Programarea calculatoarelor

ecursivitate. Citirea caracterelor. Declararea variabilelor

10 martie 2009

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)

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

```
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));
}
```

– numărul de apeluri necesar e $1 + \lfloor \log_2 n \rfloor$
(exponentul se înjumătățește la fiecare apel recursiv)
de ex.: `pow2(5, 6)` → `pow2(25, 3)` → `pow2(625, 1)` → `pow2(625, 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)

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:

obținem pentru exponent $n = 3$:

```
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);
}
```

```
exponent 3
exponent 1
exponent 0
exponent 0
exponent 1
exponent 0
exponent 0
```

– 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 $x \cdot \ldots \cdot x$
⇒ *ATENȚIE* la repetarea ineficientă a rezolvării acelorași subprobleme

Puterea cu înjumătățirea exponentului (cont.)

– dar, 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);
}
```

(similar cu prima variantă, dar nu mai e nevoie de `sqr`)
se transmite *valoarea* calculată a lui `x*x`, NU se substituie expresia!
⇒ uneori o mică reformulare a problemei duce la o soluție foarte diferită

Exemplu: șirul lui Fibonacci

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

```
unsigned fib(unsigned n) {
  printf("calculez fib(%d)\n",n);
  return n<2 ? 1 : fib(n-1)+fib(n-2);
}
// pentru apelul fib(4) obținem:
```

```
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)
```

Modificăm: transmitem 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);
}             // Soluția cerută: o functie cu un singur parametru
unsigned fib1(unsigned n) {
  return n < 2 ? 1 : fibc(n, 1, 1, 1);
}
```


Recursivitate: probleme cu secvențe (șiruri)

Folosesc definiția: un șir e un element sau un șir urmat de un element

Exemplu: număr natural în baza 10: – fie are o singură cifră,
– fie e format din ultima cifră, precedată de alt număr în baza 10.

Descompunem cu formula: $n = 10 \cdot (n/10) + n\%10$ 1457 = 10 · 145 + 7
Cu această descompunere: soluții recursive la probleme cu numere:

```
unsigned nrcifre(unsigned n) {   // numarul de cifre dintr-un numar
  return n < 10 ? 1 : 1 + nrcifre(n / 10);
}                // sau, in varianta cu rezultat partial acumulat:
unsigned nrcif2(unsigned n, unsigned r) {
  return n < 10 ? r : nrcif2(n / 10, r + 1);
}    // r : numarul de cifre numarate deja
```

Soluția cerută trebuie să aibă un singur parametru, n:

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

Probleme cu numere ca siruri de cifre (cont.)

Cu aceeaşi descompunere, calculăm similar maximul cifrelor unui număr

Dacă numărul e de o cifră, cea mai mare cifră e chiar numărul
altfel, e maximul dintre ultima cifră, şi maximul cifrelor restului

```
unsigned max(unsigned a, unsigned b) { return a > b ? a : b; }
unsigned maxcifra(unsigned n) {      // cifra maxima din numar
  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
– altfel, e maximul cifrelor de la zeci în sus, ţinând cont de maximul deja găsit între cel precedent, `mc` şi 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); }
```


Probleme cu numere ca siruri de cifre (cont.)

Se dă un număr, calculăm numărul cu aceleaşi cifre în ordine inversă.
Formăm numărul de la ultima cifră şi reţinem două valori:
– fragmentul rămas de inversat `n` (iniţial tot numărul)
– fragmentul deja inversat `r` (iniţial vid, valoare 0)
Exemplu: $1472 \to 147,\ 2 \to 14,\ 27 \to 1,\ 274 \to 2741$

Funcţia recursivă de inversare: – dacă $n = 0$, gata, rezultatul e `r`
– altfel, rezultatul e inversarea restului (de la cifra zecilor), pornind cu rezultatul parţial `r` la care s-a adăugat *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 şi 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` (spaţiu): caractere de control
– cifrele; literele mari; literele mici: în secvenţe contigue
ASCII: $\leq$ `0x7f` (127); apoi: standarde ISO (caractere naţionale, 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 *biţi*)

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 şi literele mari sunt dispuse *consecutiv* ⇒ 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 şi scrierea unui caracter

`int getchar(void);` (*declaraţie* în `stdio.h`) *apel*: `getchar()`
fără parametri, returnează caracterul (codul ASCII) ca `unsigned char` convertit la `int`, sau returnează valoarea `EOF` (-1, nu e `unsigned char`) dacă nu s-a citit un caracter (la sfârşit de fişier, end-of-file)
La tastatură, caracterele sunt introduse *cu ecou*, într-un *tampon*, şi pot fi preluate de program (ex. `getchar()` doar după tastarea *Enter*.

ATENŢIE! Programul nu are control asupra datelor introduse la citire
⇒ trebuie *verificate datele introduse* şi tratate erorile

`int putchar(int c);` (*declaraţie* în `stdio.h`) ex. *apel*: `putchar('7')`
– *scrie* un `unsigned char` (dat ca şi `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 definiţia recursivă a numărului, evidenţiind ultima cifră.
Fie numărul $c_1c_2\ldots c_m$, şi secvenţele parţiale c_1, c_1c_2, $c_1c_2c_3$, ...
Avem: $r_0 = 0$, $r_k = 10 \cdot r_{k-1} + c_k$, $(k > 0)$. Definim recursiv o funcţie care calculează numărul pornind de la r_{k-1} şi 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
Ţinem cont că `getchar()` returnează codul ASCII, nu valoarea cifrei
⇒ ajustăm cu `-'0'`, de ex. `6 == '6' - '0'`

```
#include <ctype.h>      // pt. functia isdigit (caract. e cifra ?)
#include <stdio.h>
// citeşte nr.nat. cât timp vede cifre. r = val.părţii deja citite
unsigned readnat_rc(unsigned r, int c) {  // c = urm.caracter citit
  return isdigit(c) ? readnat_rc(10*r + (c-'0'), getchar()) : r;
}
```

Exemplu: Citirea unui număr întreg (cont.)

Ca soluție finală, scriem o funcție fără parametri auxiliari:
`int readnat(void) { return readnat_rc(0, getchar()); }`

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 ⇒*declarăm variabile*

Ex: citirea de număr: caracterul curent `c` nu e dat în enunțul problemei ⇒ 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 ⇒ 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 $f(x) = \ldots$ și $g(x) = \ldots$
⇒ 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 ⇒ *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 == '.' ? r + readfrac(0, 0.1) : r;  // adaugă p. fracţionară
}
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;
}
```