

Limbaje de programare

Recursivitate. Citirea de caractere

10 octombrie 2011

Cum folosim recursivitatea ?

Recursivitatea nu e doar pentru șiruri recurente.

Dacă putem reduce o problemă la un caz mai simplu (mai mic) al ei,

putem exprima acest lucru *recursiv*.

Cazul de bază e cel prea simplu ca să îl mai putem reduce.

Scriem o *funcție* care returnează răspunsul problemei.

Datele de care depinde problema devin *parametrii* funcției.

Reluăm ca exemplu: calculul sumei întregilor între a și b

Cazul de bază: nu putem separa un număr (interval vid, $a > b$).

$$\text{sum_all}(a, b) = a + \text{sum_all}(a + 1, b) \quad (\text{dacă } a \geq b)$$

```
int sum_all(int a, int b)
{
    return a > b ? 0 : a + sum_all(a+1, b);
}
```

Calculul cu aproximații: rădăcina pătrată

Din matematică: $a_0 = 1$, $a_{n+1} = \frac{1}{2}(a_n + \frac{x}{a_n})$

Șirul aproximațiilor e *recurent* \Rightarrow problema e natural recursivă
ce se *dă* (parametri): x și aproximația curentă
ce se *cere* = o aproximație suficient de bună (precizie ϵ)

Formulăm problema: calculează \sqrt{x} știind aproximația curentă a_n

Cazul de bază:

la precizie bună $|a_{n+1} - a_n| < \epsilon$ returnăm *aproximația curentă* a_n
altfel, returnăm valoarea pornind de la *noua aproximație* a_{n+1}
(apel recursiv)

Se poate demonstra: distanța față de \sqrt{x} e mai mică decât cea dintre ultimii doi termeni.

Calculul cu aproximații: rădăcina pătrată

```
#include <math.h>
// pentru declarația double fabs(double x); (val. abs. real)

double rad(double x, double an) { // rad.lui x, data aprox.an
    return fabs(an - x/an)<2e-3 ? an : rad(x, (an + x/an)/2);
}

double radacina(double x) { return x < 0 ? -1 : rad(x, 1.0); }
```

Soluția e funcția radacina: apelează rad cu aprox. inițială 1

Pentru argument negativ, returnează -1 (îl interpretăm ca eroare)

Calculul sumei unei serii

Forma: $s_0 = t_0$, $s_n = s_{n-1} + t_n$, pentru $n > 0$
(t_n = termenul general, pentru care avem o formulă)

Exemplu pentru seria armonică ($t_n = 1/n$)

$$s_n = 1/1 + 1/2 + \dots + 1/n$$

$$\text{recursiv: } s_0 = 0, \quad s_n = s_{n-1} + 1/n \text{ pentru } n > 0$$

În cuvinte: știm să răspundem direct cât e s_0 : 0.

Nu putem calcula direct s_n (pentru $n > 0$),

dar dacă aflăm cât e s_{n-1} mai trebuie să adunăm $1/n$.

⇒

Funcția care calculează pe $s(n)$ răspunde 0 dacă $n = 0$
iar altfel, calculează $s(n - 1)$, adună $1/n$ și returnează rezultatul.

Calculul sumei unei serii

```
#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;
}
```

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

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

ATENȚIE: $1/n$ dă valoarea 0 când $n > 1$ (împărțire întreagă)

Suma unei serii – variantă cu rezultat acumulat

În $s_n = s_{n-1} + 1/n$, trebuie adunat $1/n$, dar nu știm încă s_{n-1}

⇒ folosim un rezultat parțial rez la care adunăm $1/n$

⇒ apelăm recursiv cu valoarea rez + $1.0/n$

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

Când $n = 0$, totul e adunat deja în rez, care e returnat ca rezultat

În apelul inițial, rezultatul acumulat e zero: `suma_inv(100, 0.0)`

rez e o valoare auxiliară, nu face parte din enunțul problemei

Definim o funcție cu un singur parametru, care apelează `suma_inv`

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

Calculul sumei unei serii cu precizie dată

Calculăm $s_n = s_{n-1} + t_n$ ($n \geq 0$), cu $s_0 = 0$
până când valoarea absolută a termenului $t_n = x^n/n!$ e neglijabilă.

Gândim recursiv: calculul *sumei dorite*, dată fiind *suma curentă*

s_{n-1} :

- dacă termenul t_n e suficient de mic, returnăm suma curentă
- altfel: apel recursiv, calculăm de la *noua sumă* $s_{n-1} + t_n$

Exemplu: seria $1 + 1/2^2 + 1/3^2 + \dots = \pi^2/6$ $t_n = 1/n^2$ ($n > 0$):

```
double sum_2(unsigned n, double s_n_1) {  
    return 1./n/n < 1e-6 ? s_n_1 : sum_2(n+1, s_n_1 + 1./n/n);  
} // 1. == 1.0 = 1 real, forteaza impartire reala
```

și folosim apelul inițial $\text{sum_2}(1, 0)$ (pornind de la $n = 1$, $s_0 = 0$)

Recursivitate: numere ca șiruri de cifre

Putem privi (recursiv) un *număr natural în baza 10* ca șir de cifre:
are o singură cifră
sau e format din ultima cifră, precedată de *alt număr în baza 10*.

Găsim *cele două părți* folosind împărțirea la 10 cu rest:

$n = 10 \cdot (n/10) + n\%10$	$1457 = 10 \cdot 145 + 7$
ultima cifră din n e $n\%10$	$1457\%10 = 7$
numărul rămas în față e $n/10$	$1457/10 = 145$

Probleme care au soluție recursivă:
care e suma cifrelor unui număr?
dar numărul cifrelor? cea mai mare/cea mai mică cifră?

Soluția: *urmărind structura definiției recursive*:

care e *rezultatul* (răspunsul) pentru un număr de *o singură cifră*?
cum *combin* ultima cifră cu *rezultatul* (recursiv) pt. *nr. dinainte*?

Câte cifre are un număr?

1, dacă are doar o cifră. (numerele de o cifră sunt < 10)

Dacă nu, are cu o cifră mai mult decât nr. fără ultima cifră ($n/10$)

```
unsigned nrcifre(unsigned n) {  
    return n < 10 ? 1 : 1 + nrcifre(n / 10);  
}
```

Varianta cu acumulator (reținem în r câte cifre am numărat deja)

– începem să numărăm de la 1 (sigur are o cifră)

– dacă am ajuns la o singură cifră, returnăm cifrele numărate (r)

– altfel, numărăm pt. $n/10$, pornind de la o cifră mai mult

```
unsigned nrcif2(unsigned n, unsigned r) {  
    return n < 10 ? r : nrcif2(n / 10, r + 1);  
}
```

Soluția cerută trebuie să aibă un singur parametru, n :

```
unsigned nrcif(unsigned n) { return nrcif2(n, 1); }
```

Maximul cifrelor unui număr

Dacă numărul e de o cifră, cea mai mare cifră e chiar numărul
altfel e maximul dintre ultima cifră și maximul numărului rămas

```
unsigned max(unsigned a, unsigned b) { return a > b ? a : b; }  
unsigned maxcifra(unsigned n) {  
    return n < 10 ? n : max(n%10, maxcifra(n/10));  
}
```

Varianta cu rezultat acumulat: mc: maximul cifrelor văzute deja
– dacă numărul e 0, maximul e cel calculat până acum (mc)
– altfel, e maximul pentru numărul fără ultima cifră, ținând cont
de maximul curent (între cel de până acum: mc, și ultima cifră)

```
unsigned maxcif2(unsigned n, unsigned mc) {  
    return n == 0 ? mc : maxcif2(n/10, max(mc, n%10));  
}  
unsigned maxcif(unsigned n) { return maxcif2(n/10, n%10); }
```

Inversarea cifrelor unui număr

Se dă un număr, calculăm numărul cu cifrele în ordine inversă.

Construim numărul pornind de la ultima cifră. *Reținem două valori:*

– partea de număr *rămas de inversat* n (inițial tot numărul)

– partea de număr *deja inversat* r (inițial vid, valoare 0)

Exemplu: 1472 \rightarrow 147, 2 \rightarrow 14, 27 \rightarrow 1, 274 \rightarrow 2741

Funcția *recursivă* de inversare:

– dacă $n = 0$ (am terminat), rezultatul e r (partea deja inversată)

– altfel, rezultatul e *inversarea restului* (de la cifra zecilor), pornind cu rezultatul deja inversat r la care adăugăm ultima cifră din n

```
unsigned revnum_r(unsigned n, unsigned r) {  
    return n == 0 ? r : revnum_r(n / 10, 10 * r + n % 10);  
}  
unsigned revnum(unsigned n) { return revnum_r(n, 0); }
```

Caractere. Codul ASCII

ASCII = American Standard Code for Information Interchange

Caracterele sunt memorate ca și cod numeric = indice în tabel

ex. '0' == 48, 'A' == 65, 'a' == 97, etc.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

0x0	\0							\a	\b	\t	\n	\v	\f	\r		
0x10:																
0x20:		!	"	#	\$	%	&	'	()	*	+	,	-	.	/
0x30:	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0x40:	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0x50:	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
0x60:	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0x70:	p	q	r	s	t	u	v	w	x	y	z	{		}	~	

Prefixul **0x** denotă *constante hexazecimale* (în baza 16)

Caracterele < 0x20 (spațiu): *caractere de control*

Cifrele; literele mari; literele mici: 3 secvențe contigue

Codurile ASCII: $\leq 0x7f$ (127); apoi vin caractere naționale, etc.

Tipul caracter în C

Tipul standard `char` reprezintă caractere (codul ASCII – un întreg)

⇒ în C, tipul `char` e un *tip întreg*, dar cu domeniu de valori mai mic decât `int` sau `unsigned`

⇒ poate fi memorat pe *un octet* (8 *biți*)

Cf. standard: `char` poate fi `signed char`, de la -128 la 127, sau `unsigned char`, de la 0 la 255. Ambele sunt incluse în `int`.

În program, *constantele caracter* se scriu între *apostroafe* ' '

Au valori întregi (cod ASCII). În calcul: convertite automat la `int`.

Cifrele, literele mici și literele mari sunt *consecutive* ⇒ avem:

'7' == '0' + 7 '5' - '0' == 5 'E' - 'A' == 4 'f' == 'a' + 5

Reprezentări pentru caractere speciale:

'\0'	null	'\n'	linie nouă
'\a'	alarm	'\r'	carriage return
'\b'	backspace	'\f'	form feed
'\t'	tab	'\''	apostrof
'\v'	vertical tab	'\\'	backslash

Citirea unui caracter: `getchar()`

Declarația funcției, în `stdio.h` : `int getchar(void);`

Apelul funcției: `getchar()` fără parametri, dar cu `()`

Returnează codul ASCII ca `unsigned char` convertit la `int`, sau returnează valoarea EOF dacă nu s-a citit un caracter (la sfârșit de fișier, end-of-file)

E nevoie ca `getchar()` să returneze `int` și nu `char` pentru a putea exprima și constanta EOF (-1, diferită de orice `unsigned char`)

La tastatură, caracterele sunt introduse *cu ecou*, într-un *tampon*, programul le preia (ex. cu `getchar()`) doar după tastarea *Enter*.

ATENȚIE! Programul nu are control asupra datelor de intrare!

⇒ trebuie *verificate datele introduse* și tratate erorile.

Scrierea unui caracter: putchar

Declarația funcției, în `stdio.h`: `int putchar(int c);`

Apelul funcției (exemplu): `putchar('7')`

Scrie un `unsigned char` (dat ca `int`); returnează valoarea scrisă

```
#include <stdio.h>
int main(void) {
    putchar('A'); putchar(':'); // scrie A apoi :
    putchar(getchar());         // scrie caracterul citit
    return 0;
}
```


Exemplu: Citirea unui număr natural

Folosim tot definiția recursivă a numărului, evidențiind ultima cifră.

Fie numărul $c_1c_2 \dots c_m$, și secvențele parțiale $c_1, c_1c_2, c_1c_2c_3, \dots$

Avem:

$r_0 = 0, r_k = 10 \cdot r_{k-1} + c_k, (k > 0)$. Definim recursiv o funcție care calculează numărul pornind de la r_{k-1} și cifra curentă c_k :

– când caracterul citit nu mai e cifră, numărul e gata format în r

– altfel, continuă recursiv de la $10 \cdot r + c$, citind următorul caracter

Atenție, `getchar()` returnează codul ASCII, nu valoarea cifrei

\Rightarrow ajustăm cu `'0'`, de ex. `6 == '6' - '0'`

Citirea unui număr natural (cont.)

`ctype.h` conține declarațiile funcțiilor de clasificare a caracterelor: `isalpha`, `isalnum`, `isdigit`, `isspace`, etc.

Ele iau ca parametru un caracter și returnează adevărat sau fals (caracterul e de tipul respectiv, sau nu)

```
#include <ctype.h>
#include <stdio.h>
unsigned readnat_rc(unsigned r, int c) {
    return isdigit(c) ? readnat_rc(10*r + (c-'0'), getchar()) : r;
}
```

`r`: numărul deja acumulat, `c`: caracterul curent citit de la intrare

Ca soluție finală, scriem o funcție fără parametri auxiliari:

```
int readnat(void) { return readnat_rc(0, getchar()); }
```

Exemplu: Citirea unui număr întreg (cont.)

Scriem o funcție care citește un întreg, ce poate avea și semn:

```
// functie auxiliara: c: primul caracter
int readint_c(int c)
{
    return c == '-' ? - readnat() :
           c == '+' ? readnat() : readnat_rc(0, c);
}
// functia ceruta fara parametru
int readint(void) { return readint_c(getchar()); }
int main(void)
{
    printf("numarul citit este: %d\n", readint());
    return 0;
}
```

Noțiunea de efect lateral

Un *calcul* pur nu are alte efecte: următorul program nu *scrie* nimic!

```
int sqr(int x) { return x * x; }  
int main(void) { return sqr(2); }
```

Apelul repetat al unei funcții (în matematică, sau cele scrise până acum: `sqr`, `fact`, etc.) cu aceiași parametri dă același rezultat.

Tipărirea (`printf`) produce un efect vizibil (și ireversibil).

`getchar()` returnează *alt* caracter din intrare la fiecare apel; caracterul e *consumat*.

O modificare în starea mediului de execuție a programului se numește *efect lateral* (ex. citire, scriere, atribuire – v. ulterior).

Uneori e necesar să *memorăm* o valoare (caracter citit de la intrare, pentru a nu-l pierde; rezultat de funcție, pentru a nu-l recalcula).

Vom discuta cum se face aceasta prin *declararea* unei *variabile*.

Declararea variabilelor

Într-o funcție: ce se dă (parametrii); ce se cere (rezultatul).

Pentru rezultate/valori intermediare \Rightarrow *declarăm variabile*

Ex: citirea de număr: caract. curent *c* nu e în enunțul problemei
 \Rightarrow e ceva ajutător, poate fi citit în funcție. Declarăm o variabilă:

```
unsigned readnat_r(unsigned r) {  
    int c = getchar(); // declaram și inițializăm c  
    return isdigit(c) ? readnat_r(10*r + c - '0') : r;  
}
```

O *variabilă* e un obiect cu un *nume* și un *tip*. E utilă la memorarea unor valori (altele decât parametrii de funcție) necesare în calcule.

Declarația de variabile: una sau mai multe variabile de același tip:
double x; int a = 1, b, c; (a e inițializat cu 1, restul nu)

Declarăm variabile când e nevoie să *reținem rezultate* (de exemplu returnate de funcții) pentru *folosire ulterioară*.

Despre variabile

Un program C: o colecție de funcții, fiecare rezolvă o subproblemă; programul principal `main` le combină (apelează funcțiile).

Numele *parametrilor* unor funcții diferite *nu* se influențează; ca și în matematică putem avea $f(x) = \dots$ și $g(x) = \dots$
⇒ la fel pentru variabilele declarate în funcții (*variabile locale*)

Domeniul de vizibilitate al unui identificator (de ex. variabilă)
= partea de program unde poate e cunoscut (și poate fi utilizat).

Parametrii și variabilele declarate în funcții au domeniul de vizibilitate corpul funcției ⇒ *nu* sunt vizibile în exteriorul funcției.

Variabilele locale au *durată de memorare* automată:

create la fiecare apel al funcției, distruse la încheierea acestuia (între apeluri nu există și deci nu își păstrează valoarea).

Corpul `{ }` unei funcții C e o *secvență de declarații și instrucțiuni*

- în C99, declarațiile și instrucțiunile pot apărea în orice ordine
- în standardele anterioare: întâi declarații, apoi instrucțiuni