```
<tip date> a[dim1][dim2][dim3]; a[i][j][k] ↔  
<tip date> *p=&a[0][0][0];     *(p+i*dim2*dim3+j*dim3+k)
```

> pointerii sunt adesea folosiți pentru a avea acces în matrice, aritmetica pointerilor este mai rapidă decât accesul prin indici.

Pointeri și matrice (continuare)

P **Enunț:** să se realizeze un program ce permite afișarea valorilor unui șir de caractere s prin intermediul unui pointer la char (pointerul este folosit pentru a indica fiecare element al șirului de caractere).

Variabile de intrare/lucru:

```
char s[256];  
char *pc;
```

Structură program:

- se citește șirul de caractere s,
- se inițializează pointer,
- se parcurge cu pointer fiecare locație din șirul de caractere s.

Variabile de ieșire:

-

Pointeri și matrice (continuare)

```
char s[256];  
char *pc;
```

```
pc=&s[0]; //initalizare pointer cu adresa primului element
```

```
printf("s="); //citire șir de caractere  
scanf("%s",s); //scanf asigura adaugarea \0 la final
```

```
while (*pc) //cat timp continutul indicat de pc este <> 0  
printf("%c",*pc++); //afiseaza caracter si incrementeaza adresa
```

Pointeri și funcții

> În general, în limbajul C, parametrii unei funcții sunt transmiși prin **valoare**.

→ funcția apelată primește o copie a variabilei, adică valoarea variabilei copiată la o altă adresă de memorie decât cea a variabilei originale

> Exemplu:

```
void schimba(int a, int b);
```

variabilele **a** și **b** sunt transmise prin valorile acestora, deci nu vor fi modificate de funcție.

> O altă modalitate de transmitere a parametrilor este prin **referință** sau **adresă**.

→ funcția apelată primește o copie a adresei variabilei ce conține parametrul efectiv (un pointer).

Pointeri și funcții (continuare)

> *Exemplu:*

```
void schimba(int *pa, int *pb);
```

funcția primește două valori de adrese ce corespund locațiilor la care se află variabilele ce vrem să le modificăm.

> *Detaliu funcție:*

```
void schimba(int *pa, int *pb)
{
    int tmp;
    tmp=*pa;
    *pa=*pb;
    *pb=tmp;
}
```

funcția interschimbă valorile a două variabile de tip întreg.

tmp preia conținutul locației de memorie indicată de **pa** (*pa),

conținutul lui **pa** (*pa) preia conținutul lui **pb** (*pb),

conținutul lui **pb** (*pb) ia valoarea **tmp**.

Curs Structuri de Date și Algoritmi, Prof. Bogdan IONESCU, 2015-2016 41/65

Pointeri și funcții (continuare)

> *Cum apelăm funcția din programul principal ?*

```
int main()
{
    int a=3, b=6;
    schimba(&a, &b);
}
```

```
void schimba(int *pa, int *pb)
{
    int tmp;
    tmp=*pa;
    *pa=*pb;
    *pb=tmp;
}
```

> *Să vedem ce se întâmplă în memorie ?*

1.

a	b	pa	pb
3	6	adr.a	adr.b
adr.a	adr.b	adr.pa	adr.pb

Curs Structuri de Date și Algoritmi, Prof. Bogdan IONESCU, 2015-2016 42/65

Pointeri și funcții (continuare)

```
int main()
{
    int a=3, b=6;
    schimba(&a, &b);
}
```

```
void schimba(int *pa, int *pb)
{
    int tmp;
    tmp=*pa;
    *pa=*pb;
    *pb=tmp;
}
```

2.

a	b	pa	pb	tmp
3	6	adr.a	adr.b	???
adr.a	adr.b	adr.pa	adr.pb	adr.tmp

3.

a	b	pa	pb	tmp
3	6	adr.a	adr.b	3
adr.a	adr.b	adr.pa	adr.pb	adr.tmp

Curs Structuri de Date și Algoritmi, Prof. Bogdan IONESCU, 2015-2016 43/65

Pointeri și funcții (continuare)

```
int main()
{
    int a=3, b=6;
    schimba(&a, &b);
}
```

```
void schimba(int *pa, int *pb)
{
    int tmp;
    tmp=*pa;
    *pa=*pb;
    *pb=tmp;
}
```

4.

a	b	pa	pb	tmp
6	6	adr.a	adr.b	3
adr.a	adr.b	adr.pa	adr.pb	adr.tmp

5.

a	b	pa	pb	tmp
6	3	adr.a	adr.b	3
adr.a	adr.b	adr.pa	adr.pb	adr.tmp

Curs Structuri de Date și Algoritmi, Prof. Bogdan IONESCU, 2015-2016 44/65

Pointeri și funcții (continuare)

P *Enunț:* să se realizeze o funcție de citire a unei matrice de numere reale ce permite citirea inclusiv a dimensiunilor acesteia și returnarea lor. Să se realizeze un program ce permite citirea și afișarea valorilor unei matrice de numere reale folosind această funcție.

Funcții:

```
void citireMat(float M[100][100], int *dim1, int *dim2);
void afisareMat(float M[100][100], int dim1, int dim2);
```

Variabile de intrare/lucru main(): *Variabile de ieșire:*

```
float M[100][100];
int dim1, dim2;
```

Curs Structuri de Date și Algoritmi, Prof. Bogdan IONESCU, 2015-2016 45/65

Pointeri și funcții (continuare)

```
void citireMat(float M[100][100], int *dim1, int *dim2)
{
    int i, j;
    printf("dim1=");
    scanf("%d", dim1); //dim1 reprezinta deja adresa

    printf("dim2=");
    scanf("%d", dim2); //dim2 reprezinta deja adresa

    for (i=0; i<*dim1; i++) /*dim1 continutul de la adresa lui dim1
    for (j=0; j<*dim2; j++) /*dim2 continutul de la adresa lui dim2
        scanf("%f", &M[i][j]);
}
```

Curs Structuri de Date și Algoritmi, Prof. Bogdan IONESCU, 2015-2016 46/65

Pointeri și funcții (continuare)

```
void afisareMat(float M[100][100], int dim1, int dim2)
{
    for (int i=0;i<dim1;i++)
    {
        for (int j=0;j<dim2;j++)
            printf("%12.2f",M[i][j]);
        printf("\n");
    }
}

int main()
{
    float M[100][100];
    int dim1, dim2;
    citireMat(M,&dim1,&dim2);
    afisareMat(M,dim1,dim2);
}
```

Pointeri la funcții

> Atunci când este compilată o funcție, codul sursă este transformat în cod obiect și se **stabilește un punct (adresă) de intrare în funcție**.

> Această adresă asociată funcției este folosită pentru apelarea funcției în program.

> *Idee*: poate o funcție atunci să fie folosită pe post de argument într-o altă funcție, ca orice variabilă?

Definire pointer la o funcție:

```
<tip data> (* <nume functie>) (<tip data>, <tip data>, ...);
```

Pointeri la funcții (continuare)

```
<tip data> (* <nume functie>) (<tip data>, <tip data>, ...);
```

tipul datelor returnate de funcție

numele funcției în paranteze, * specifică pointer

tipul datelor de intrare în paranteze separate de virgulă

vs.

```
<tip data> <nume functie>(<tip data> <nume var.>, ...);
```

tipul datelor returnate de funcție

numele funcției

variabilele de intrare în paranteze separate de virgulă

Pointeri la funcții (continuare)

Exemplu:

```
int (* functie1)(int,int);
```

> **Efect**: variabila *functie1* va fi un pointer ce indică către o funcție ce returnează o valoare întregă și primește la intrare două argumente întregi.

> Pentru a fi folosit, este necesară inițializarea pointerului *functie1*?

> Cum putem inițializa un pointer la o funcție?
(un pointer la o valoare era inițializat cu o adresă de memorie ce poate stoca acea valoare)

Pointeri la funcții (continuare)

Exemplu:

```
int adunare(int a, int b)
{
    return (a+b);
}
```

am definit funcția **adunare** care primește două valori întregi și returnează suma acestora.

```
int diferenta(int a, int b)
{
    return (a-b);
}
```

am definit funcția **diferență** care primește două valori întregi și returnează diferența acestora.

```
int calcul(int x, int y, int (*functie)(int,int))
{
    return (*functie)(x,y);
}
```

am definit funcția **calcul** care primește două valori întregi și un pointer, **functie**, la o funcție ce returnează un întreg și primește două valori întregi.

~ apelare funcție cu parametrii x și y

Pointeri la funcții (continuare)

Exemplu (continuare):

```
int main()
{
    int rezultat;
    int (*operatie)(int,int);

    operatie = adunare;
    rezultat = calcul(7, 5, operatie);
    printf("%d\n",rezultat);

    operatie = diferenta;
    rezultat = calcul(7, 5, operatie);
    printf("%d\n",rezultat);
}
```

rezultat este o variabilă întregă.

operatie este un pointer la o funcție ce returnează o valoare întregă și are ca argumente două valori întregi.

pointerul **operatie** este inițializat cu adresa funcției **adunare**.

```
int adunare(int a, int b);
int diferenta(int a, int b);
int calcul(int x, int y, int (*functie)(int,int));
```

Pointeri la funcții (continuare)

Exemplu (continuare):

```
int main()
{
    int rezultat;
    int (*operatie)(int,int);

    operatie = adunare;
    rezultat = calcul(7, 5, operatie);
    printf("%d\n",rezultat);

    operatie = diferenta;
    rezultat = calcul(7, 5, operatie);
    printf("%d\n",rezultat);
}
```

este apelată funcția **calcul** cu argumentele **a=7** și **b=5** și pointerul la funcție, **operatie**.

> 12

```
int adunare(int a, int b);
int diferenta(int a, int b);
int calcul(int x, int y, int (*functie)(int,int));
```

Curs Structuri de Date și Algoritmi, Prof. Bogdan IONESCU, 2015-2016 53/65

Pointeri la funcții (continuare)

Exemplu (continuare):

```
int main()
{
    int rezultat;
    int (*operatie)(int,int);

    operatie = adunare;
    rezultat = calcul(7, 5, operatie);
    printf("%d\n",rezultat);

    operatie = diferenta;
    rezultat = calcul(7, 5, operatie);
    printf("%d\n",rezultat);
}
```

pointerul **operatie** este initializat cu adresa funcției **diferenta**.

este apelată funcția **calcul** cu argumentele **a=7** și **b=5** și pointerul la funcție, **operatie**.

> 2

```
int adunare(int a, int b);
int diferenta(int a, int b);
int calcul(int x, int y, int (*functie)(int,int));
```

Curs Structuri de Date și Algoritmi, Prof. Bogdan IONESCU, 2015-2016 54/65

1.3. Alocarea dinamică a memoriei

Curs Structuri de Date și Algoritmi, Prof. Bogdan IONESCU, 2015-2016 55/65

Alocarea memoriei

> **alocare statică** (folosită până în acest punct al cursului): presupune alocarea "*permanentă*" a locațiilor de memorie necesare variabilelor și datelor programului la *momentul execuției* – acestea rămân alocate chiar dacă nu sunt folosite.

```
int V[100];
```

se alocă în memorie 100 de locații de 32 de biți.

> **alocare dinamică**: presupune alocarea de necesar de memorie pe *parcursul execuției* programului cu posibilitatea de eliberare a memoriei dacă nu este folosită.

> **motivație**: în majoritatea programelor nu se cunoaște de la început necesarul de memorie + eficientizare folosire memorie.

Curs Structuri de Date și Algoritmi, Prof. Bogdan IONESCU, 2015-2016 56/65

Alocarea memoriei (continuare)

> **alocarea memoriei**: funcția **malloc**

Prototipul funcției:

```
void *malloc(unsigned <nr_bytes>);
```

returnează un pointer de tip **void** la locația de memorie

specifică necesarul de memorie exprimat în bytes

Exemplu 1:

```
char *c;
c=(char * malloc(256));
```

- **c** este un pointer la date de tip caracter;
- se alocă **256** locații de 8 biți de memorie și se atribuie lui **c** adresa la primul byte.

Curs Structuri de Date și Algoritmi, Prof. Bogdan IONESCU, 2015-2016 57/65

Alocarea memoriei (continuare)

> **alocarea memoriei**: funcția **malloc** (continuare)

Exemplu 2:

```
double *x;
x=(double *) malloc(50*sizeof(double));
```

- **x** este un pointer la date de tip **double**;
- se alocă **50 x 8** locații de 8 biți de memorie și se atribuie lui **x** adresa primului byte.

> **sizeof()** - returnează dimensiunea în bytes a unei variabile sau tip de date:

```
int a;
a=sizeof(char);
```

a=1

```
int i;
double a;
i=sizeof(a);
```

i=8

Curs Structuri de Date și Algoritmi, Prof. Bogdan IONESCU, 2015-2016 58/65

Alocarea memoriei (continuare)

> alocarea memoriei: funcția **malloc** (continuare)

Ce se întâmplă dacă necesarul de memorie dorit nu este disponibil din anumite motive?

→ se returnează un pointer **NULL**;

Exemplu 3:

```
double *x;
x=(double *) malloc(50*sizeof(double));

if (!x)
    printf("Nu s-a putut aloca memoria");
```

x == NULL

Alocarea memoriei (continuare)

> eliberarea memoriei: funcția **free**

Prototipul funcției:

```
void free(void *<nume_pointer>);
```

nu returnează nimic

specifică un pointer de tip void la o zonă de memorie alocată anterior

Exemplu 4:

```
char *c;
c=(char *) malloc(256);
...
free(c);
```

- s-a alocat dinamic memorie pentru 256 de bytes referențiați prin **c**;
- după folosire sunt eliberați cu funcția **free**.

Alocarea memoriei (continuare)

Exemplu 5:

```
int *v, i, n;
printf("dimensiune vector=");
scanf("%d",&n);

v=(int *) malloc(n*sizeof(int));
for (i=0;i<n;i++)
{
    printf("v[%i]=",i);
    scanf("%d",&v[i]);
}
free(v);
```

v este un pointer la **int** iar **i** și **n** sunt variabile întregi;

se citește valoarea lui **n** (dimensiunea vectorului);

se alocă **n x 4** locații de memorie de 8 biți ce vor fi referențiate prin **v** care indică la primul byte;

sunt parcurse cele **n** locații și stocate în ele valori citite de la tastatură (~vector normal);

eliberare memorie **v**.

Problemă pointeri și alocare dinamică

P

Enunț: să se realizeze următoarele funcții:

- o funcție ce permite citirea de la tastatură a numărului de elemente ale unui vector de numere întregi, alocarea dinamică a memoriei pentru acesta, citirea valorilor vectorului și returnarea acestuia;
- o funcție de afișare a unui vector de numere întregi de dimensiune dată;
- o funcție care permite eliberarea memoriei pentru un vector de numere întregi specificat ca parametru de intrare;
- funcția **main()** care permite citirea unui vector, afișarea acestuia și eliberarea memoriei (cu funcțiile de mai sus).



Problemă pointeri și alocare dinamică (continuare)

```
int *creareVector(int *V, int *dim)
{
    printf("Introduceti dimensiune vector:"); scanf("%d",dim);

    V=(int *)malloc((*dim)*sizeof(int)); //alocare memorie *dim x int
    //dim - adresa, *dim - continut

    if (!V)
    { printf("Memoria nu a putut fi alocata!"); exit(1); }

    printf("\nCitire valori vector:\n");
    for (int i=0;i<*dim;i++)
    { printf("V[%d]=",i); scanf("%d",&V[i]); //valorile citite sunt stocate la
    //adresele V+i <-> V[i]
    }


    return V; //V este transmis prin valoare si trebuie returnat
}
```

Problemă pointeri și alocare dinamică (continuare)

```
void afisareVector(int *V, int dim)
{
    for (int i=0;i<dim;i++)
        printf("%d ",*(V+i)); //continutul de la adresa V+i
}

void eliberareVector(int *V)
{
    if (V!=NULL) free(V); //verificam daca V a fost alocat
}

int main ()
{
    int *V,dim;
    V=creareVector(V,&dim); //dim este transmis prin adresa
    afisareVector(V,dim); //dim este transmis prin valoare
    eliberareVector(V);
}
```



Sfârșitul Cursului 1