

Programarea calculatoarelor

Recursivitate

Marius Minea

6 martie 2007

Să înțelegem: apelul de funcție

```
#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;
```

```
}
```

$$x^6 = (x \cdot x^2)^2$$

În ce ordine se
scrie pe ecran ?

Patratul lui 2 e 4

Patratul lui 8 e 64

2 la a 6-a e 64

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 exemplu: programul începe cu execuția lui `main`, deci tipărirea `printf`

- `printf` are nevoie de valoarea argumentelor sale. Prima se știe (o *constantă șir*), a doua trebuie *calculată*: `sqr(2 * sqr(2))`
- pentru a efectua apelul *exterior* al lui `sqr` trebuie știut argumentul, adică `2 * sqr(2)`. Deci se efectuează întâi apelul *interior*, `sqr(2)`

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

Cum s-ar mai putea altfel, dar **NU se face** în C:

NU: funcția începe execuția și își calculează argumentele la nevoie

- `printf` ar tipări întâi `2` la puterea `6` e, apoi îi trebuie valoarea
- ar apela `sqr` exterior care scrie Patratul lui, apoi îi trebuie `x`
- ar apela `sqr(2)` care scrie Patratul lui `2` e `4`, returnează `4`, etc.

NU: se substituie *expresiile* argument pentru parametrii funcției

- din `printf` s-ar apela `sqr` exterior cu *expresia* `2 * sqr(2)`
- pt. a calcula `(2*sqr(2))*(2*sqr(2))` s-ar apela `sqr(2)` de două ori

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

Recursivitate: Noțiuni fundamentale

Recursivitatea e un concept fundamental în matematică și informatică. Un obiect (noțiune) e recursiv(ă) dacă e *folosit în propria sa definiție*.

Exemplu din matematică: șiruri recurente:

- progresie aritmetică: $x_0 = a$, $x_n = x_{n-1} + p$, pentru $n > 0$
- progresie geometrică: $x_0 = b$, $x_n = a \cdot x_{n-1}$, pentru $n > 0$
- ș.a.m.d.: combinări C_n^k , șirul lui Fibonacci, ...

Alte exemple:

- *obiecte* definite recursiv:

un șir e un singur element, sau un element urmat de un șir
ex.: cuvânt (șir de litere); număr (șir de cifre zecimale)

- *acțiuni* definite recursiv:

un drum e un pas, sau un drum urmat de încă un pas
(de exemplu o cale într-un graf)

Elementele unei definiții recursive

1. cazul de bază (conditia de oprire)
 - cel mai simplu caz pentru definiția (noțiunea) respectivă. Exemple:
 - termenul inițial dintr-un șir recurent
 - cel mai mic obiect (un element, în cazul șirului)
 - cea mai simplă acțiune (un pas, în cazul drumului)
2. relația de recurență
 - definește noțiunea, folosind un caz mai simplu al aceleiași noțiuni
3. demonstrație (argument) de oprire a definiției după nr. finit de pași (ex. o cantitate nenegativă care descrește urmărind definiția)
 - pentru șiruri: indicele (scade în definiție, e nenegativ)
 - pentru obiecte: dimensiunea (la fel, scade)

Sunt recursive, și corecte, următoarele definiții ?

- $x_{n+1} = 2 \cdot x_n$
- $x_n = x_{n+1} - 3$
- $a^n = a \cdot a \cdot \dots \cdot a$ (de n ori)
- o frază e o înșiruire de cuvinte
- un șir e un șir mai mic urmat de un alt șir mai mic
- un șir e un caracter urmat de un șir

O definiție recursivă trebuie să fie *bine formată* (v. condițiile 1-3)

- ceva nu se poate defini doar în funcție de sine însuși ($x = f(x)$)
- se pot utiliza doar noțiuni deja definite
- nu se poate genera un calcul infinit (trebuie să se oprească)

Recursivitate și inducție

Recursivitatea e strâns legată de inducția matematică; ambele:

- au un caz de bază
- leagă o noțiune de ea însăși (relatia de recurent / pasul inductiv)

Diferă al treilea element, sensul în care se face raționamentul:

- principiul inducției matematice: o afirmație $P(n)$ e valabilă pentru orice n (*crescând* spre infinit) dacă:

e adevărat $P(0)$ și

$P(n) \Rightarrow P(n + 1)$ (dacă $P(n)$ adevărat atunci $P(n + 1)$ adevărat)

- recurența definește ceva “mai mare” prin ceva “mai mic”
(se oprește când dimensiunea (măsura) noțiunii definite scade la zero).

Exemplu: funcția putere

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

```
#include <stdio.h>
float pwr(float 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 reprezintă o *declarație* a ei, deci poate fi folosită în orice punct ulterior (inclusiv în propriul corp – cazul apelului recursiv)
- chiar dacă scriem `putere(-2, 3)`, `-2` va fi convertit la real, întrucât se cunoaște tipul necesar pentru fiecare parametru

Mecanismul apelului recursiv

Funcția `pwr` face două calcule:

- un *test* (`n == 0 ?` a ajuns la cazul de bază ?) dacă da, `return 1`
- dacă nu, o *înmulțire*; pt. operandul drept trebuie 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.)

Remarcăm, executând pas cu pas, și urmărind figura:

- fiecare apel generează “*în cascadă*” un alt apel, până la cazul de bază
- când se ajunge la cazul de bază, toate apelurile sunt *începute și încă neterminate*
- fiecare apel își urmează propria execuție prin corpul funcției și are propriile valori pentru parametri

Exemplu: cel mai mare divizor comun

```
unsigned cmmdc(unsigned a, unsigned b) {
    return a == b ? a
           : a > b ? cmmdc(a-b, b)
           : cmmdc(a, b-a);
}
```

$cmmdc(a, b) =$

$$\begin{cases} a & a = b \\ cmmdc(a - b, b) & a > b \\ cmmdc(a, b - a) & a < b \end{cases}$$

```
int main(void) {
    printf("cmmdc(20, 8) e %u\n",
           cmmdc(20, 8));
    return 0;
}
```

- numerele unsigned se tipăresc folosind formatul %u
- calculul e corect doar cu a și b nenule. Pentru a trata și cazul zero:

```
return a == 0 ? b
       : b == 0 ? a
       : a > b ? cmmdc(a-b, b) : cmmdc(a, b-a);
```

Factorialul: două variante

```
unsigned fact1(unsigned n)
{
    return n == 0 ? 1 : n * fact1(n-1);
}
// varianta 2: apelată cu fact2(n, 1)
unsigned fact2(unsigned n, unsigned res)
{
    return n == 0 ? res : fact2(n-1, n*res);
}
```

v.1: calcul făcut la sfârșitul funcției, *după* revenirea din apelul recursiv
calcule succesive: 1*1 (1), 2*1 (2), 3*2 (6), 4*6 (24), 5*24 (120), etc.

v.2: funcția primește un rezultat parțial, care e actualizat prin calcul
și transmis mai departe (calcul *înainte* de apelul recursiv);

ajuns la cazul de bază, rezultatul e complet și e returnat până sus
apeluri (arg.2): 1, 5 (5*1), 20 (4*5), 60 (3*20), 120 (2*60), 120 (1*120)

Factorialul: secvența de apeluri

```

fact1(3)
  apel↓ ↑6
    3 * fact1(2)
      apel↓ ↑2
        2 * fact1(1)
          apel↓ ↑1
            1 * fact1(0)
              apel↓ ↑1
                1
    
```

- apelul se face în calculul rezultatului
- înmulțirea: după revenirea din apel
 calcul: $3*(2*(1*1))$

```

fact2(3, 1)
  apel↓ ↑6
    fact2(2, 3)
      apel↓ ↑6
        fact2(1, 6)
          apel↓ ↑6
            fact2(0, 6)
              apel↓ ↑6
                6
    
```

- calculul: înainte de apel
 (valoarea pt. param. doi,
 cu rol de rezultat parțial)
- la revenire: se transmite
 rezultatul înapoi până sus
 calcul: $((((1*3)*2)*1)$

Calculul sumei unei serii

Forma: $s_0 = 0$, $s_n = s_{n-1} + t_n$, pentru $n > 0$ ($t_n =$ termenul general)

Exemplu recursiv pentru seria armonică $1/1 + 1/2 + \dots + 1/n$

```
#include <stdio.h>
double suma_rec(unsigned n) {
    return n == 0 ? 0 : 1.0 / n + suma_rec(n-1);
}
int main(void) {
    printf("suma pana la 1/100: %f\n", suma_rec(100));
    return 0;
}
```

– am transcris direct definiția recursivă $s_0 = 0$, $s_n = 1/n + s_{n-1}$ ($n > 0$)

termenii se adună începând de la $1/1$ la $1/100$, la revenirea din apel

– `double`: tip pentru numere reale în dublă precizie (tipul implicit pentru constante reale, folosit uzual în calcule, și în funcțiile standard)

– tipărire cu formatul `%f` (`printf` convertește și pe `float` la `double`)

`1.0 / n` : operație între real și întreg : întregul convertit la real

Calculul unei serii – variante

Putem rescrie, ca și la `fact2`, transmițând mai departe suma parțială (calculată tot pornind invers, de la termenul de ordin cel mai mare)

```
double suma_inv(unsigned n, double rez) {  
    return n == 0 ? rez : suma_inv(n - 1, rez + 1.0/n);  
}
```

`suma_inv(n, rez)` e suma primilor n termeni (încă necalculată), plus rezultatul `rez` deja calculat al termenilor din dreapta celui curent (t_n) – dacă $n = 0$, totul e adunat deja în `rez`, care e returnat;

– altfel, rezultatul e același cu suma primilor $n - 1$ termeni (încă necalculată), plus rezultatul parțial la care se adaugă $1/n$ (termenul curent)

Pentru calcul, se apelează cu rezultatul inițial 0 : `suma_inv(100, 0.0)`

Pentru a simplifica folosirea de către utilizator, se poate defini o funcție cu un singur parametru, care o apelează pe aceasta ca funcție auxiliară:

```
double serie_armonica(unsigned n) { return suma_inv(n, 0.0); }
```

Calculul cu aproximații: rădăcina pătrată

Putem exprima recursiv calculele numerice cu aproximații succesive. Cazul de bază (oprirea) înseamnă aici atingerea unei precizii suficiente.

- pentru rădăcina pătrată \sqrt{x} avem: $a_0 = 1$, $a_{n+1} = \frac{1}{2}(a_n + \frac{x}{a_n})$
- ne oprim din calcul atunci când s-a atins precizia dorită $|a_{n+1} - a_n| < \epsilon$

```
#include <math.h>
double rad(double x, double a_n) {
    return fabs(a_n - x/a_n) < 1e-3 ? a_n
        : rad(x, (a_n + x/a_n)/2);
}
double radacina(double x) { return x < 0 : -1 ? rad(x, 1.0); }
```

$\text{rad}(x, a_n)$ înseamnă rădăcina lui x cu aproximație inițială a_n

– dacă precizia e suficient de bună, returnăm valoarea curentă

– altfel, returnăm rad calculat pentru x și noua aproximație a_{n+1}

$\text{double fabs}(\text{double } x)$: fct. valoare absolută pentru reali (din `math.h`)

pt. utilizator, definim fct. `radacina`, care întoarce codul de eroare (-1) pentru argument negativ, altfel apelează `rad` cu aproximația inițială 1

Serii cu număr necunoscut de termeni și precizie dată

Calculăm $e^x = x^0/0! + x^1/1! + x^2/2! + \dots$ cu recurența $s_n = s_{n-1} + t_n$ până valoarea absolută a termenului $t_n = x^n/n!$ e suficient de mică. Pentru a nu recalcula x^n și $n!$ exprimăm recursiv și $t_n = t_{n-1} \cdot x/n$. Transmitem la funcție n , s_{n-2} și t_{n-1} pentru a calcula s_{n-1} și t_n și a apela recursiv mai departe. Inițial: $n = 0$, $t_0 = 1$, $s_{-1} = 0$

- caz de oprire: termen $< \epsilon$ stabilit \Rightarrow neglijabil \Rightarrow ret. suma curentă
- recursiv: continuă și returnează suma calculată cu valori actualizate

```
#include <math.h>
#include <stdio.h>
double e_x(double x, unsigned n, double s_n_2, double t_n_1) {
    return fabs(t_n_1)<1e-6 ? s_n_2
        : e_x(x, n+1, s_n_2+t_n_1, t_n_1 * x/n);
}
int main(void) {
    printf("e^-1 = %f\n", e_x(-1, 1, 0.0, 1.0));
    return 0;
}
```

Recursivitatea în sintaxa limbajelor de programare

Multe elemente de limbaj pot fi oricât de complexe, dar au o structură riguros definită \Rightarrow se pretează la definiții recursive

- înșiruri liniare: un program are oricâte funcții, o funcție are oricâte argumente și instrucțiuni, etc.
- structuri mai complexe, ex. expresie formată din operator și 2 expresii

Structura (*gramatica*) limbajului se reprezintă uzual printr-o notație standard numită BNF (Backus-Naur Form). Exemplu:

antet-funcție ::= tip identificador (parametri)

parametri ::= void | lista-parametri

lista-parametri ::= tip identificador | tip identificador , lista-parametri

unde $::=$ denotă *definiție* iar $|$ *alternativă* (alegere)

Cazuri particulare: recursivitate *la stânga* și *la dreapta*, după locul în care apare noțiunea recursivă în corpul definiției