

Limbaje de programare

Recursivitate. Citirea caracterelor. Declararea variabilelor

13 octombrie 2009

## Recursivitate: putere cu înjumătățirea exponentului

---

Recursivitatea = reducere la un caz *mai simplu* al *aceleiași* probleme.

Adesea, e eficientă împărțirea în două probleme cât mai egale  
= strategie *divide et impera* (divide and conquer)

$$x^n = \begin{cases} 1 & n = 0 \\ (x^{n/2})^2 & n \text{ par} \\ x \cdot (x^{n/2})^2 & n \text{ impar} \end{cases}$$

```
double sqr(double x) { return x*x; }
double pow2(double x, unsigned n) {
    return n == 0 ? 1
           : n % 2 == 0 ? sqr(pow2(x, n/2))
           : x * sqr(pow2(x, n/2));
}
```

## Să urmărim apelurile recursive

```
#include <stdio.h>
double sqr(double x) { return x*x; }
double pow2(double x, unsigned n) {
    printf("exponent %u\n", n);
    return n == 0 ? 1 : n % 2 == 0 ? sqr(pow2(x, n/2))
                                   : x * sqr(pow2(x, n/2));
}
int main(void) {
    printf("5 la 6 = %f\n", pow2(5, 6));
    return 0;
}
```

Fiecare apel la `pow2` înjumătățește exponentul  $\Rightarrow 1 + \lceil \log_2 n \rceil$  apeluri

Ex.: `pow2(5, 6) → pow2(25, 3) → pow2(625, 1) → pow2(625, 0)`

Argumentul `pow(x, n/2)` la `sqr` *se evaluează o singură dată*

Funcțiile lucrează cu *valoarea* (numerică) a argumentelor,  
*nu substituie expresia* lor matematică în corpul funcției!

## Atenție la calcule repetate ineficient!

---

Dacă *înlocuim direct*  $x^{n/2} \cdot x^{n/2}$  în locul funcției `sqr` și tipărim exponentul pentru a urmări desfășurarea apelurilor recursive:

obținem pentru exponent $n = 3$ :	exponent 3
<code>double pow2(double x, unsigned n) {</code>	exponent 1
<code>printf("exponent %u\n", n);</code>	exponent 0
<code>return n == 0 ? 1</code>	exponent 0
<code>: n % 2 == 0 ? pow2(x, n/2) * pow2(x, n/2)</code>	exponent 1
<code>: x * pow2(x, n/2) * pow2(x, n/2);</code>	exponent 0
<code>}</code>	exponent 0

Cele două expresii `pow(x, n/2)` se evaluează *succesiv, independent!*

Fără optimizări, compilatorul nu caută expresii egale, și *recalculează*

Numărul de apeluri e mai mare decât la înmulțirea obișnuită  $x \cdot \dots \cdot x$   
 $\Rightarrow$  **ATENȚIE**, nu repetați ineficient rezolvarea aceleiași subprobleme

## Puterea cu înjumătățirea exponentului (cont.)

---

Putem rescrie definiția chiar mai simplu:

$$x^n = \begin{cases} 1 & n = 0 \\ (x^2)^{n/2} & n \text{ par} \\ x \cdot (x^2)^{n/2} & n \text{ impar} \end{cases}$$

```
double pow2(double x, unsigned n) {
    return n == 0 ? 1
           : n % 2 == 0 ? pow2(x*x, n/2)
           : x * pow2(x*x, n/2);
}
```

Am scris  $(x^2)^{n/2}$  în loc de  $(x^{n/2})^2 \Rightarrow$  nu se recalculează nici o expresie  $x * x$  se transmite *prin valoare* (după ce a fost evaluat), ca orice argument de funcție

$\Rightarrow$  din nou o soluție eficientă,  $1 + \lfloor \log_2 n \rfloor$  apeluri

$\Rightarrow$  uneori o mică reformulare a problemei duce la o soluție foarte diferită

## Exemplu: șirul lui Fibonacci (ineficient)

---

$$F_0 = F_1 = 1, F_n = F_{n-1} + F_{n-2} \quad (n \geq 2)$$

```
#include <stdio.h>
unsigned fib(unsigned n)
{
    printf("calculez fib(%u)\n",n);
    return n<2 ? 1 : fib(n-1) + fib(n-2);
}
int main(void)
{
    printf("fib(4) = %u\n", fib(4));
    return 0;
}
```

```
calculez fib(4)
calculez fib(3)
calculez fib(2)
calculez fib(1)
calculez fib(0)
calculez fib(1)
calculez fib(2)
calculez fib(1)
calculez fib(0)
fib(4) = 5
```

## Șirul lui Fibonacci (fără calcule repetate)

---

Modificăm, și scriem o funcție cu mai mulți parametri (ajutători):

- *ultimii doi termeni calculați*  $fk$  și  $fk\_1$  (așa putem face suma)
- indicele  $k$  până la care s-a făcut calculul
- și indicele  $n$  pentru care se cere  $F_n$ :

```
unsigned fibc(unsigned n, unsigned k, unsigned fk, unsigned fk_1) {  
    return k == n ? fk : fibc(n, k+1, fk+fk_1, fk);  
}
```

Ca soluție scriem funcția `fib1` cu *un singur parametru* (cum s-a cerut).

Ea apelează funcția ajutătoare `fibc` cu *valorile inițiale potrivite*:

- 1 pentru indicele  $k$  de la care pornim (știm  $F_k$  și  $F_{k-1}$ ),
- și tot 1 pentru valorile acestora ( $F_1$  și  $F_0$ ).

```
unsigned fib1(unsigned n) {  
    return n < 2 ? 1 : fibc(n, 1, 1, 1);  
}
```

## Recursivitate: numere ca șiruri de cifre

---

Definiție recursivă: un șir e un element sau un șir urmat de un element

Putem privi 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 cifra ?

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. din față* ei?



## Câte cifre are un număr?

---

1, dacă are doar o cifră. (cum testăm? 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 (ținem minte 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 ca înainte

```
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 ( $n/10$ )

```
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 aceleași cifre în ordine inversă.

Construim numărul pornind de la ultima cifră și *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 adaugăm *în spate* 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 = indicele în acest 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: ≤ 0x7f (127); apoi vin caractere naționale, etc.

## Tipul caracter în C

---

Tipul standard `char` reprezintă caractere (codul lor ASCII – un întreg) ⇒ în C, tipul `char` e un *tip întreg*, dar cu domeniu de valori mai restrâns decât `int` sau `unsigned` ⇒ poate fi memorat pe *un octet* (8 *biți*)

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

În program, *constantele caracter* se scriu între *apostroafe* (simple) `' '`. Au valori întregi: codul ASCII. În calcul se convertesc automat la `int`. Cifrele, literele mici și literele mari sunt dispuse *consecutiv* ⇒ avem:

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

Reprezentări pentru caractere speciale:	<code>'\0'</code>	null	<code>'\n'</code>	linie nouă
	<code>'\a'</code>	alarm	<code>'\r'</code>	carriage return
	<code>'\b'</code>	backspace	<code>'\f'</code>	form feed
	<code>'\t'</code>	tab	<code>'\''</code>	apostrof
	<code>'\v'</code>	vertical tab	<code>'\''</code>	backslash

## Citirea unui caracter: `getchar()`

---

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

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

Returnează caracterul (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*, și pot fi preluate de program (ex. `getchar()` doar după tastarea *Enter*).

**ATENȚIE!** Programul nu are control asupra datelor introduse la citire ⇒ 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 și `int`); returnează valoarea scrisă

```
#include <stdio.h>
```

```
int main(void) {
```

```
    putchar('A'); putchar(':');           // scrie caracterul A apoi :
```

```
    putchar(getchar());                  // scrie un caracter 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, continuăm recursiv de la  $10 \cdot r + c$ , citind următorul caracter

Ținem cont că `getchar()` returnează codul ASCII, nu valoarea cifrei

⇒ ajustăm cu `'0'`, de ex.  $6 == '6' - '0'$



## Citirea unui număr natural (cont.)

---

`ctype.h` conține declarații pentru funcții de clasificare a caracterelor: `isalpha`, `isalnum`, `isdigit`, `isspace`, etc.

Funcțiile 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.)

---

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

```
int readint_c(int c) { // tine cont de semn; c: primul caracter
    return c == '-' ? - readnat() :
           c == '+' ? readnat() : readnat_rc(0, c);
}
int readint(void) { return readint_c(getchar()); } // fara param.
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 din cele scrise până acum: *sqr*, *fact*, etc.) cu aceiași parametri produce același rezultat (repetiția poate fi ineficientă, dar rezultatul e același, ex. *pow2*, *fib*)

În contrast, tipărirea (*printf*) produce un efect vizibil (și ireversibil).

Citirea cu *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 se pierde sau rezultat de funcție, pentru a nu-l recalcula).

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

## Declararea variabilelor

---

La o problemă (funcție): ce se dă (parametrii); ce se cere (rezultatul). Uneori, e nevoie de rezultate/valori intermediare  $\Rightarrow$  *declarăm variabile*

Ex: citirea de număr: caracterul curent  $c$  nu e dat în enunțul problemei  $\Rightarrow$  e ceva ajutător, funcția îl poate citi singur. Declarăm o variabilă:

```
unsigned readnat_r(unsigned r) {  
    int c = getchar(); // declaram c, initializat cu caract. citit  
    return isdigit(c) ? readnat_r(10*r + c - '0') : r;  
}
```

O *variabilă* e un obiect cu un *nume* și un *tip*. Se folosește 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  $C$  e o colecție de funcții  $\Rightarrow$  e scris *modular*: fiecare funcție rezolvă o subproblemă; programul principal `main` le apelează/combină.

Numele *parametrilor* unor funcții diferite *nu* se influențează;

ca și în matematică putem avea  $f(x) = \dots$  și  $g(x) = \dots$

$\Rightarrow$  la fel pentru variabilele declarate în funcții (*variabile locale*)

*Domeniul de vizibilitate* al unui identificator (de ex. variabilă)

= partea de program unde poate fi utilizat (înțelesul său e cunoscut).

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

Variabilele locale au *durată de memorare* automată:

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

Corpul  $\{ \}$  unei funcții  $C$  conține o *secvență de declarații și instrucțiuni*

– în C99, declarațiile și instrucțiunile pot apărea în orice ordine

– în standardele anterioare: întâi declarații, apoi instrucțiuni

## Exemplu: citirea unui număr real

---

Adaptăm readnat: dacă întâlnim punct, citim partea fracționară

```
#include <ctype.h>
#include <stdio.h>
double readfrac(double r, double p10) { // r: fractia acumulata
    int c = getchar(); // p10: puterea subunitara a zecimalei curente
    return isdigit(c) ? readfrac(r + (c-'0')*p10, 0.1*p10) : r;
}
double readreal_r(double r) { // valoarea acumulată: reală
    int c = getchar();
    return isdigit(c) ? readreal_r(10*r + (c-'0')) : // p. întreagă
        c == '.' ? readfrac(r, 0.1) : r; // adaugă partea fracționară
}
double readreal(void) { return readreal_r(0); }
int main(void) {
    printf("%f\n", readreal());
    return 0;
}
```

## Exemplu: Inversarea unei linii de text

---

Funcție care inversează o linie de text și returnează nr. de caractere:

Inversarea recursivă unui șir: – dacă șirul e vid, inversul e vid

– altfel: punem primul element după inversul restului șirului

⇒ caracterul citit trebuie *memorat* până inversăm restul liniei

```
#include <stdio.h>
unsigned revline(void)
{
    int c = getchar();
    unsigned len = ((c == '\n') ? 0 : 1 + revline());
    putchar(c);
    return len;
}
int main(void)
{
    printf("\nLinia a avut %u caractere\n", revline());
    return 0;
}
```