Limbaje de programare

Recursivitate. Citirea de caractere

1 octombrie 2012

Recapitulare

Rezolvăm o problemă (de calcul) scriind o *funcție*.

Răspunsul la problemă (ce se cere) = *rezultatul funcției*
rezultatul e întors cu instrucțiunea `return` *expresie* ;

Datele de intrare (ce se dă) = *parametrii funcției*
(de care depinde calculul valorii)

Recapitulare: ce face o funcție

Calculează o valoare

```
double discrim(double a, double b, double c)
{
  return b*b - 4*a*c;
}
```

Produce un efect (ex. afișează un mesaj)

```
void eroare(int cod) // tipul void: nu returnează nimic
{
  printf("Eroare cu codul %d\n", cod);
}
```

Efect + valoare (scrie + calculează): mai multe instructiuni

```
int sqr(int x)
{
  printf("Calculam patratul lui %d\n", x);
  return x * x;
}
```

Recapitulare: tiparul unui program simplu

```
#include <stdio.h> // daca citim sau scriem ceva
#include <math.h> // daca folosim functii matematice

// definitie de functie: latura opusa unghiului
double latura3(unsigned a, unsigned b, double alfa)
{
  // expresia contine apeluri la alte 2 functii: cos, sqrt
  return sqrt(a*a + b*b - 2*a*b*cos(alfa));
}

int main(void)
{
  // apel de functie cu valori pt. param.; scrie rezultatul
  printf("Latura a 3-a: %f\n", latura3(3, 5, atan(1)));
  return 0;
}
```

Recursivitate: putere cu înjumătățirea exponentului

Recursivitatea = reducere la *aceeași* problemă, dar caz *mai simplu*

Cazul de bază e așa de simplu încât nu mai trebuie redus
(poate fi calculat direct)

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

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

Să urmărim apelurile recursive

```
#include <stdio.h>

double pow2(double x, unsigned n)
{
  printf("baza %f la %u\n", x, n);
  return n < 2 ? n < 1 ? 1 : x
               : n / 2 == 0 ? pow2(x*x, n/2)
                            : x * pow2(x*x, n/2);
}
int main(void)
{
  printf("5 la 6 = %f\n", pow2(5, 6));
  return 0;
}
```

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

`pow2(5, 6)` $\rightarrow$ `pow2(25, 3)` $\rightarrow$ `pow2(625, 1)` $\rightarrow$ `pow2(625, 0)`

Elementele unei definiții recursive

1. *Cazul de bază* (*NU* necesită apel recursiv)
= cel mai simplu caz pentru definiția (noțiunea) dată, definit direct
termenul inițial dintr-un șir recurent: x_0
un element, în definiția: șir = element sau șir + element

E o *EROARE* dacă lipsește cazul de bază (apel recursiv infinit!)

2. *Relația de recurență* propriu-zisă
– definește noțiunea, folosind un caz mai simplu al aceleiași noțiuni

3. Demonstrație de *oprire a recursivității* după număr finit de pași
(ex. o mărime nenegativă care descrește când aplicăm definiția)
– la șiruri recurente: indicele (≥ 0 dar mai mic în corpul definiției)
– la obiecte: dimensiunea (definim obiectul prin alt obiect mai mic)

Sunt recursive, și corecte, următoarele definiții ?

? $x_{n+1} = 2 \cdot x_n$
? $x_n = x_{n+1} - 3$
? $a^n = a \cdot a \cdot \ldots \cdot a$ (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
- se pot utiliza doar noțiuni deja definite
- nu se poate genera un calcul infinit (trebuie să se oprească)

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

Modul de calcul:

la precizie bună $|a_{n+1} - a_n| < \epsilon$ returnăm *aproximația curentă* a_n
(cazul de bază)

altfel, returnăm valoarea pornind de la *noua aproximație* a_{n+1}
(apel recursiv)

Nu avem nevoie de indicele n, și cazul de bază NU e $n = 0$
(dar e tot momentul în care nu mai e nimic de calculat)

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

Calculul cu aproximații: rădăcina pătrată

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

// radacina lui x cu eroare < 1e-6 data fiind aproximatia a_n
double rad(double x, double a_n)
{
  return fabs(a_n - x/a_n) < 2e-6 ? a_n
                                  : rad(x, (a_n + x/a_n)/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)

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:

$n = 10 \cdot (n/10) + n\%10$ $\quad$ $1457 = 10 \cdot 145 + 7$

- ultima cifră din n e $n\%10$ $\quad$ $1457\%10 = 7$
- numărul rămas în față e $n/10$ $\quad$ $1457/10 = 145$

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. 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); }
```

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!
$\Rightarrow$ 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 $c_1 c_2 \ldots c_m$, și secvențele parțiale c_1, $c_1 c_2$, $c_1 c_2 c_3$, ...

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 partea deja citită 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, islower, isupper`,
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).
Citirea cu `getchar()` returnează la fiecare apel *alt* caracter din intrare; caracterul e *consumat*.

O modificare în starea mediului de execuție a programului se numește *efect lateral* (ex. citire, scriere; atribuire (va urma)).

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 ⇒ *declarăm variabile*

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

```
unsigned readnat_r(unsigned r) {
  int c = getchar();  // declaram şi iniţializăm 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 $f(x) = \ldots$ și $g(x) = \ldots$
$\Rightarrow$ 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 $\Rightarrow$ *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

Instrucțiunea condițională (`if`)

Operatorul condițional `? :` selectează din două *expresii* de evaluat

Instrucțiunea condițională selectează între *instrucțiuni* de executat

Sintaxa:

`if` `(` *expresie* `)`
  instrucțiune1
`else`
  instrucțiune2

sau

`if` `(` *expresie* `)`
  instrucțiune1

Efectul:

Dacă expresia e *adevărată* se execută *instrucțiune1*,
altfel se execută *instrucțiune2* (sau nimic, dacă nu există)

Fiecare ramură are *o singură* instrucțiune. Dacă sunt mai multe instrucțiuni, trebuie grupate într-o *instrucțiune compusă* `{ }`

Parantezele `( )` din jurul condiției sunt obligatorii.