

# Limbaje de programare

Introducere. Recursivitate

Marius Minea

24 septembrie 2012

# Programarea necesită

*Atenție*

*Precizie* – limbajul are regulile lui

*Înțelegere* – *întrebați dacă e neclar!*

*Gândire* – rezolvăm probleme (noi)

*Exercițiu* – *acasă* și în laborator

## Despre limbajul C

dezvoltat în 1972 la *AT&T Bell Laboratories* de Dennis Ritchie asociat cu sistemul de operare UNIX și utilitarele acestuia  
(C dezvoltat inițial sub UNIX, apoi UNIX a fost rescris în C)  
Brian Kernighan, Dennis Ritchie: *The C Programming Language* (1978)

Limbaj vechi, dar în evoluție

standardul ANSI C, 1989 (American National Standards Institute)  
apoi standardul ISO 9899 (versiuni: C90, C99, **C11 - curentă**)

# Despre limbajul C

dezvoltat în 1972 la *AT&T Bell Laboratories* de Dennis Ritchie asociat cu sistemul de operare UNIX și utilitarele acestuia  
(C dezvoltat inițial sub UNIX, apoi UNIX a fost rescris în C)  
Brian Kernighan, Dennis Ritchie: *The C Programming Language* (1978)

Limbaj vechi, dar în evoluție

standardul ANSI C, 1989 (American National Standards Institute)  
apoi standardul ISO 9899 (versiuni: C90, C99, **C11 - curentă**)

## De ce folosim C?

*versatil*: acces direct la reprezentarea binară a datelor, libertate în lucrul cu memoria, bună interfață cu hardware

*matur*, bază mare de cod (biblioteci pentru multe scopuri)

*eficient*: compilatoare bune, generează cod compact, rapid

**ATENȚIE**: foarte ușor de făcut *erori*!

# Calcul, funcții și programe

## Un program

*citește* date de intrare

le *prelucrează* prin *calcul* (matematice)

*produce* (scrie) niște *rezultate*

# Calcul, funcții și programe

## Un program

*citește* date de intrare

le *prelucrează* prin *calcul* (matematice)

*produce* (scrie) niște *rezultate*

În matematică, exprimăm calculele prin *funcții*:

*cunoaștem* funcții predefinite (sin, cos, etc.)

*definim* funcții noi (pentru problema dată)

*combinăm* funcțiile existente și definite de noi

le *folosim* într-o anumită ordine

La fel folosim funcțiile în programare.

# Funcții în matematică și în C

Ridicarea la pătrat pentru întregi:

$$\text{sqr} : \mathbb{Z} \rightarrow \mathbb{Z}$$

$$\text{sqr}(x) = x \cdot x$$

tipul funcției    numele funcției    tipul și numele parametrului

```
int sqr(int x)
{
    return x * x;
}
```

# Funcții în matematică și în C

Ridicarea la pătrat pentru întregi:

$$\text{sqr} : \mathbb{Z} \rightarrow \mathbb{Z}$$

$$\text{sqr}(x) = x \cdot x$$

tipul funcției    numele funcției    tipul și numele parametrului

```
int sqr(int x)
{
    return x * x;
}
```

*Definiția* unei funcții conține:

*antetul* funcției, care specifică: domeniul de valori (întregi), numele funcției (sqr) și parametrii acesteia (întregul x)

*corpul* funcției, cu { } : aici, o singură *instrucțiune* (return), cu o *expresie* care dă valoarea funcției (pornind de la parametri)

Limbajul are *reguli* precise de scriere (*sintaxa*):

elementele se scriu într-o anumită *ordine*;

se folosesc *separatori* pentru a le delimita precis: ( ) ; { }



## O a doua funcție

Ridicarea la pătrat pentru numere *reale*

$sqrf : \mathbb{R} \rightarrow \mathbb{R}$	<code>float sqrf(float x)</code>
	<code>{</code>
$sqrf(x) = x \cdot x$	<code>    return x * x;</code>
	<code>}</code>

Alt domeniu de definiție și de valori (reali)  $\Rightarrow$  altă funcție  
(chiar și operația  $*$  e alta, fiind definită pe altă mulțime)

Pentru a o deosebi în program de `sqr` trebuie să-i dăm alt nume.

## O a doua funcție

Ridicarea la pătrat pentru numere *reale*

$sqr\!f : \mathbb{R} \rightarrow \mathbb{R}$	<pre>float sqr\!f(float x) {     return x * x; }</pre>
$sqr\!f(x) = x \cdot x$	

Alt domeniu de definiție și de valori (reali)  $\Rightarrow$  altă funcție  
(chiar și operația  $*$  e alta, fiind definită pe altă mulțime)

Pentru a o deosebi în program de `sq` trebuie să-i dăm alt nume.

Cuvintele `int`, `float` denotă *tipuri*.

Un *tip* e o *mulțime de valori* împreună cu un *set de operații* permise pentru aceste valori.

Pentru reali, e preferabil tipul `double` (dublă precizie)  
(folosit și de funcțiile standard: `sin`, `cos`, `exp`, etc.)

# Întregi și reali

Tipurile numerice diferă între C și matematică.

În matematică,  $\mathbb{Z} \subset \mathbb{R}$ , ambele sunt infinite,  $\mathbb{R}$  e densă.

În C, `int`, `float`, `double` sunt tipuri finite; realii au precizie finită.

*Constantele* numerice au tip determinat de modul de scriere:

2 e un întreg, 2.0 e un real

putem scrie un real în notație științifică: 1.0e-3 în loc de 0.001

sunt echivalente scrierile 1.0 și 1. respectiv 0.1 și .1

## Operații matematice

+ - \* /      Înmulțirea trebuie scrisă explicit!  
nu putem scrie  $2x$ , ci  $2 * x$  (sau  $x * 2$ )

Unele operații sunt diferite pentru întregi și reali:

*Împărțirea întreagă* e *împărțire cu rest* !!!

$7 / 2$  dă valoarea 3, dar  $7.0 / 2.0$  dă valoarea 3.5

$-7 / 2$  dă valoarea -3, la fel cu  $-(7 / 2)$

## Operații matematice

+ - \* /      Înmulțirea trebuie scrisă explicit!  
nu putem scrie  $2x$ , ci  $2 * x$  (sau  $x * 2$ )

Unele operații sunt diferite pentru întregi și reali:

*Împărțirea întreagă* e *împărțire cu rest* !!!

$7 / 2$  dă valoarea 3, dar  $7.0 / 2.0$  dă valoarea 3.5

$-7 / 2$  dă valoarea -3, la fel cu  $-(7 / 2)$

Operatorul *modulo* (scris %) e definit doar pentru întregi.

$$\begin{array}{l|l|l|l} 9 / 5 = 1 & 9 \% 5 = 4 & 9 / -5 = -1 & 9 \% -5 = 4 \\ -9 / 5 = -1 & -9 \% 5 = -4 & -9 / -5 = 1 & -9 \% -5 = -4 \end{array}$$

Semnul restului e același cu semnul deîmpărțitului.

Ecuția împărțirii cu rest:  $a = a / b * b + a \% b$

## Puțină terminologie

*Cuvinte cheie*: au un înțeles predefinit (nu poate fi schimbat)

Exemple: instrucțiuni (`return`), tipuri (`int`, `float`, `double`), etc.

## Puțină terminologie

*Cuvinte cheie*: au un înțeles predefinit (nu poate fi schimbat)

Exemple: instrucțiuni (`return`), tipuri (`int`, `float`, `double`), etc.

*Identificatori* (de ex. `sqx`, `x`) aleși de programator pentru a denumi funcții, parametri, variabile, etc.

Un identificator e o secvență de caractere formată din litere (mari și mici), liniuța de subliniere `_` și cifre, care nu începe cu o cifră și nu este un cuvânt cheie.

Exemple: `x3`, `a12_34`, `_exit`, `main`, `printf`, `int16_t`

## Puțină terminologie

*Cuvinte cheie*: au un înțeles predefinit (nu poate fi schimbat)

Exemple: instrucțiuni (`return`), tipuri (`int`, `float`, `double`), etc.

*Identificatori* (de ex. `sqx`, `x`) aleși de programator pentru a denumi funcții, parametri, variabile, etc.

Un identificator e o secvență de caractere formată din litere (mari și mici), liniuța de subliniere `_` și cifre, care nu începe cu o cifră și nu este un cuvânt cheie.

Exemple: `x3`, `a12_34`, `_exit`, `main`, `printf`, `int16_t`

*Constante* (întreg: `-2`; real: `3.14`; caracter: `'a'`, șir: `"a"`)



## Puțină terminologie

*Cuvinte cheie*: au un înțeles predefinit (nu poate fi schimbat)

Exemple: instrucțiuni (`return`), tipuri (`int`, `float`, `double`), etc.

*Identificatori* (de ex. `sqr`, `x`) aleși de programator pentru a denumi funcții, parametri, variabile, etc.

Un identificator e o secvență de caractere formată din litere (mari și mici), liniuța de subliniere `_` și cifre, care nu începe cu o cifră și nu este un cuvânt cheie.

Exemple: `x3`, `a12_34`, `_exit`, `main`, `printf`, `int16_t`

*Constante* (întreg: `-2`; real: `3.14`; caracter: `'a'`, șir: `"a"`)

*Semne de punctuație*, cu diverse semnificații:

- \* e un operator

- ; delimitează sfârșitul unei instrucțiuni

- parantezele ( ) grupează parametrii funcției sau o subexpresie

- acoladele { } grupează instrucțiuni sau declarații

## Funcții cu mai mulți parametri

Exemplu: discriminantul ecuației de gradul II:  $a \cdot x^2 + b \cdot x + c = 0$

```
float discrim(float a, float b, float c)
{
    return b * b - 4 * a * c;
}
```

Între parantezele rotunde ( ) din antetul funcției putem specifica oricâți parametri, fiecare cu tipul propriu, separați prin virgulă.

## Apelul de funcție

Până acum, am *definit* funcții, fără să le folosim.

Valoarea unei funcții poate fi *folosită* într-o expresie.

Sintaxa: ca în matematică:  $funcție(param, param, \dots, param)$

Exemplu: în discriminant, puteam folosi funcția `sqr`:

```
return sqr(b) - 4 * a * c;
```

## Apelul de funcție

Până acum, am *definit* funcții, fără să le folosim.

Valoarea unei funcții poate fi *folosită* într-o expresie.

Sintaxa: ca în matematică:  $funcție(param, param, \dots, param)$

Exemplu: în discriminant, puteam folosi funcția `sqrf`:

```
return sqrf(b) - 4 * a * c;
```

Sau, folosind funcția `sqr` dinainte putem defini:

```
int cube(int x)
{
    return x * sqr(x);
}
```

## Apelul de funcție

Până acum, am *definit* funcții, fără să le folosim.

Valoarea unei funcții poate fi *folosită* într-o expresie.

Sintaxa: ca în matematică:  $funcție(param, param, \dots, param)$

Exemplu: în discriminant, puteam folosi funcția `sqrf`:

```
return sqrf(b) - 4 * a * c;
```

Sau, folosind funcția `sqr` dinainte putem defini:

```
int cube(int x)
{
    return x * sqr(x);
}
```

**IMPORTANT:** înainte de a folosi orice identificator (nume) în C, el trebuie să fie *declarat* (trebuie să știm ce reprezintă)

⇒ Exemplele sunt corecte dacă `sqrf` respectiv `sqr` sunt definite *înainte* de `discrim`, respectiv `cube` în program.

## Un prim program C

```
int main(void)
{
    return 0;
}
```

Cel mai mic program: nu face nimic !

Orice program conține funcția *main* și e executat prin apelarea ei (programul poate conține și alte funcții)

În acest caz: funcția nu are parametri (*void*)

*void* e un cuvânt cheie pentru tipul vid (fără nici un element)

În standard: *main* returnează sistemului de operare un cod întreg (convenție: 0 pt. terminare cu succes,  $\neq$  0 pt. cod de eroare)

## Un program comentat

```
/* Acesta este un comentariu */  
int main(void) // comentariu pana la capat de linie  
{  
    /* Acesta e un comentariu pe mai multe linii  
       obisnuit, aici vine codul programului */  
    return 0;  
}
```

Programele pot conține comentarii, înscrise între /\* și \*/  
sau începând cu // și terminându-se la capătul liniei

Orice conținut între aceste caractere nu are nici un efect asupra  
generării codului și execuției programului

## Un program comentat

```
/* Acesta este un comentariu */  
int main(void) // comentariu pana la capat de linie  
{  
    /* Acesta e un comentariu pe mai multe linii  
       obisnuit, aici vine codul programului */  
    return 0;  
}
```

Programele pot conține comentarii, înscrise între /\* și \*/  
sau începând cu // și terminându-se la capătul liniei

Orice conținut între aceste caractere nu are nici un efect asupra  
generării codului și execuției programului

Programele *trebuie* comentate

pentru ca un cititor să le înțeleagă (alții, sau noi, mai târziu)  
ca documentație și specificație: funcționalitate, restricții, etc.  
ce reprezintă parametrii funcțiilor, rezultatul, variabilele,  
ce condiții trebuie îndeplinite, cum se comportă la eroare



## Tipărirea (scrierea)

```
#include <stdio.h>
int main(void)
{
    printf("hello, world!\n");    // tipareste un text
    return 0;
}
```

`printf` (de la "print formatted"): o funcție standard

nu este *instrucțiune* sau *cuvânt cheie*

e apelată aici cu un parametru șir de caractere

constantele șir de caractere: incluse între ghilimele " "

`\n` este notația pentru caracterul de linie nouă

## Tipărirea (scrierea)

```
#include <stdio.h>
int main(void)
{
    printf("hello, world!\n");    // tipareste un text
    return 0;
}
```

`printf` (de la "print formatted"): o funcție standard

nu este *instrucțiune* sau *cuvânt cheie*

e apelată aici cu un parametru șir de caractere

constantele șir de caractere: incluse între ghilimele " "

`\n` este notația pentru caracterul de linie nouă

Prima linie e o *directivă de preprocesare*, include fișierul `stdio.h` cu *declarațiile* funcțiilor standard de intrare / ieșire

*Declarația* = tip, nume, parametri: necesare pentru folosire

*Implementarea* (codul obiect, compilat): într-o bibliotecă din care compilatorul ia cele necesare la generarea programului executabil

## Tipărirea de numere

```
#include <math.h>
#include <stdio.h>
int main(void)
{
    printf("cos(0) = ");
    printf("%f", cos(0));
    return 0;
}
```

```
#include <stdio.h>
int sqr (int x) { return x * x; }
int main(void)
{
    printf("2 ori -3 la patrat e ");
    printf("%d", 2 * sqr(-3));
    return 0;
}
```

Pentru a tipări valoarea unei expresii, `printf` ia două argumente:

– un șir de caractere (specificator de format):

`%d` (întreg, *decimal*), `%f` (real, *floating point*)

– expresia, al cărei tip trebuie să fie compatibil cu cel indicat (verificarea cade în sarcina programatorului !!!)

**Secvențierea:** instrucțiunile unei funcții se execută *una după alta*

Excepții: instrucțiunea `return` încheie execuția funcției (după ea nu se mai execută nimic)

## Funcții definite pe cazuri

$$abs : \mathbb{Z} \rightarrow \mathbb{Z} \quad abs(x) = \begin{cases} x & x \geq 0 \\ -x & \text{altfel} \end{cases}$$

Cu cele învățate pâna acum, nu putem defini această funcție în C.

Valoarea funcției nu e dată de o singură expresie, ci de una din două expresii diferite ( $x$  sau  $-x$ ), depinzând de o condiție ( $x \geq 0$ )

$\Rightarrow$  În limbaj, trebuie să putem *decide* valoarea luată de expresie în funcție de valoarea unei *condiții* (adevărat/fals)

## Operatorul condițional ? : în C

*Expresia condițională* în C are sintaxa: *condiție ? expr1 : expr2*

- dacă condiția e adevărată, se evaluează doar *expr1*, și întregă expresie ia valoarea acesteia
- dacă e falsă, se evaluează doar *expr2* și întregă expresie ia valoarea acesteia

```
int abs(int x)
{
    return x >= 0 ? x : -x;        // operator minus unar
}
```

Operatorii de comparație: == (egalitate), != (diferit), <, <=, >, >=

IMPORTANT! Testul de egalitate în C e == și nu = simplu !!!

Funcția abs există ca funcție standard, declarată în `stdlib.h`

## Funcții definite pe mai mult de două cazuri

$$\operatorname{sgn} : \mathbb{Z} \rightarrow \{-1, 0, 1\} \quad \operatorname{sgn}(x) = \begin{cases} -1 & x < 0 \\ 0 & x = 0 \\ 1 & x > 0 \end{cases}$$

Nu putem transcrie funcția direct în C

(operatorul condițional permite doar decizia cu *două* ramuri (adevărat/fals), nu cu mai multe condiții / ramuri)

⇒ trebuie să descompunem decizia asupra valorii lui  $x$

⇒ *descompunerea în subprobleme* mai mici:

principiu *foarte important* în rezolvarea de probleme

Rescriem funcția cu o singură decizie în fiecare punct:

$$\operatorname{sgn}(x) = \begin{cases} \text{dacă } x < 0 & -1 \\ \text{altfel } (x \geq 0) & \begin{cases} \text{dacă } x = 0 & 0 \\ \text{altfel } (x > 0) & 1 \end{cases} \end{cases}$$

## Scrierea unei funcții pe mai multe cazuri în C

$$\text{sgn}(x) = \begin{cases} \text{dacă } x < 0 & -1 \\ \text{altfel } (x \geq 0) & \begin{cases} \text{dacă } x = 0 & 0 \\ \text{altfel } (x > 0) & 1 \end{cases} \end{cases}$$

```
int sgn (int x)
{
    return x < 0 ? -1
           : x == 0 ? 0 : 1;
}
```

Putem grupa arbitrar de mulți operatori `? :`  
*expr1* și *expr2* pot fi la rândul lor expresii condiționale  
O expresie scrisă corect are un `:` pentru fiecare `?`

## Descompunerea în subprobleme mai simple

Minimul a două numere reale se poate scrie simplu:

```
double min2(double x, double y) {  
    return x < y ? x : y;  
}
```

Pentru minimul a *trei* numere, comparațiile directe se complică:

$$\min3(x, y, z) = \begin{cases} \text{dacă } x < y & \begin{cases} \text{dacă } x < z & \mathbf{x} \\ \text{altfel } (x \geq z) & \mathbf{z} \end{cases} \\ \text{altfel } (x \geq y) & \begin{cases} \text{dacă } y < z & \mathbf{y} \\ \text{altfel } (y \geq z) & \mathbf{z} \end{cases} \end{cases}$$

Se repetă structura funcției `min2`  $\Rightarrow$  putem gândi mai simplu:  
Rezultatul e minimul între minimul primelor două și al treilea.  
 $\Rightarrow$  folosim direct funcția dinainte *fără a mai despărți* pe cazuri.

```
double min3(double x, double y, double z) {  
    return min2(min2(x, y), z);    // sau min2(x, min2(y,z))  
}
```



## Să înțelegem: apelul de funcție

Programul dat calculează  $x^6 = (x \cdot x^2)^2$

```
#include <stdio.h>
int sqr(int x) {
    printf("Patratul lui %d e %d\n", x, x*x);
    return x * x;
}
int main(void) {
    printf("2 la a 6-a e %d\n", sqr(2 * sqr(2)));
    return 0;
}
```

În ce ordine se scrie pe ecran ?

Patratul lui 2 e 4

Patratul lui 8 e 64

2 la a 6-a e 64

## Să înțelegem: apelul de funcție

În C, transmiterea parametrilor la funcții se face *prin valoare*.  
se *evaluatează* (calculează valoarea) toate argumentele funcției  
valorile se atribuie la *parametrii formali* (numele din def. fct.)  
apoi se începe execuția funcției cu aceste valori

## Să înțelegem: apelul de funcție

În C, transmiterea parametrilor la funcții se face *prin valoare*.  
se *evaluatează* (calculează valoarea) toate argumentele funcției  
valorile se atribuie la *parametrii formali* (numele din def. fct.)  
apoi se începe execuția funcției cu aceste valori

Programul începe cu execuția lui `main`, deci apelul la `printf`  
Funcția `printf` are nevoie de valoarea argumentelor sale.

valoarea primului argument se știe (o *constantă șir*)

pentru al doilea argument trebuie apelat: `sqr(2 * sqr(2))`

la rândul lui, `sqr` exterior are nevoie de valoarea argumentului  
`2 * sqr(2)`, pentru care trebuie apelat întâi `sqr(2)`

⇒ ordinea apelurilor: `sqr(2)`, apoi `sqr(8)`, apoi `printf` din `main`

## Să înțelegem CORECT apelul de funcție

Cum **NU se face** în C (deși ne-am putea închipui ...)

Funcția **NU** începe execuția fără să aibă argumentele calculate  
printf ar tipări 2 la puterea 6 e , apoi îi trebuie valoarea  
ar apela sqrt exterior care scrie Patratul lui, apoi îi trebuie x  
ar apela sqrt(2) care scrie Patratul lui 2 e 4, returnează 4,  
etc.

## Să înțelegem CORECT apelul de funcție

Cum **NU se face** în C (deși ne-am putea închipui ...)

Funcția **NU** începe execuția fără să aibă argumentele calculate  
printf ar tipări 2 la puterea 6 e , apoi îi trebuie valoarea  
ar apela sqrt exterior care scrie Patratul lui, apoi îi trebuie x  
ar apela sqrt(2) care scrie Patratul lui 2 e 4, returnează 4,  
etc.

**NU** se substituie *expresiile* argument pentru parametrii funcției  
din printf s-ar apela sqrt exterior cu *expresia* 2 \* sqrt(2)  
pentru (2\*sqrt(2))\*(2\*sqrt(2)) s-ar apela sqrt(2) de două ori

⇒ În C, o funcție calculează numai cu *valori*, niciodată cu *expresii*

Recursivitate

## Recursivitate: definiție, exemple

Din matematică cunoaștem *șiruri recurente*:

progresie aritmetică:

$$\begin{cases} x_0 = b & \text{(adică: } x_n = b \text{ pentru } n = 0) \\ x_n = x_{n-1} + r & \text{pentru } n > 0 \end{cases}$$

Exemplu: 1, 4, 7, 10, 13, ... ( $b = 1$ ,  $r = 3$ )

## Recursivitate: definiție, exemple

Din matematică cunoaștem *șiruri recurente*:

progresie aritmetică:

$$\begin{cases} x_0 = b & \text{(adică: } x_n = b \text{ pentru } n = 0) \\ x_n = x_{n-1} + r & \text{pentru } n > 0 \end{cases}$$

Exemplu: 1, 4, 7, 10, 13, ... ( $b = 1$ ,  $r = 3$ )

progresie geometrică:

$$\begin{cases} x_0 = b & \text{(adică: } x_n = b \text{ pentru } n = 0) \\ x_n = x_{n-1} \cdot r & \text{pentru } n > 0 \end{cases}$$

Exemplu: 3, 6, 12, 24, 48, ... ( $b = 3$ ,  $r = 2$ )

Nu calculează  $x_n$  *direct*, ci *din aproape în aproape*, folosind  $x_{n-1}$ .



## Recursivitate: definiție, exemple

Din matematică cunoaștem *șiruri recurente*:

progresie aritmetică:

$$\begin{cases} x_0 = b & \text{(adică: } x_n = b \text{ pentru } n = 0) \\ x_n = x_{n-1} + r & \text{pentru } n > 0 \end{cases}$$

Exemplu: 1, 4, 7, 10, 13, ... ( $b = 1$ ,  $r = 3$ )

progresie geometrică:

$$\begin{cases} x_0 = b & \text{(adică: } x_n = b \text{ pentru } n = 0) \\ x_n = x_{n-1} \cdot r & \text{pentru } n > 0 \end{cases}$$

Exemplu: 3, 6, 12, 24, 48, ... ( $b = 3$ ,  $r = 2$ )

Nu calculează  $x_n$  *direct*, ci *din aproape în aproape*, folosind  $x_{n-1}$ .

O noțiune e *recursivă* dacă e *folosită în propria sa definiție*.

**Exercițiu:** scrieți relațiile pentru: combinări  $C_n^k$ , șirul Fibonacci, ...

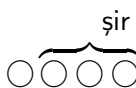
## Recursivitate: definiție, exemple

Recursivitatea e fundamentală în informatică:

reduce o problemă la un caz mai simplu al *aceleiași* probleme

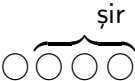
*obiecte*: un *șir* e  $\left\{ \begin{array}{l} \text{un singur element} \quad \bigcirc \\ \text{un element urmat de un } \textit{șir} \end{array} \right.$

ex. cuvânt (șir de litere); număr (șir de cifre zecimale)

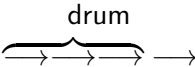


# Recursivitate: definiție, exemple

Recursivitatea e fundamentală în informatică:  
reduce o problemă la un caz mai simplu al *aceleiași* probleme

*obiecte*: un *șir* e  $\left\{ \begin{array}{l} \text{un singur element} \quad \bigcirc \\ \text{un element urmat de un } \mathbf{\textit{șir}} \end{array} \right.$  

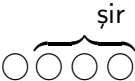
ex. cuvânt (șir de litere); număr (șir de cifre zecimale)

*acțiuni*: un *drum* e  $\left\{ \begin{array}{l} \text{un pas} \quad \longrightarrow \\ \text{un } \mathbf{\textit{drum}} \text{ urmat de un pas} \end{array} \right.$  

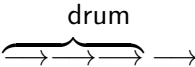
ex. parcurgerea unei căi într-un graf

# Recursivitate: definiție, exemple

Recursivitatea e fundamentală în informatică:  
reduce o problemă la un caz mai simplu al *aceleiași* probleme

*obiecte*: un *șir* e  $\left\{ \begin{array}{l} \text{un singur element} \quad \bigcirc \\ \text{un element urmat de un } \mathbf{\textit{șir}} \end{array} \right.$  

ex. cuvânt (șir de litere); număr (șir de cifre zecimale)

*acțiuni*: un *drum* e  $\left\{ \begin{array}{l} \text{un pas} \quad \longrightarrow \\ \text{un } \mathbf{\textit{drum}} \text{ urmat de un pas} \end{array} \right.$  

ex. parcurgerea unei căi într-un graf

O *expresie*:  $\left\{ \begin{array}{l} \text{număr (7)} \\ \text{identificator (x)} \\ \text{expresie + expresie} \\ \text{expresie - expresie} \\ \text{( expresie ), etc} \end{array} \right.$

## Exemplu: funcția putere

$$x^n = \begin{cases} 1 & n = 0 \\ x \cdot x^{n-1} & \text{altfel } (n > 0) \end{cases}$$

```
#include <stdio.h>
double pwr(double x, unsigned n) {
    return n==0 ? 1 : x * pwr(x, n-1);
}
int main(void) {
    printf("-2 la 3 = %f\n", pwr(-2.0, 3));
    return 0;
}
```

## Exemplu: funcția putere

$$x^n = \begin{cases} 1 & n = 0 \\ x \cdot x^{n-1} & \text{altfel } (n > 0) \end{cases}$$

```
#include <stdio.h>
double pwr(double x, unsigned n) {
    return n==0 ? 1 : x * pwr(x, n-1);
}
int main(void) {
    printf("-2 la 3 = %f\n", pwr(-2.0, 3));
    return 0;
}
```

Tipul `unsigned` reprezintă întregi fără semn (numere naturale)

*Antetul funcției* `pwr` reprezintă o *declarație* a ei  
deci putem folosi funcția în propriul corp (apelul recursiv)

Chiar dacă scriem `pwr(-2, 3)`, *întregul* `-2` va fi *convertit la real*,  
(se cunoaște tipul declarat pentru fiecare parametru)

## Mecanismul apelului recursiv

Funcția `pwr` face două calcule:

- un *test* (`n == 0` ? *cazul de bază* ?) dacă da, returnează 1
- altfel, o *înmulțire*; operandul drept necesită un *nou apel, recursiv*

```
pwr(5, 3)
  apel↓↑125
    5 * pwr(5, 2)
      apel↓↑25
        5 * pwr(5, 1)
          apel↓↑5
            5 * pwr(5, 0)
              apel↓↑1
                1
```

## Mecanismul apelului recursiv (cont.)

În calculul recursiv al funcției putere:

Fiecare apel face “*în cascadă*” *un nou apel*, până la cazul de bază

Fiecare apel execută *același cod*, dar cu *alte date*  
(valori proprii pentru parametri)

Ajunși la cazul de bază, toate apelurile *începute* sunt încă  
*neterminate*  
(fiecare mai are de făcut înmulțirea cu rezultatul apelului efectuat)

Revenirea se face *în ordine inversă* apelării  
(apelul cu exponent 0 revine primul, apoi cel cu exponent 1, etc.)