

6 Instrucțiuni iterative

6.1 Iterația implementată recursiv

Am văzut că prin *recursivitate* putem defini structuri *repetitive*: șiruri, liste, secvențe, și putem implementa prelucrări asupra lor: calcul de șiruri recurente, sume de serii, aproximări cu o precizie dată, etc.

E important să putem exprima repetiția (iterația) și în mod direct, pentru eficiență, dar și pentru scriere mai simplă. Ca exemplu simplu, considerăm numărarea (cu tipărire) de la o valoare inițială la o valoare finală dată.

Ilustrând zicala că un drum oricât de lung începe cu un singur pas, scriem:

$$\text{numără de la } s \text{ la } f: \begin{cases} \text{dacă } s \leq f & \text{scrie } s; \text{ numără de la } s + 1 \text{ la } f \\ \text{altfel} & \text{stop} \end{cases}$$

Schema de mai sus se traduce direct într-o funcție recursivă. Desigur că în locul tipăririi s-ar putea face orice altă prelucrare a secvenței generate.

```
void count(int start, int final)
{
    if (start <= final) {
        printf("%d\n", start);
        count(start + 1, final);
    }
}
```

Discutăm această funcție simplă. Pentru a genera fiecare element al secvenței e necesar un apel recursiv, cu transmiterea parametrilor. Aceasta afectează timpul de calcul dar mai ales necesarul de memorie. Apelul de funcție consumă timp și spațiu pentru salvarea pe stivă mai multor regiștri, care sunt restaurați la revenire; în plus se ocupă spațiu pentru parametri și eventualele variabile locale.

În particular, deși valoarea lui `final` nu se modifică, ea e transmisă ca parametru la fiecare apel recursiv, ceea ce pe lângă repetiția în scriere e o problemă de eficiență. Ca răspuns, unele limbaje oferă construcții sintactice (*closures*) care execută o funcție într-un mediu unde poate accesa anumite valori legate anterior la variabile, fără a necesita transmiterea lor ca parametri. În C, o soluție mai puțin elegantă ar fi definirea lui `final` ca variabilă globală.

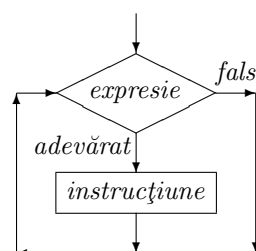
Funcția `count` e *recursivă la dreapta*: în corpul funcției, apelul recursiv se execută ultimul, efectul fiind de înlănțuire succesivă a instrucțiunilor din fiecare instanță apelată (test și tipărire). Toate acestea sunt argumente pentru definirea unei construcții de limbaj care să exprime direct iterația fără regia suplimentară datorată apelului recursiv.

6.2 Instrucțiunea while

Limbajul C definește trei instrucțiuni iterative (repetitive, de ciclare). Dintre ele, fundamentală este instrucțiunea `while`, iar celelalte două se pot defini pornind de la aceasta. Sintaxa ei este:

```
while ( expresie )
    instrucțiune
```

După cum spune numele, corpul ciclului, *instrucțiune*, e executat *atât timp cât* condiția *expresie* e adevărată. Întâi se evaluează expresia. Dacă aceasta este falsă (nulă), ciclul `while` se încheie fără execuția corpului *instrucțiune*, și se trece la instrucțiunea următoare. Dacă expresia e adevărată (nenulă), se execută corpul ciclului, după care *se revine* la evaluarea expresiei, pentru a decide dacă ciclul se continuă sau nu. Schema logică pentru ciclul `while` e:



Structural, sunt valabile aceleași precizări făcute pentru instrucțiunea `if`: expresia poate avea orice tip scalar (întreg, real sau adresă); parantezele () sunt obligatorii în sintaxa instrucțiunii; corpul e o *singură* instrucțiune, care poate fi însă compusă; acoladele { } sau terminatorul ; nu fac parte din sintaxa `while`, ci eventual din instrucțiunea-corp.

Ca exemplu, rescriem secvența de numărare folosind un ciclu. Prelucrările iterative necesită de regulă una sau mai multe *variabile* reprezentând datele prelucrate la fiecare iterație – aici, `n` pentru numărul curent. Variabila `n` e inițializată cu valoarea lui `start` și actualizată la fiecare iterație printr-o atribuire (treccerea la numărul următor). Ciclul continuă cât timp `n <= final`, condiție *reevaluată* la fiecare iterație, folosind valoarea actualizată a lui `n`.

```
void count(int start, int final)
{
    int n = start;
    while (n <= final) {
        printf("%d\n", n);
        n = n + 1;          // sau n++ sau ++n
    }
}
```

În general, decizia de a continua ciclul, prin evaluarea condiției, ține cont de ultima prelucrare făcută. Pentru aceasta, expresia folosită ca și condiție de continuare trebuie să depindă de o variabilă atribuită în cadrul ciclului. Altfel, valoarea condiției rămâne constantă, și presupunând că a fost inițial adevărată, permițând intrarea în ciclu, acesta se va executa la infinit!

Comparând, funcția recursivă e mai concisă: nu are variabile suplimentare, ci folosește ca număr curent parametrul `start`, care în fiecare instanță apelată are valoarea cu 1 mai mare decât cea din locul apelului. Putem renunța la `n` și în varianta iterativă, incrementându-l pe `start`. Independent, putem scrie codul mai concis plasând efectul lateral de incrementare în apelul de tipărire:

```
while (start <= final) {          while (start <= final)
    printf("%d\n", start);        printf("%d\n", start++);
    ++start; // sau start++
}
```

Aceste variante *modifică* valoarea lui `start`: la încheierea ciclului ea va fi `final+1` (prima valoare care nu este `<= final`). Dacă funcția nu conține alte instrucțiuni, sau cele ulterioare nu folosesc valoarea lui `start`, programul se comportă neschimbat (parametrul `start` poate fi accesat doar în funcție). Dacă însă ulterior în funcție avem nevoie de valoarea lui `start`, numărarea trebuie făcută cu o variabilă, ca în varianta inițială.

Exemplu: Descompunerea în factori primi a unui număr natural Doarece e firesc să determinăm divizorii în ordine, rezolvăm întâi subproblema de a găsi un divizor cât mai mic al numărului dat. Acest proces e iterativ: începem cu un potențial divizor (de exemplu, 2) și încercăm să vedem dacă numărul e divizibil; în caz contrar, încercăm ca divizor numărul următor. Procesul se va opri cel târziu când încercăm ca divizor numărul dat, deoarece evident se divide cu el însuși. Putem scrie astfel o funcție care returnează cel mai mic divizor al lui `n` care e cel puțin egal cu numărul de început `d`.

```
unsigned nextdiv(unsigned n, unsigned d)
{
    while (n % d) d = d + 1;
    return d;
}
```

În funcția de sus nu a fost nevoie să scriem explicit `n % d != 0` deoarece acesta e chiar înțelesul unei condiții: valoarea e considerată adevărată e nenulă.

Folosim repetat această funcție pentru a găsi toți divizorii lui `n`. Odată un divizor `d` găsit, îl tipărim, și repetăm procedeul pentru câtul rămas, `n/d`. De fiecare dată, pornim cu căutarea de la ultimul divizor găsit (care s-ar putea să îl dividă pe `n` de mai multe ori) – nu are sens să căutăm divizori mai mici, pentru că ei au fost găsiți deja. Același raționament ne asigură că divizorii găsiți sunt primi. Procesul se încheie când ultimul divizor găsit e egal cu `n`:

```

void printfact(unsigned n)
{
    unsigned d = 2;          // prima incercare
    while (n > (d = nextdiv(n, d)) {
        printf("%u*", d);
        n = n / d;
    }
    printf("%u\n", n); // divizorul ramas
}

```

Apelăm funcția cu numărul dorit, de exemplu `printfact(108)` care va tipări $2 \cdot 2 \cdot 2 \cdot 3 \cdot 3$. Condiția de intrare în ciclu actualizează întâi prin atribuire valoarea următorului divizor, și apoi îl compară cu n . Astfel, la intrarea în ciclu știm sigur că în afară de divizorul găsit vom mai avea încă unul, și putem anticipa tipărind semnul de înmulțire. Ultimul divizor (care va fi chiar valoarea rămasă a lui n) e tipărit după ieșirea din ciclu.

Prezentăm și două soluții care nu au efect lateral (atribuirea la d) în condiția din ciclu. În prima variantă, e nevoie să scriem atribuirea la d de două ori în corpul funcției: prima dată ca inițializare, iar a doua oară la finalul ciclului, pentru ca noua valoare să fie testată la următoarea iterație.

A doua variantă tipărește toți factorii urmați de un semn de înmulțire $*$ până când n devine 1. Apoi, semnul de înmulțire e șters scriind caracterul `\b` (*backspace*) care deplasează înapoi cursorul, după care e suprascris cu un spațiu, și cursorul e deplasat din nou înapoi. Soluția e utilă pentru tipărirea pe ecran, dar nerecomandabilă pentru programe care ar putea fi folosite prin redirectarea ieșirii (discutată ulterior) pentru scrierea într-un fișier.

```

void printfact1(unsigned n)      void printfact2(unsigned n)
{
    unsigned d = nextdiv(n, 2);  {
    while (n > d) {              unsigned d = 2;
        printf("%u*", d);        while (n > 1) {
        n = n / d;                d = nextdiv(n, d);
        d = nextdiv(n, d);        n = n / d;
    }                             printf("%u*", d);
    printf("%u\n", n);           }
}                                 printf("\b \b\n");
}

```

6.3 Transformarea recursivității în iterație

Rezolvările recursive și iterative ale unei probleme au caracteristici diferite: soluția recursivă se scrie de regulă fără a folosi atribuirea, datele curente la fiecare apel fiind reprezentate de parametrii funcției. Pentru aceasta, funcția poate avea nevoie de parametri suplimentari pentru a transmite spre instanța apelată recursiv rezultate parțiale deja calculate. În cazul scrierii iterative, în acest scop se folosesc variabile declarate în funcție.

În scrierea recursivă e necesar un test (instrucțiune `if` sau expresie condițională) pentru a decide dacă se continuă secvența de apeluri recursive sau s-a ajuns la cazul de bază. În scrierea iterativă, aceasta este condiția de continuare a ciclului. În locul unui apel recursiv, corpul ciclului actualizează variabilele cu expresiile corespunzătoare noilor valori ale parametrilor. Aceste aspecte sunt ilustrate mai jos comparând scrierea recursivă și iterativă a funcțiilor factorial și de citire a unui număr natural.

```

unsigned fact_r(unsigned n, unsigned r)  unsigned fact(unsigned n)
{
    // apel initial: fact_r(n, 1);        {
    if (n > 0)                             unsigned r = 1;
        return fact_r(n - 1, r * n);      while (n > 0) {
    else                                     r = r * n; n = n - 1;
        return r;                          }
}                                           return r;
}

```

```

unsigned readnat_r(unsigned r)    unsigned readnat(void)
{                                  {
    // apel initial: readnat_r(0);  unsigned r = 0;
    int c;                          int c;
    if (isdigit(c=getchar()))       while (isdigit(c=getchar()))
        return readnat_r(r*10+(c-'0'));    r = r*10 + (c-'0');
    ungetc(c, stdin);              ungetc(c, stdin);
    return r;                       return r;
}                                  }

```

În ultimul exemplu, apelul `isdigit(c = getchar())` combină trei acțiuni: se citește un caracter din intrare, e atribuit variabilei `c` pentru folosire ulterioară și transmis ca argument pentru a testa dacă e cifră. Cum instrucțiunea `return` de pe ramura de test adevărat în varianta recursivă încheie execuția funcției, ultimele două instrucțiuni se execută doar atunci când testul e fals, fără a necesita scrierea explicită a unei ramuri `else`.

Șirul lui Fibonacci Unul din cele mai cunoscute șiruri recurente e șirul lui Fibonacci, definit prin relațiile: $F_0 = F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$ (pentru $n \geq 2$). Transcriind direct definiția recursivă obținem:

```

unsigned fib_r(unsigned n)
{
    return n < 2 ? 1 : fib_r(n-1) + fib_r(n-2);
}

```

Execuția mentală a acestui cod (sau instrumentarea cu o tipărire) ne arată că în calculul lui `fib_r(n)` se repetă multe apeluri pentru calculul termenilor de ordin inferior. Rescriem recursivitatea mai eficient. Cum calculul fiecărui termen depinde de cei doi anteriori, calculăm termenii progresiv, transmițând ca parametri ultimii doi termeni calculați. Avem nevoie de asemenea de indici pentru termenul dorit, `n` și cel curent, `k`. Apelul recursiv actualizează cei doi parametri, reprezentând pe F_k și F_{k-1} după relația de recurență: ultimul e exprimat ca sumă, penultimul e înlocuit cu ultimul. Pentru folosire directă de utilizator definim funcția `fib1` care inițiază apelul cu parametrii potriviți.

```

unsigned fibc(unsigned n, unsigned k, unsigned fk, unsigned fk_1)
{
    return k < n ? fibc(n, k+1, fk+fk_1, fk) : fk;
}
unsigned fib(unsigned n) { return n < 2 ? 1 : fibc(n, 1, 1, 1); }

```

Rescriem funcția folosind atribuiri și iterație. Parametrii suplimentari definiți pentru funcția `fibc` devin variabile actualizate prin atribuire în fiecare iterație. Ele își păstrează același rol, de indice curent, ultim și penultim termen:

```

unsigned fib_it(unsigned n)
{
    unsigned k = 1, fk = 1, fk_1 = 1;
    while (k++ < n) {
        fk += fk_1;
        fk_1 = fk - fk_1;    // fostul fk
    }
    return fk;
}

```

Ciclul trebuie executat atât timp cât valoarea curentă `k` pentru care e calculat ultimul termen nu a atins încă indicele dorit `n`. În fiecare ciclu, `k` e incrementat, chiar în evaluarea condiției; operatorul de *post*incrementare ne asigură că în condiție se folosește vechea valoare. În ciclu, ar trebui să facem două atribuiri simultane: `fk = fk + fk_1` (corespunzând lui F_{k+1}) și `fk_1 = fk`. Limbajul nu permite

atribuiri simultane: putem folosi o variabilă temporară pentru a reține valoarea veche a lui `fk` sau, ca în codul dat, să o exprimăm în funcție de valoarea deja actualizată. Ciclul se oprește când `k` ajunge la `n`, deci putem returna `fk`. Când `n` e 0 sau 1, nu se intră în ciclu, și se returnează valoarea inițială a lui `fk`, 1, corectă în ambele cazuri.

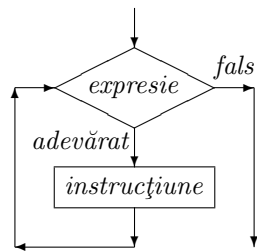
Folosirea recursivității sau a iterației ține într-o anumită măsură de filosofia și stilul individual de programare. Limbajele funcționale încurajează abordarea recursivă, fundamentală din punct de vedere al expresivității. De fapt, e ușor de văzut că instrucțiunea `while` poate fi definită ea însăși recursiv prin decizie și o nouă instrucțiune `while`:

<pre>while (expresie) instrucțiune</pre>	e echivalent cu	<pre>if (expresie) { instrucțiune while (expresie) instrucțiune }</pre>
--	-----------------	---

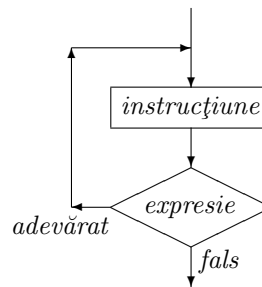
În limbajele imperative, cum e C, predomină folosirea iterației. Ea se justifică (pe lângă obișnuință) în principal prin eficiență, deși un compilator bun poate efectua automat transformări ca acele de mai sus, convertind recursivitatea la dreapta în iterație. Există însă multe probleme pentru care soluția naturală e recursivă, și rescrierea iterativă e complicată, necesitând de fapt simularea stivei de apel. Recursivitatea rămâne deci o unealtă puternică, și vom continua să o folosim acolo unde se justifică.

6.4 Instrucțiunea do - while

Ciclul cu test inițial (`while`) e gândit pentru cazul general, când e posibil ca un ciclu să se execute de oricâte ori, inclusiv deloc. Există însă situații în care ciclul trebuie executat cel puțin o dată: mai precis, se execută întâi corpul ciclului, după care se evaluează condiția care decide dacă ciclul se continuă sau se încheie. Cele două situații sunt comparate în figură:



a) ciclul cu test inițial



b) ciclul cu test final

În limbajul C, ciclul cu test final are sintaxa

```
do
    instrucțiune
while ( expresie );
```

Se execută `instrucțiune`, apoi se evaluează `expresie`. Dacă valoarea este nenulă, se reia ciclul cu execuția instrucțiunii, în caz contrar, ciclul se încheie.

Și pentru instrucțiunea `do - while`, corpul ciclului poate fi o instrucțiune compusă, parantezele ce încadrează expresia sunt obligatorii, iar expresia poate fi de orice tip scalar. Terminatorul `;` face parte din instrucțiune.

Folosim instrucțiunea `do - while` atunci când știm că ciclul trebuie executat cel puțin o dată. Reluăm ca exemplu calculul rădăcinii pătrate până când două aproximări succesive sunt mai apropiate decât precizia dorită.

Folosim două variabile, ultima aproximație și cea curentă. În fiecare ciclu, ultima aproximație e actualizată cu cea curentă, iar aceasta se recalculează cu formula dată, $(crt + x/crt)/2$. Acesta e un tipar frecvent folosit când avansăm cu doi termeni consecutivi într-un șir. Ciclul se oprește când diferența în valoare absolută între cele două aproximări nu depășește precizia cerută.

```
#include <math.h>
#define EPS 1e-6
```

```
double radacina(double x)      // trebuie x >= 0
{
    double ult, crt = 1.0;    // crt trebuie initializat
    do {
        ult = crt;
        crt = (crt + x/crt)/2;
    } while (fabs(crt - ult) > EPS);
    return crt;
}
```

Filtrarea de texte Dăm un alt exemplu legat de prelucrarea de texte. Dorim să extragem informația utilă dintr-un text scris în HTML (sau XML), unde ea este intercalată cu etichete de forma *<nume-eticheta>*. Practic, trebuie să executăm repetat două acțiuni: tipărirea textului util până la caracterul *<*, apoi ignorarea până la *>*, și reluarea procedurii.

Pentru aceasta scriem întâi două funcții generice, care ignoră, respectiv tipăresc, toate caracterele de intrare până la un caracter de oprire dat ca parametru. În ambele cazuri, trebuie să luăm precauția și să oprim prelucrarea la atingerea sfârșitului de fișier, pentru a evita blocarea la infinit. Ambele funcții returnează caracterul specificat la care s-au oprit, sau **EOF**.

```
#include <stdio.h>

int skipchars(int stop) {
    int c = getchar();
    while (c != EOF && c != stop)
        c = getchar();
    return c;
}

int writechars(int stop) {
    int c = getchar();
    while (c != EOF && c != stop) {
        putchar(c);
        c = getchar();
    }
    return c;
}
```

Programul alternează cele două funcții, începând cu tipărirea, care se va opri imediat în caz că întâlnește *<*, programul tratând astfel corect și texte care încep cu etichetă, și mai multe etichete succesive. Ambele funcții revin din apel dacă întâlnesc **EOF**, deci e suficient să testăm doar caracterul la care s-a oprit a doua prelucrare (filtrarea etichetelor), și să încheiem ciclul dacă s-a ajuns la sfârșitul intrării.

```
int main(void) {
    int c = 0;
    do {
        writechars('<');
        c = skipchars('>');
    } while (c != EOF);
    return 0;
}
```