Programarea calculatoarelor

funcţii; sintaxă; transmiterea argumentelor prin valoare şi prin adresă; transmiterea vectorilor şi a matricilor; recursivitate

Pe măsură ce aplicaţiile devin mai complexe, este necesar ca sarcinile de rezolvat să poată fi împărţite în subsarcini pentru ca acestea să poată fi implementate mai uşor. Pentru aceasta, limbajul C oferă posibilitatea de a implementa diverse sarcini în **funcţii** separate, fiecare funcţie realizând o anumită operaţie. O funcţie are următoarea sintaxă:

tip_returnat nume_funcţie(param$_1$, param$_2$, ..., param$_N$)
{
instrucţiuni
}

Cu următoarele componente:

- **tip_returnat** – tipul returnat de funcţie (ex: *int*, *float*). Dacă funcţia nu returnează nicio valoare, tipul returnat va fi *void* („nimic”).
- **nume_funcţie** – numele funcţiei, analog cu numele unei variabile
- **param$_1$, …, param$_N$** – parametrii funcţiei, fiecare dat ca o definire separată de variabilă (ex: *int a*, *float c*). La parametrii unei funcţii este nevoie ca fiecare parametru să aibă propriul său tip, chiar dacă e identic cu cel anterior (nu se acceptă „*int a,b*”). În C nu se pot da valori iniţiale parametrilor (nu se acceptă „*int a=3*”). Parametrii se pun între paranteze. Chiar dacă funcţia nu are niciun parametru, tot trebuie să existe parantezele de parametri, scrise ca „()”.
- **{…instrucţiuni…}** – corpul funcţiei. Constă dintr-o serie de instrucţiuni puse între acolade „{}”.

O definiţie de funcţie corespunde notaţiei matematice:

r=nume_funcţie(p_1,p_2,…,p_N), cu r ∈ tip_returnat, $p_1 \in T_1$, $p_2 \in T_2$, …,$p_N \in T_N$ unde T_1,T_2,…,T_n sunt tipurile lui p_1,p_2,…p_N

Exemplu: Se cere un număr impar n>4. Să se deseneze litera „E” în aşa fel încât pe verticală şi pe orizontală să fie câte *n* steluţe.

Varianta 1: Rezolvarea acestei probleme **fără funcţii** arată ca în programul următor:

```
#include <stdio.h>
int main()
{
   int n, i;
   printf("n=");
   scanf("%d", &n);
   for (i = 0; i < n; i++)              // @1 - linia orizontala de sus
      printf("*");
   printf("\n");
   for (i = 0; i < (n - 3) / 2; i++)    // @2 - linia verticala superioara
      printf("*\n");
   for (i = 0; i < n; i++)              // @3 - linia orizontala de la mijloc
      printf("*");
```

```
    printf("\n");
    for (i = 0; i < (n - 3) / 2; i++)      // @4 - linia verticala inferioara
        printf("*\n");
    for (i = 0; i < n; i++)                // @5 - linia orizontala de jos
        printf("*");
    printf("\n");
    return 0;
}
```

Se poate constata faptul că există două secvenţe de cod care se repetă: trasarea unei linii orizontale se repetă la liniile @1,3,5 şi trasarea unei linii verticale se repetă la liniile @2,4. Avem astfel două secvenţe de cod duplicate, una de 3 ori, alta de 2 ori.

Duplicarea de cod într-o aplicaţie este în mod ferm interzisă de normele de practică în programare.

Următoarele două motive sunt evidente:

- dacă avem de modificat ceva, atunci acea modificare trebuie realizată în toate secvenţele duplicate.
- dacă într-o asemenea secvenţă există o eroare, atunci eroarea va fi prezentă în toate duplicatele. Când se doreşte remedierea erorii, remedierea trebuie şi ea propagată în toate duplicatele.

Când programele sunt mici, aceste aspecte nu sunt foarte importante. Când dimensiunea programelor creşte şi codul este scris în mai multe fişiere, este foarte posibil să uităm să modificăm în toate locurile unde apare duplicată acea secvenţă, sau este posibil ca la unul dintre duplicate să se strecoare o greşeală la modificare. În această situaţie în aplicaţie vor apărea erori (bugs) care în general sunt greu de localizat şi de reparat.

Duplicarea de cod se rezolvă prin **factorizare**, extrăgând codul comun şi implementându-l cu ajutorul unei funcţii, funcţie care ulterior va putea fi apelată din toate punctele din program de unde este nevoie. Prin această metodă, cele două deficienţe de mai sus dispar:

- dacă avem de modificat ceva, vom modifica într-un singur loc (în corpul funcţiei)
- depanarea erorilor se face şi ea mai simplu, testând şi modificând doar codul funcţiei

Un alt avantaj major al funcţiilor este faptul că în timp, pe măsură ce implementăm diverşi algoritmi, vom avea o bibliotecă de funcţii bine puse la punct. Vom putea refolosi aceste funcţii şi în proiecte viitoare, ceea ce va reduce considerabil timpul de dezvoltare al unei aplicaţii.

Limbajul C dispune de o colecţie bogată de funcţii standard (C standard library). Acestea sunt grupate pe categorii şi sunt accesibile prin intermediul includerii unor fişiere antet (header, .h). Printre funcţiile standard se găsesc funcţii pentru fişiere, pentru caractere sau şiruri de caractere, matematice, formatare, gestiunea memoriei, etc. Este bine să se parcurgă aceste funcţii pentru a ne forma o idee despre ce ni se pune la dispoziţie şi deci ce putem folosi fără a mai fi nevoie ca noi să implementăm de la zero.

Pe lângă funcţiile C standard, mai putem găsi biblioteci de funcţii pentru cele mai variate aplicaţii: grafică, reţele de calculatoare, simulare, baze de date, etc. Multe dintre aceste biblioteci au fost dezvoltate ani de zile de către echipe de programatori şi implementează funcţionalităţi complexe, astfel încât folosirea lor ne poate scuti de foarte mult efort la scrierea aplicaţiei noastre.

Varianta 2: rezolvarea problemei anterioare **folosind funcţii**:

```
#include <stdio.h>

void linieOrizontala(int n)
{
    int i;
    for (i = 0; i < n; i++)
```

```
        printf("*");
    printf("\n");
}

void linieVerticala(int n)
{
    int i;
    for (i = 0; i < (n - 3) / 2; i++)
        printf("*\n");
}

int main()
{
    int n;
    printf("n=");
    scanf("%d", &n);
    linieOrizontala(n);             // linia orizontala de sus
    linieVerticala(n);              // linia verticala superioara
    linieOrizontala(n);             // linia orizontala de la mijloc
    linieVerticala(n);              // linia verticala inferioara
    linieOrizontala(n);             // linia orizontala de jos
    return 0;
}
```

În această variantă s-au folosit două funcţii:

- *linieOrizontala* - pentru trasarea unei linii orizontale
- *linieVerticala* - pentru trasarea unei linii verticale

Fiecare funcţie primeşte un singur parametru „*int n*” şi nu returnează nicio valoare (sunt de tip „**void**”). Se poate constata că fiecare funcţie este destul de simplă şi uşor de citit, deoarece conţine doar o secvenţă cu funcţionalitate specifică de cod. În acelaşi timp şi programul principal, *main*, este mult mai simplu şi mai lizibil, deoarece dacă funcţiile au nume bine alese, programatorul are o imagine intuitivă asupra rolului lor.

Şi în această a doua variantă apare o uşoară duplicare de cod: secvenţa *linieOrizontală/linieVerticală* se repetă de două ori. Putem elimina şi această duplicare, implementând o funcţie ”*linieOV(int lungime, int inaltime)*” ceea ce ne-ar ajuta şi la scrierea altor litere, gen L, F, C.

În C, fiecare funcţie trebuie scrisă în afara altei funcţii, deci nu am putut să scriem cele două funcţii auxiliare în interiorul lui *main*. La fel ca şi în cazul variabilelor, **o funcţie trebuie mai întâi definită şi abia apoi folosită.** Din acest motiv, dacă am fi implementat cele două funcţii după *main*, am fi avut eroare de compilare în *main*, fiindcă nu s-ar fi ştiut ce înseamnă *linieOrizontala* şi *linieVerticala*. De aceea, funcţiile au trebuit implementate **înainte** de a fi folosite.

Funcţia **main** este o funcţie obişnuită ca sintaxă, dar compilatorul o interpretează ca fiind **începutul** programului, chiar dacă înaintea ei mai sunt şi alte funcţii. Din acest motiv, întotdeauna un program C începe cu funcţia *main*, oriunde ar fi aceasta. Valoarea returnată de funcţia *main* este considerată de către sistemul de operare (Windows, Linux, etc) un indicator al felului în care s-a executat aplicaţia: cu succes (valoarea returnată 0) sau cu eroare (altă valoare).

Dacă o variabilă se declară într-o funcţie (ex: *i* din funcţia *linieOrizontala*), atunci acea variabilă este **locală** funcţiei respective şi ea nu există decât în interiorul acelei funcţii (este vizibilă doar în interiorul funcţiei). La fel şi parametrii unei funcţii sunt locali funcţiei în care au fost declarati. Mai multe funcţii pot să aibă aceleaşi nume pentru parametri şi pentru variabile locale (ex: *linieOrizontala* şi *linieVerticala* au fiecare definită câte o variabilă *i*), fără ca cele două variabile să se suprapună sau să existe vreo problemă în selectarea lor de către compilator.

Dacă o variabilă este definită în afara unei funcţii, ea este **globală**. Iată un tabel comparativ între variabilele locale şi cele globale:

Variabile locale	Variabile globale
Nu sunt iniţializate de compilator, deci iniţial au valori nedeterminate.	Sunt iniţializate chiar de la începutul programului cu valoarea 0.
Există (ocupă memorie) doar pe durata apelului funcţiei.	Există de la începutul până la sfârşitul programului,
Sunt vizibile (accesibile) doar în funcţia în care sunt declarate.	Sunt vizibile din tot programul.

Aplicaţia 9.1: Se cere un număr impar n>4. Să se deseneze cifra „8”, scris ca un pătrat cu o linie orizontală în mijloc, în aşa fel încât pe verticală şi pe orizontală să fie câte *n* steluţe. În program nu vor fi admise duplicări de cod.

O funcţie poate returna o valoare sau poate fi întreruptă din execuţie folosind instrucţiunea **return**:

```
return expresie;        //pentru funcţii care returnează o valoare
return;                 //pentru funcţii void
```

return poate să apară oriunde în corpul unei funcţii, nu doar la sfârşit. El are ca efect terminarea imediată a funcţiei respective şi revenirea în programul apelant.

Aplicaţia 9.2: Să se scrie o funcţie care returnează maximul dintre 3 valori de tip *float*, primite ca parametri. Funcţia nu va folosi nicio variabilă, cu excepţia parametrilor săi. Se va testa funcţia cu valori introduse de la tastatură.

Transmiterea argumentelor prin valoare şi prin adresă

Când se apelează o funcţie, valorile expresiilor date ca argumente se vor copia în parametrii funcţiei şi apoi funcţia va folosi aceste copii ale valorilor originare. Acest mod de apel se numeşte **transmitere prin valoare**. Din acest motiv, chiar dacă modificăm valoarea unui parametru în interiorul funcţiei, această modificare se efectuează doar pe copia valorii cu care a fost iniţializat acel parametru, fără ca valoarea originară să fie afectată.

Exemplu: Să se scrie o funcţie *swap(x,y)*, care să interschimbe valorile variabilelor date ca parametri.

Varianta 1: *swap1* **nu** implementează corect cerinţa din exemplu!!!

```
void swap1(int x,int y)
{
   int tmp=x;
   x=y;
   y=tmp;
}

int main()
{
   int a=5,b=7;
   swap1(a,b);
   printf("%d %d\n",a,b);        // 5 7
   return 0;
```

```
}
```

Într-o primă variantă încercăm să scriem o funcţie *swap1*, care are 2 parametri pe care-i interschimbă. Apelăm această funcţie cu două variabile *a* şi *b*. Ne-am aştepta ca după apelul funcţiei, valorile lui *a* şi *b* să fie interschimbate, iar pe ecran să se afişeze ”7 5”. Cu toate astea, se vor afişa tot valorile iniţiale, deci funcţia *swap1* nu le-a interschimbat.

De ce *swap1* nu a interschimbat valorile lui *a* şi *b*? Răspunsul constă chiar în transmiterea parametrilor prin valoare. În acest caz, la apelul lui *swap1(a,b)*, valorile lui *a* şi *b* au fost copiate în *x* şi *y*. Ulterior, în timpul execuţiei lui *swap1*, când *x* şi *y* au fost interschimbate, de fapt s-au interschimbat valorile din aceste două variabile, fără ca originalele *a* şi *b* să fie afectate în vreun fel.

Pentru a implementa funcţii care modifică valorile originare ale variabilelor cu care au fost apelate, putem folosi **transmiterea prin adresă**. Pentru aceasta, vom transmite funcţiei adresele variabilelor, iar funcţia va folosi pointeri pentru a opera cu valorile de la aceste adrese. În acest fel, orice modificare operată asupra valorii de la o anumită adresă, va modifica de fapt chiar variabila originară, deoarece acea variabilă se află amplasată la adresa respectivă din memorie.

Varianta 2: *swap* foloseşte transfer prin adresă şi astfel reuşeşte să interschimbe valorile variabilelor *a* şi *b*

```
void swap(int *x,int *y)        // swap are ca parametri doi pointeri, deci două adrese de memorie
{
   int tmp=*x;                  // se operează asupra valorilor pointate de cei doi pointeri
   *x=*y;                       // (valorile de la adresele stocate în x şi y)
   *y=tmp;
}

int main()
{
   int a=5,b=7;
   swap(&a,&b);                 // transmitere prin adresa: funcţia swap primeşte adresele variabilelor a şi b
   printf("%d %d\n",a,b);       // 7 5
   return 0;
}
```

Folosind transmiterea prin adresă, am reuşit să implementăm funcţia *swap*. Cu acest mod de transmitere, putem implementa orice funcţie care trebuie să modifice valorile parametrilor săi. Putem astfel înţelege de ce funcţia *scanf* are nevoie de adresele variabilelor în care va depune valorile citite (*ex: scanf(”%d”,&a);*): funcţia *scanf* foloseşte adresele variabilelor pentru a şti unde anume în memorie să depună valorile citite. Altfel, dacă am fi încercat să scriem o funcţie *scanf* fără pointeri, la apelul ei s-ar fi copiat valorile variabilelor în parametrii funcţiei, citirea s-ar fi făcut în aceste copii, iar variabilele originare ar fi rămas neschimbate.

Folosind transmiterea prin adresă, putem scrie funcţii care returnează mai multe valori, nu doar una, ca în cazul folosirii lui *return*. Pentru aceasta, vom include în funcţie câte un nou parametru pentru fiecare valoare returnată, parametru care va folosi transmitere prin adresă. La apelul funcţiei, aceşti parametri vor fi iniţializaţi cu adresele variabilelor destinaţie pentru valorile returnate. În funcţie se vor folosi parametrii respectivi pentru a seta la adresele variabilelor destinaţie valorile care trebuie returnate.

Aplicaţia 9.3: Să se scrie o funcţie care returnează două valori pe care le citeşte de la tastatură, sortate în mod crescător. Se va verifica funcţia.

Transmiterea vectorilor şi a matricilor ca argumente

Când o funcţie primeşte ca argument un vector, ea va primi de fapt adresa acelui vector. În acest caz, vectorul este interpretat ca pointer şi se transmite funcţiei doar adresa lui, fără a se copia niciun element. Deci vectorii se transmit implicit prin adresă. Din acest motiv, orice modificare efectuată în funcţie asupra valorilor unui vector, de fapt se va efectua chiar asupra vectorului originar.

Deoarece pointerii nu ştiu numărul de elemente care se află în memorie începând de la adresa pointată de ei, rezultă că la transmiterea vectorilor va trebui să transmitem funcţiei încă un parametru suplimentar, în care să specificăm numărul de elemente din vector.

Exemplu: Să se scrie o funcţie care primeşte ca parametru un vector de tip *float* şi numărul său de elemente şi returnează 1 (true) dacă toate valorile sunt pozitive sau 0 (false) dacă există şi valori negative în vector. Se va testa funcţia folosind un program care cere un n<=10 şi apoi *n* elemente de tip *float*.

```
#include <stdio.h>

int pozitive(float v[], int n)                  // alternativ: float *v
{
   int i;
   for (i = 0; i < n; i++){
      if (v[i] < 0)                             // daca a gasit un element negativ
         return 0;                              // iese imediat din functie returnand 0
   }
   return 1;                                    // daca nu a gasit niciun element negativ
}

int main()
{
   int n, i;
   float v[10];
   printf("n=");
   scanf("%d", &n);
   for (i = 0; i < n; i++){
      printf("v[%d]=", i);
      scanf("%g", &v[i]);
   }
   if (pozitive(v, n)){
      printf("toate elementele sunt pozitive");
      }else{
      printf("exista si elemente negative");
   }
   return 0;
}
```

Vectorii transmişi ca argumente se scriu fără să se specifice dimensiunea lor (ex: *float v[]*). Se iterează vectorul, şi dacă se găseşte un element negativ se revine imediat din funcţie, folosind „*return 0;*”, pentru că în acest caz nu mai are rost să se continue execuţia funcţiei. Dacă s-a terminat instrucţiunea *for*, înseamnă că nu s-a descoperit niciun element negativ şi atunci în final se returnează 1. În programul principal, valoarea returnată de funcţie se foloseşte direct în *if*, pe baza faptului că în C „0” înseamnă *false* şi orice altă valoare înseamnă *true*.

Aplicaţia 9.4: Să se scrie un program care citeşte un număr n<10 iar apoi citeşte 2 vectori *v1* şi *v2* de câte *n* elemente de tip *int*. Pentru citirea elementelor unui vector se foloseşte o funcţie *citire(v,n)*. Se va implementa o funcţie *egal(v1,v2,n),* care testează dacă toate elementele din *v1* sunt egale cu cele din *v2* la poziţii corespondente.

În acelaşi mod ca vectorii, matricile, atunci când sunt transmise ca argumente, nu se copiază în funcţie, ci este transmisă doar adresa lor. Spre deosebire de vectori, la matrici parametrul funcţiei trebuie să aibă specificat numărul de coloane de la declararea matricii (nu cel folosit efectiv, conform dimensiunilor introduse de la tastatură), deoarece acest număr intervine în formula de calcul a adresei unui element (&a[i][j] = &a[0][0] + nr_col_max*i + j), deci compilatorul are nevoie de el.

Exemplu: Se consideră o matrice care are maxim 10 linii şi 10 coloane. Să se scrie două funcţii, una care citeşte valori în matrice şi alta care afişează matricea.

```
#include <stdio.h>

// citeste elementele unei matrici a[m][n]
// numarul de coloane trebuie sa coincida cu cel de la declararea matricii
void citire(int a[][10],int m,int n)
{
   int i,j;
   for(i=0;i<m;i++){
      for(j=0;j<n;j++){
         printf("a[%d][%d]=",i,j);
         scanf("%d",&a[i][j]);
      }
   }
}

void afisare(int a[][10],int m,int n)
{
   int i,j;
   for(i=0;i<m;i++){
      for(j=0;j<n;j++){
         printf("%d\t",a[i][j]);
      }
      printf("\n");
   }
}

int main()
{
   int a[10][10];
   int m,n;
   printf("m=");scanf("%d",&m);
   printf("n=");scanf("%d",&n);
   citire(a,m,n);
   afisare(a,m,n);
   return 0;
}
```

Ţinând cont că elementele unei matrici sunt dispuse în memorie în mod consecutiv, linie după linie, putem să preluăm adresa unei linii şi să considerăm că de la această adresă începe un vector de elemente, care sunt de fapt chiar elementele din linia respectivă. Putem astfel să folosim la matrici orice funcţii care procesează vectori.

Exemplu: considerăm că avem o funcţie *vmax(v,n)*, care returnează maximul unui vector *v* de *n* elemente:

```
// returneaza maximul unui vector v de n elemente
int vmax(int *v, int n){
   int m=*v;
   for(v++ ; --n ; v++){
      if (m < *v) {
         m = *v;
      }
   }
   return m;
}
```

Această funcţie iniţializează prima oară maximul cu valoarea primului element, iar apoi foloseşte parametrul *v* ca iterator, pentru a parcurge toate elementele. Chiar dacă în funcţie valoarea lui *v* se modifică, această schimbare nu este vizibilă din exterior (ex: nu se modifică adresa unui vector care este transmis funcţiei), deoarece *v* în sine este transmis prin valoare (adresa unui vector *se copiază* în v).

Folosind *vmax*, putem afişa foarte simplu maximul unei matrici *a[m][n]*, considerând matricea ca fiind un vector cu *m*n* elemente:

```
printf("%d\n", vmax(&a[0][0],m*n));          // matricea este vazuta ca un vector de m*n elemente
```

Tot folosind *vmax*, putem determina maximul elementelor de pe linia *k*, ţinând cont că adresa de început a acestei linii este *&a[k][0]* şi va trebui să testăm *n* elemente:

```
printf("%d\n", vmax(&a[k][0],n));
```

În general, dacă folosim funcţii pe vectori, putem să operăm asupra oricăror elemente consecutive dintr-o matrice, inclusiv în interiorul unei linii (ex: să determinăm maximul ultimelor 3 elemente din linie). În schimb, cu funcţiile pe vectori, nu vom putea să operăm la nivel de coloane, deoarece elementele unei coloane nu sunt dispuse alăturat în memorie.

Deoarece operarea cu o singură linie dintr-o matrice este destul de des întâlnită în practică, limbajul C ne permite să izolăm direct adresa de început a unei linii, scriind **a[k]**, ceea ce este echivalent cu *&a[k][0]*. Constatăm că se respectă şi aici faptul că de fapt matricile sunt vectori de vectori, prin faptul că scriind doar *a[k]*, de fapt accesăm vectorul corespunzător liniei *k* din matrice. Cu această simplificare, afişarea maximului unei linii devine:

```
printf("%d\n", vmax(a[k],n));
```

Aplicaţia 9.5: Se citesc numerele m<10, 3<n<10 şi apoi elementele de tip *float* ale unei unei matrici *a[m][n]*. Ulterior, se va citi un număr 0<k<5 şi apoi elementele de tip *float* ale unui vector *v[k]*. Să se afişeze toate poziţiile unde vectorul *v* se găseşte în liniile matricii. De exemplu, dacă *v* începe în matrice la poziţia *a[i][j]*, se va afişa (i,j). Dacă vectorul nu a fost găsit în nicio linie, se va afişa un mesaj corespunzător. Pentru testarea egalităţii elementelor se va folosi funcţia *egal*, definită la aplicaţia 9.4.

Recursivitate

În anumite cazuri, soluţia unei probleme este definită în funcţie de rezultatele anterioare ale aceleiaşi probleme. De exemplu „n!” (factorialul) se poate defini ca:

1 dacă n=1
n*(n-1)! dacă n>1

Adică factorialul unui număr n>1 se defineşte în funcţie de factorialul numărului său anterior. Numim această modalitate de definire a unei probleme „**recursivă**” sau „**prin recurenţă**”. Recursivitatea este o modalitate foarte puternică de definire şi de multe ori duce la soluţii concise şi clare. Ea nu este limitată doar la matematică ci se foloseşte în multe aplicaţii, de exemplu:

- *relaţie recursivă copil-parinte*: Un om are un tată şi o mamă. Aceştia la rândul lor au fiecare un tată şi o mamă. Să se determine dacă doi oameni sunt înrudiţi genealogic.
- *definire recursivă a unei expresii în funcţie de expresii mai simple*: O expresie matematică este un număr sau o aplicare a unuia dintre operatorii adunare, scădere, înmulţire, împărţire, având ca operanzi expresii. Să se evalueze o expresie dată.
- *definirea recursivă a unor figuri geometrice de tip fractal*: Un copac începe cu o singură ramură (trunchi) din care pot creşte mai multe ramuri. La rândul lor, din acestea pot creşte alte ramuri sau frunze. Când a crescut o frunză, aceasta se consideră terminal şi din ea nu mai pot creşte alte ramuri sau frunze.

Recursivitatea este uşor de implementat cu ajutorul unei funcţii care se apelează pe ea însăşi (este recursivă).

Exemplu: Să se scrie o funcţie care primeşte ca parametru un întreg n>=0 şi returnează Fibonacci(n), definit astfel: F(0)=0, F(1)=1, F(n)=F(n-2)+F(n-1), pentru n>1.

```
#include <stdio.h>

int f(int n)
{
   if (n == 0)                    // conditii de terminare a recursivitatii
      return 0;
   if (n == 1)
      return 1;
   return f(n - 2) + f(n - 1);    // apel recursiv
}

int main()
{
   int n;
   printf("n=");
   scanf("%d", &n);
   printf("Fibonacci(%d): %d", n, f(n));
   return 0;
}
```

Când se implementează o funcţie recursivă, în primul rând trebuie să fie definită clar condiţia sau condiţiile de terminare a recursivităţii. Dacă aceste condiţii nu există sau nu acoperă toate cazurile posibile, funcţia va continua să se apeleze pe ea însăşi la infinit.

Pentru a se înţelege mai bine ce se petrece la un apel recursiv, trebuie să se ţină cont de faptul că într-un apel recursiv, funcţia apelantă (cea care a iniţiat apelul) continuă să existe şi pe lângă ea va apărea în memorie şi funcţia apelată. Vom avea astfel simultan în memorie două sau mai multe apeluri de funcţii, fiecare cu proprii săi

parametri şi variabile locale. De exemplu, putem avea simultan în memorie 1000 de apeluri ale funcţiei Fibonacci, fiecare cu propriul său parametru *n*. Funcţia apelantă va aştepta ca funcţia apelată să-şi încheie execuţia, şi apoi va continua cu valoarea returnată de aceasta. De exemplu, Fibonacci(4) va trece prin următoarea secvenţă de apeluri: f(4) = f(2)+f(3) = (f(0)+f(1))+(f(1)+f(2)) = (0+1)+(1+(f(0)+f(1))) = 1+(1+(0+1)) = 1+(1+1) = 1+2 = 3. Se poate reprezenta o asemenea secvenţă de apeluri sub forma unui arbore care pleacă de la funcţia iniţială şi fiecare ramură corespunde unui apel recursiv.

Aplicaţia 9.6: Cel mai mare divizor comun (*cmmdc*) al două numere a şi b este definit recursiv astfel:

a	dacă *a==b*
cmmdc(a-b,b)	dacă *a>b*
cmmdc(a,b-a)	dacă *b>a*

Să se scrie o funcţie recursivă pentru calculul *cmmdc* şi să se testeze aceasta pe două numere de la tastatură.

Aplicaţii propuse

Aplicaţia 9.7: Folosind funcţii standard din biblioteca matematică C (math.h), să se scrie o funcţie care primeşte ca parametri coordonatele reale a două puncte (x0,y0,x1,y1) şi returnează unghiul în grade [0º, 360º) dintre segmentul (x0,y0)-(x1,y1) şi axa OX.

***Aplicaţia* 9.8:** Se cere să se scrie funcţia lui *Ackermann* pentru două numere *m,n* date ca parametri. Să se testeze funcţia cu valori citite de la tastatură. Funcţia lui Ackermann se defineşte astfel:

n+1	dacă m=0
A(m-1,1)	dacă m>0 şi n=0
A(m-1,A(m,n-1))	dacă m>0 şi n>0

Aplicaţia 9.9: Scrieți o funcție care parcurge un vector şi returnează indicii elementelor minim şi maxim.