6 Instrucţiuni iterative

6.1 Iteraţia implementată recursiv

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

E important să putem exprima repetiţia (iteraţia) şi în mod direct, pentru eficienţă, dar şi pentru scriere mai simplă. Ca exemplu simplu, considerăm numărarea (cu tipărire) de la o valoare iniţială la o valoare finală dată.

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

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

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

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

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

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

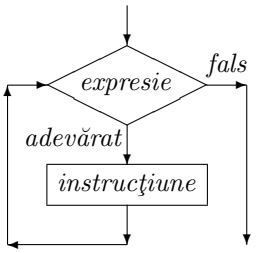
Funcţia `count` e *recursivă la dreapta*: în corpul funcţiei, apelul recursiv se execută ultimul, efectul fiind de înlănţuire succesivă a instrucţiunilor din fiecare instanţă apelată (test şi tipărire). Toate acestea sunt argumente pentru definirea unei construcţii de limbaj care să exprime direct iteraţia fără regia suplimentară datorată apelului recursiv.

6.2 Instrucţiunea `while`

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

`while (` *expresie* `)`
instrucţiune

După cum spune numele, corpul ciclului, *instrucţiune*, e executat *atât timp cât* condiţia *expresie* e adevărată. Întâi se evaluează expresia. Dacă aceasta este falsă (nulă), ciclul `while` se încheie fără execuţia corpului *instrucţiune*, şi se trece la instrucţiunea următoare. Dacă expresia e adevărată (nenulă), se execută corpului 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 aceleaşi precizări făcute pentru instrucţiunea `if`: expresia poate avea orice tip scalar (întreg, real sau adresă); parantezele `( )` sunt obligatorii în sintaxa instrucţiunii; corpul e *o singură* instrucţiune, care poate fi însă compusă; acoladele `{ }` sau terminatorul `;` nu fac parte din sintaxa `while`, ci eventual din instrucţiunea-corp.

Ca exemplu, rescriem secvenţa de numărare folosind un ciclu. Prelucrările iterative necesită de regulă una sau mai multe *variabile* reprezentând datele prelucrate la fiecare iteraţie – aici, `n` pentru numărul curent. Variabila `n` e iniţializată cu valoarea lui `start` şi actualizată la fiecare iteraţie printr-o atribuire (trecerea la numărul următor). Ciclul continuă cât timp `n <= final`, condiţie *reevaluată* la fiecare iteraţie, 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 condiţiei, ţine cont de ultima prelucrare făcută. Pentru aceasta, expresia folosită ca şi condiţie de continuare trebuie să depindă de o variabilă atribuită în cadrul ciclului. Altfel, valoarea condiţiei rămâne constantă, şi presupunând că a fost iniţial adevărată, permiţând intrarea în ciclu, acesta se va executa la infinit!

Comparând, funcţia recursivă e mai concisă: nu are variabile suplimentare, ci foloseşte ca număr curent parametrul `start`, care în fiecare instanţă apelată are valoarea cu 1 mai mare decât cea din locul apelului. Putem renunţa la `n` şi 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) {
  printf("%d\n", start);
  ++start; // sau start++
}
```

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

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

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 potenţial divizor (de exemplu, 2) şi 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 însuşi. Putem scrie astfel o funcţie care returnează cel mai mic divizor al lui n care e cel puţin egal cu numărul de început d.

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

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

Folosim repetat această funcţie pentru a găsi toţi divizorii lui n. Odată un divizor d găsit, îl tipărim, şi repetăm provcedeul 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 găsiţi deja. Acelaşi raţionament ne asigură că divizorii găsiţi 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 funcţia cu numărul dorit, de exemplu `printfact(108)` care va tipări `2*2*2*3*3` . Condiţia de intrare în ciclu actualizează întâi prin atribuire valoarea următorului divizor, şi apoi îl compară cu `n`. Astfel, la intrarea în ciclu ştim sigur că în afară de divizorul găsit vom mai avea încă unul, şi putem anticipa tipărind semnul de înmulţire. Ultimul divizor (care va fi chiar valoarea ramasă a lui `n`) e tipărit după ieşirea din ciclu.

Prezentăm şi două soluţii care nu au efect lateral (atribuirea la `d`) în conditia din ciclu. În prima variantă, e nevoie să scriem atribuirea la `d` de două ori în corpul funcţiei: prima dată ca iniţializare, iar a doua oară la finalul ciclului, pentru ca noua valoare să fie testată la următoarea iteraţie.

A doua variantă tipăreşte toţi factorii urmaţi de un semn de înmulţire `*` până când `n` devine 1. Apoi, semnul de înmulţire e şters scriind caracterul `\b` (*backspace*) care deplasează înapoi cursorul, după care e suprascris cu un spaţiu, şi cursorul e deplasat din nou înapoi. Soluţia e utilă pentru tipărirea pe ecran, dar nerecomandabilă pentru programe care ar putea fi folosite prin redirectarea ieşirii (discutată ulterior) pentru scrierea într-un fişier.

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

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

6.3 Transformarea recursivităţii în iteraţie

Rezolvările recursive şi iterative ale unei probleme au caracteristici diferite: soluţia recursivă se scrie de regulă fără a folosi atribuirea, datele curente la fiecare apel fiind reprezentate de parametrii funcţiei. Pentru aceasta, funcţia poate avea nevoie de parametri suplimentari pentru a transmite spre instanţa apelată recursiv rezultate parţiale deja calculate. În cazul scrierii iterative, în acest scop se folosesc variabile declarate în funcţie.

În scrierea recursivă e necesar un test (instrucţiune `if` sau expresie condiţională) pentru a decide dacă se continuă secvenţa de apeluri recursive sau s-a ajuns la cazul de bază. În scrierea iterativă, aceasta este condiţia 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 şi iterativă a funcţiilor factorial şi de citire a unui număr natural.

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

```
unsigned fact(unsigned n)
{
  unsigned r = 1;
  while (n > 0) {
    r = r * n; n = n - 1;
  }
  return r;
}
```

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

```
unsigned readnat(void)
{
  unsigned r = 0;
  int c;
  while (isdigit(c=getchar()))
    r = r*10 + (c-'0');
  ungetc(c, stdin);
  return r;
}
```

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

Şirul lui Fibonacci Unul din cele mai cunoscute şiruri recurente e şirul lui Fibonacci, definit ca: $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$ (pentru $n \geq 2$). Transcriind direct definiţia recursivă obţinem:

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

Execuţia 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, transmiţând ca parametri ultimii doi termeni calculaţi. Avem nevoie de asemenea de indici pentru termenului dorit, `n` şi cel curent, `k`. Apelul recursiv actualizează cei doi parametri, reprezentând pe F_k şi F_{k-1} după relaţia de recurenţă: ultimul e exprimat ca sumă, penultimul e înlocuit cu ultimul. Pentru folosire directă de utilizator definim funcţia `fib1` care iniţiază apelul cu parametrii potriviţi.

```
unsigned fibc(unsigned n, unsigned k, unsigned fk_1, unsigned fk)
{
  return k < n ? fibc(n, k+1, fk, fk+fk_1) : fk;
}
unsigned fib(unsigned n) { return n < 2 ? n : fibc(n, 1, 0, 1); }
```

Rescriem funcţia folosind atribuiri şi iteraţie. Parametrii suplimentari definiţi pentru funcţia `fibc` devin variabile actualizate prin atribuire în fiecare iteraţie. Ele îşi păstrează acelaşi rol, de indice curent, ultim şi penultim termen:

```
unsigned fib_it(unsigned n)
{
  unsigned k = 0, fk_1 = 1, fk = 0;
  while (k++ < n) {
    fk += fk_1;
    fk_1 = fk - fk_1;   // fostul fk
  }
  return fk;
}
```

Ciclul trebuie executat cât timp indicele curent `k` al ultimului termen calculat nu a atins valoarea dorită `n`. Ciclul incrementează `k` chiar în evaluarea condiţiei, cu operatorul de *post*incrementare, deci condiţia foloseşte vechea valoare a lui `k`. În ciclu, ar trebui să facem două atribuiri simultane: `fk = fk + fk_1` (pentru F_{k+1}) şi `fk_1 = fk`. Limbajul nu permite atribuiri simultane: putem folosi o variabilă temporară pentru a reţine valoarea veche a lui `fk` sau, ca în codul dat, să o exprimăm în

funcţie de valoarea deja actualizată. Ciclul se opreşte când `k` ajunge la `n`, deci putem returna `fk`. La iniţializare, am setat `fk_1 = 1`, ca să obţinem $F_1 = 1 = F_{-1} + F_0$ cu aceeaşi relaţie.

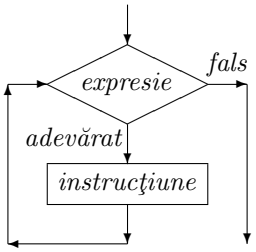
Folosirea recursivităţii sau a iteraţiei ţine într-o anumită măsură de filosofia şi stilul individual de programare. Limbajele funcţionale încurajează abordarea recursivă, fundamentală din punct de vedere al expresivităţii. De fapt, e uşor de văzut că instrucţiunea `while` poate fi definită ea însăşi recursiv prin decizie şi o nouă instrucţiune `while`:

```
while (expresie)                    if (expresie) {
  instrucţiune      e echivalent cu     instrucţiune
                                        while (expresie)
                                          instrucţiune
                                    }
```

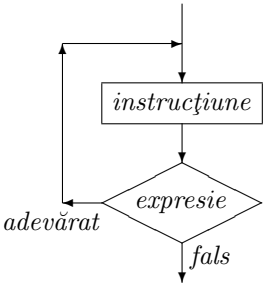
În limbajele imperative, cum e C, predomină folosirea iteraţiei. Ea se justifică (pe lângă obişnuinţă) în principal prin eficienţă, deşi un compilator bun poate efectua automat transformări ca acele de mai sus, convertind recursivitatea la dreapta în iteraţie. Există însă multe probleme pentru care soluţia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

6.4 Instrucţiunea `do - while`

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



a) ciclul cu test iniţial



b) ciclul cu test final

În limbajul C, ciclul cu test final are sintaxa

```
do
  instrucţiune
while ( expresie );
```

Se execută *instrucţiune*, apoi se evaluează *expresie*. Dacă valoarea este nenulă, se reia ciclul cu execuţia instrucţiunii, în caz contrar, ciclul se încheie.

Şi pentru instrucţiunea `do` – `while`, corpul ciclului poate fi o instrucţiune compusă, parantezele ce încadrează expresia sunt obligatorii, iar expresia poate fi de orice tip scalar. Terminatorul `;` face parte din instrucţiune.

Folosim instrucţiunea `do` – `while` atunci când ştim că ciclul trebuie executat cel puţin o dată. Reluăm ca exemplu calculul rădăcinii pătrate până când două aproximări succesive sunt mai apropiate decât precizia dorită.

Folosim două variabile, ultima aproximaţie şi cea curentă. În fiecare ciclu, ultima aproximaţie 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 şir. Ciclul se opreşte când diferenţa în valoare absolută între cele două aproximări nu depăşeşte 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 informaţia utilă dintr-un text scris în HTML (sau XML), unde ea este intercalată cu etichete de forma <*nume-eticheta*>. Practic, trebuie să executam repetat două acţiuni: tipărirea textului util până la caracterul <, apoi ignorarea până la >, şi reluarea procedurii.

Pentru aceasta scriem întâi două funcţii generice, care ignoră, respectiv tipăresc, toate caracterele de intrare până la un caracter de oprire dat ca parametru. În ambele cazuri, trebuie să luăm precauţia şi să oprim prelucrarea la atingerea sfârşitului de fişier, pentru a evita blocarea la infinit. Ambele funcţii 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 funcţii, începând cu tipărirea, care se va opri imediat în caz că întâlneşte <, programul tratând astfel corect şi texte care încep cu etichetă, şi mai multe etichete succesive. Ambele funcţii revin din apel dacă întâlnesc `EOF`, deci e suficient să testăm doar caracterul la care s-a oprit a doua prelucrare (filtrarea etichetelor), şi să încheiem ciclul dacă s-a ajuns la sfârşitul intrării.

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