

## Programarea calculatoarelor

## Recursivitate. Citirea caracterelor

Marius Minea

13 martie 2007

Programarea calculatoarelor. Curs 3

Marius Minea

## Recapitulare: folosirea funcțiilor

– pentru calcule, dar și pt. a grupa prelucrări, evitând repetarea de cod

```
int min2(int x, int y) { return x < y ? x : y; }
int min3(int a, int b, int c) {return a<b ? min2(a,c) : min2(b,c);}
```

– în `min3` am redus problema la două subprobleme mai mici  
– dacă continuam direct cu testele, înlocuind `min2` cu expresia ei:

```
int min3_direct(int a, int b, int c) {
    return a < b ? (a < c ? a : c) : (b < c ? b : c);
}
```

– trebuie repetat tiparul de cod `x < y ? x : y`, efect nedorit (cod mai mare; eventuale modificări trebuie repetate în multe locuri)  
– funcția arată mai complex, poate fi mai dificil de înțeles

Programarea calculatoarelor. Curs 3

Marius Minea

Programarea calculatoarelor. Recursivitate. Citirea caracterelor

3

## Recursivitate: putere cu înjumătățirea exponentului

– recursiv, rezolvăm o problemă reducând-o la o problemă mai simplă  
– adesea, e eficientă împărțirea în două probleme cât mai egale  
= strategie *divide et impera* (divide and conquer)

```
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));
}
```

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

– numărul de apeluri necesar e  $1 + \lceil \log_2 n \rceil$

(exponentul se înjumătățește la fiecare apel recursiv)

de ex.: `pow2(5.5, 6) → pow2(5.5, 3) → pow2(5.5, 1) → pow2(5.5, 0)`  
– evaluarea lui `pow(x, n/2)` se face o *singură dată* ca argument pt. `sqr` care lucrează cu *valoarea* obținută (nu substituie expresia de două ori)

Programarea calculatoarelor. Curs 3

Marius Minea

Programarea calculatoarelor. Recursivitate. Citirea caracterelor

4

## Apeluri și calcule repetate

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

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

– cele două expresii `pow(x, n/2)` se evaluează succesiv, independent! fără optimizări, compilatorul nu caută expresii egale, și recalculează  
– nr. de apeluri e mai mare decât la înmulțirea obișnuită  $x \cdot \dots \cdot x$

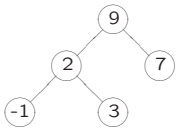
Programarea calculatoarelor. Curs 3

Marius Minea

Programarea calculatoarelor. Recursivitate. Citirea caracterelor

5

## Recursivitate: arbori binari



Arborii binari: au *noduri* și *muchii*  
un nod are doi *fii*, sau zero (e o *frunză*)  
în fiecare nod: o valoare întreagă

Definiție recursivă: un arbore binar e:

- fie o frunză
- fie un nod (rădăcină) cu subarbore stâng și subarbore drept

Am definit recursiv un *tip de date* (mulțimea tuturor arborilor binari)

Se dau: – un *nume* pentru acest tip: `arb_t`

- *declarații* ale unor funcții (operații) cu acest tip
  - implementări ale acestor funcții (un fișier C deja compilat)
- învățăm ulterior *cum să implementăm* în C tipul și funcțiile

Scriem: un program care creează/prelucrează un arbore

Necunoscând implementarea, putem folosi pt. `arb_t` doar funcțiile date

⇒ Tipul definit (`arb_t`) este un *tip de date abstract*

Programarea calculatoarelor. Curs 3

Marius Minea

Programarea calculatoarelor. Recursivitate. Citirea caracterelor

6

Fișierul antet pentru tipul de date abstract `arb_t`

Pentru a *folosi* tipul, se *dă* fișierul `arb.h` cu conținutul:

```
typedef struct arb_s *arb_t; // declara tipul arb_t
// invatam ulterior despre aceasta declaratie; acum se da

arb_t c_arb(int val, arb_t l, arb_t r); // creeaza arbore
// cu valoarea data in radacina, si doi subarbori
arb_t c_term(int val); // creeaza o frunza cu valoarea data
arb_t a_l(arb_t a); // returneaza subarbore stang
arb_t a_r(arb_t a); // returneaza subarbore drept
int a_val(arb_t a); // returneaza valoarea din radacina
int a_term(arb_t a); // adevarat daca arbore e frunza
// in C, valoare != 0 inseamna adevarat, valoare == 0 inseamna fals
void a_print(arb_t a); // tipareste arborele
```

Programarea calculatoarelor. Curs 3

Marius Minea

## Recursivitate. arbore binar

Programul trebuie compilat împreună cu codul pentru funcțiile folosite, dat într-un fișier C arb.c sau fișier obiect (compilat) arb.o

```
#include cu ghilimele caută uzual și în directorul curent
#include "arb.h"
#include <stdio.h>
int a_cnt(arb_t a) { // numara nodurile din arbore
    return a_term(a) ? 1 : 1 + a_cnt(a_l(a)) + a_cnt(a_r(a));
}
int a_sum(arbore a) { // suma numerelor din arbore
    return a_term(a) ? a_val(a)
        : a_val(a) + a_sum(a_l(a)) + a_sum(a_r(a));
}
int main(void) {
    printf("%d\n", a_sum(c_arb(5,
        c_arb(3, c_frunza(2), c_frunza(7)),
        c_frunza(-4))));

    return 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	{		}	~	

Am scris cu prefixul 0x constante hexazecimale (în baza 16)

- caracterele < 0x20 (spațiu): caractere de control
- cifrele; literele mari; literele mici: în secvențe contigue
- caracterele cu cod > 0x7f (127): nu fac parte din setul ASCII (diacritice, etc. – diverse variante standardizate de ISO)

## 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 calcule 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

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

## Citirea și scrierea unui caracter

int getchar(void) declarată în stdio.h  
– funcție fără parametri, returnează valoarea caracterului (codul ASCII)  
ca și unsigned char convertit la int  
– sau returnează valoarea specială EOF (end-of-file) (-1, diferită de orice  
unsigned char dacă nu s-a putut citi un caracter (la sfârșit de fișier)  
La tastatură, caracterele sunt introduse cu *ecou*, și devin disponibile  
pentru citire din program doar după ce se tastează *Enter*.

ATENȚIE! Programul nu are control asupra datelor introduse la citire  
⇒ trebuie verificat întotdeauna ce s-a citit / testate erorile

int putchar(int c)  
– scrie un unsigned char dat ca și int; returnează valoarea scrisă

```
#include <stdio.h>
int main(void) {
    putchar(':'); // scrie caracterul :
    putchar(getchar()); // citește și scrie un caracter
    return 0;
}
```

## Exemplu: Citirea unui număr întreg

```
#include <ctype.h> // pt. functia isdigit (caract. e cifra ?)
#include <stdio.h>
// citește nr. natural, până la primul caracter diferit de cifră
unsigned read_nat(unsigned rez, int c) // c: următorul caracter
{
    // rez: rezultatul parțial acumulat;
    return isdigit(c) ? read_nat(10*rez + (c-'0'), getchar()) : rez;
}
int readint1(int c) { // tine cont de semn; c: primul caracter
    return c == '-' ? - read_nat(0, getchar()) :
        c == '+' ? read_nat(0, getchar()) : read_nat(0, c);
}
int readint(void) { return readint1(getchar()); } // fara parametri
int main(void) {
    printf("numarul citit este: %d\n", readint());
    return 0;
}
```

## Noțiunea de efect lateral

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

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

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

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 recalcula).

Vom discuta cum se face aceasta prin *atribuire* la o *variabilă*.