Programarea calculatoarelor

Recursivitate

Marius Minea

4 martie 2008

Să înţelegem: apelul de funcţie

```
#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 funcţii se face *prin valoare*
– se *evaluează* (calculează valoarea) toate argumentele funcţiei
– valorile se atribuie la *parametrii formali* (numele din def. fct.)
– apoi se începe execuţia funcţiei cu aceste valori

Să înţelegem: apelul de funcţie

În exemplu: programul începe cu execuţia lui `main`, deci tipărirea `printf`
– `printf` are nevoie de valoarea argumentelor sale. Prima se ştie (o *constantă şir*), a doua trebuie *calculată*: `sqr(2 * sqr(2))`
– pentru a efectua apelul *exterior* al lui `sqr` trebuie ştiut 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: funcţia începe execuţia şi îşi 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 funcţiei
 – 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 funcţie calculează numai cu *valori*, niciodată cu *expresii*

Recursivitate: Noţiuni fundamentale

Recursivitatea e un concept fundamental în matematică şi informatică.
Un obiect (noţiune) e recursiv(ă) dacă e *folosit în propria sa definiţie*.

Exemplu din matematică: şiruri 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 $n > 0$
ş.a.m.d.: combinări C_n^k, şirul lui Fibonacci, ...

Alte exemple:
– *obiecte* definite recursiv:
un şir e un singur element, sau un element urmat de un şir
ex.: cuvânt (şir de litere); număr (şir de cifre zecimale)
– *acţiuni* definite recursiv:
un drum e un pas, sau un drum urmat de încă un pas
(de exemplu o cale într-un graf)

Exemplu: funcţia 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 funcţiei reprezintă o *declaraţie* 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 cunoaşte tipul necesar pentru fiecare parametru

Mecanismul apelului recursiv

Funcţia `pwr` face două calcule:
– un *test* (`n == 0` ? a ajuns la cazul de bază ?) dacă da, `return 1`
– dacă nu, *o înmulţire*; 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, şi 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 şi încă neterminate
– fiecare apel îşi urmează propria execuţie prin corpul funcţiei
şi are propriile valori pentru parametri

Elementele unei definiţii recursive

1. cazul de bază (conditia de oprire)
– cel mai simplu caz pentru definiţia (noţiunea) respectivă. Exemple:
  termenul iniţial dintr-un şir recurent
  cel mai mic obiect (un element, în cazul şirului)
  cea mai simplă acţiune (un pas, în cazul drumului)

2. relaţia de recurenţă
– defineşte noţiunea, folosind un caz mai simplu al aceleiaşi noţiuni

3. demonstraţie (argument) de oprire a definiţiei după nr. finit de paşi
(ex. o cantitate nenegativă care descreşte urmărind definiţia)
– pentru şiruri: indicele (scade în definiţie, e nenegativ)
– pentru obiecte: dimensiunea (la fel, scade)

Sunt recursive, şi corecte, următoarele definiţii ?

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

O definiţie recursivă trebuie să fie *bine formată* (v. condiţiile 1-3)
– ceva nu se poate defini doar în funcţie de sine însuşi ($x = f(x)$)
– se pot utiliza doar noţiuni deja definite
– nu se poate genera un calcul infinit (trebuie să se oprească)

Exemplu: cel mai mare divizor comun

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

```
unsigned cmmdc(unsigned a, unsigned b) {
  return a == b ? a
                : a > b ? cmmdc(a-b, b)
                        : cmmdc(a, b-a);
}
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 şi b nenule. Pentru a trata şi 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 sfârşitul funcţiei, *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: funcţia primeşte un rezultat parţial, care e actualizat prin calcul
şi transmis mai departe (calcul *înainte* de apelul recursiv);
ajuns la cazul de bază, rezultatul e complet şi e returnat până sus
apeluri (arg.2): 1, 5 (`5*1`), 20 (`4*5`), 60 (`3*20`), 120 (`2*60`), 120 (`1*120`)

Factorialul: secvenţa 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
– înmulţirea: 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 parţial)
– la revenire: se transmite
rezultatul înapoi până sus
  calcul: (((1*3)*2)*1)

Calculul sumei unei serii

Forma: $s_0 = t_0$, $s_n = s_{n-1} + t_n$, pentru $n > 0$ (t_n = termenul general)
Exemplu recursiv pentru seria armonică $1/1 + 1/2 + \ldots + 1/n$

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

– am transcris direct definiția recursivă $s_0 = t_0$, $s_n = s_{n-1} + 1/n$ ($n > 0$)
termenii se adună începând de la 1/1 la 1/100, la revenirea din apel
– `double`: tip pentru numele reale în dublă precizie (tipul implicit pentru constante reale, folosit uzual în calcule, și în funcțiile standard)
– tipărire cu formatul `%f` (`printf` convertește si 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); }
```

`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}
`double fabs(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 $e^x = x^0/0! + x^1/1! + x^2/2! + \ldots$ 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 functie n, s_{n-2} și t_{n-1} pentru a calcula s_{n-1} și t_{n+1}
și a apela recursiv mai departe. Inițial: $n = 1$, $t_0 = 1$, $s_{-1} = 0$
– caz de oprire: termen $< \epsilon$ stabilit ⇒ neglijabil ⇒ 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;
}
```


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).

Recursivitatea în sintaxa limbajelor de programare

Multe elemente de limbaj pot fi oricât de complexe, dar au o structură riguros definită ⇒ se pretează la definiții recursive
– înșiruiri 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 identificator* (*parametri*)
parametri ::= `void` | *lista-parametri*
lista-parametri ::= *tip identificator* | *tip identificator* , *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