Capitolul IB.06. Funcții. Definire şi utilizare în limbajul C

Cuvinte cheie
Subrutină, top down, funcție apelată, funcție apelantă, parametri, argumente, definire funcții, funcții void, declarare funcții, domeniu de vizibilitate (scope), apel funcție, instrucțiunea return, transmiterea parametrilor, funcții cu argumente vectori, recursivitate

IB.06.1. Importanța funcțiilor în programare

În unele cazuri, o anumită porțiune de cod este necesară în mai multe locuri (puncte) ale programului. În loc să repetăm acel cod în mai multe locuri, este preferabil să-l reprezentăm într-o aşa numită ***subrutină*** şi să *chemăm (apelăm)* această subrutină în toate punctele programului în care este necesară execuția acelui cod. Acest lucru permite o mai bună înțelegere a programului, uşurează depanarea, testarea şi eventuala sa modificare ulterioară. În C/C++ subrutinele se numesc ***funcții***.

De ce discutam acum despre funcții? Deoarece programele prezentate în continuare vor creşte în complexitate, aşa încât, pentru dezvoltarea lor, vom aplica **tehnica de analiza şi proiectare *Top Down*** sau *Stepwise Refinement* ce cuprinde următorii paşi:

1. problema se descompune în subprobleme - în paşi de prelucrare
2. fiecare subproblemă poate fi descompusă la rândul său în alte subprobleme
3. fiecare subproblemă este implementată într-o funcție
4. aceste funcții sunt apelate în *main*, ceea ce va duce la execuția pe rând a paşilor necesari pentru rezolvarea problemei.

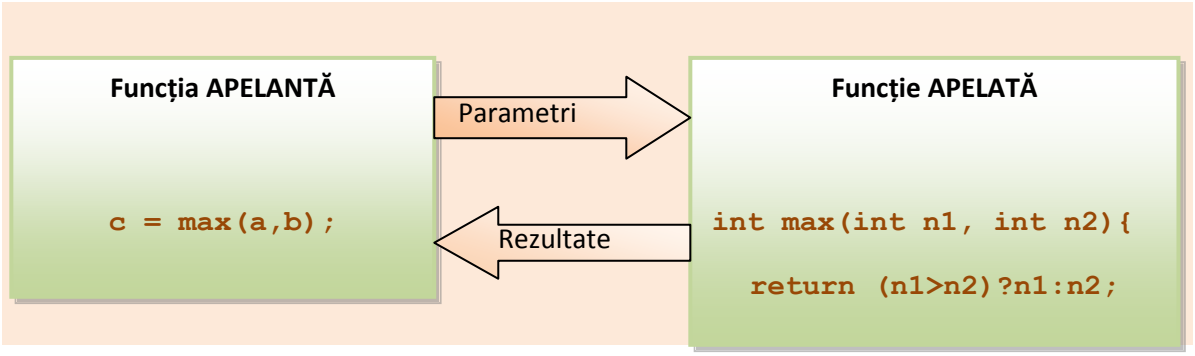
Funcția este un concept important în matematică şi programare. În limbajul C prelucrările sunt organizate ca o ierarhie de apeluri de funcții. Orice program trebuie să conțină cel puțin o funcție, funcția *main*.
Funcțiile încapsulează prelucrări bine precizate şi pot fi reutilizate în mai multe programe. Practic, nu există program care să nu apeleze atât funcții din bibliotecile existente cât şi funcții definite în cadrul aplicației respective. Ceea ce numim uzual *program* sau *aplicație* este de fapt o colecție de funcții (subprograme).

Motivele utilizării funcțiilor sunt multiple:

- Evită repetarea codului: este uşor să faci *copy* şi *paste*, dar este greu să menții şi să sincronizezi toate copiile;
- Utilizarea de funcții permite după cum spuneam dezvoltarea progresivă a unui program mare, fie de jos în sus (*bottom up*), fie de sus în jos (*top down*), fie combinat. Astfel, un program mare poate fi mai uşor de scris, de înțeles şi de modificat dacă este modular, adică format din module funcționale relativ mici;
- O funcție poate fi reutilizată în mai multe aplicații - prin adăugarea ei într-o bibliotecă de funcții - ceea ce reduce efortul de programare al unei noi aplicații;
- O funcție poate fi scrisă şi verificată separat de restul aplicației, ceea ce reduce timpul de punere la punct al unei aplicații mari (deoarece erorile pot apare numai la comunicarea între subprograme corecte);

- Întreținerea unei aplicații este simplificată, deoarece modificările se fac numai în anumite funcții şi nu afectează alte funcții (care nici nu mai trebuie recompilate);

Standardul limbajului C conține o serie de funcții care există în toate implementările limbajului. Declarațiile acestor funcții sunt grupate în *fişiere antet* cu acelaşi nume pentru toate implementările. În afara acestor funcții standard există şi alte biblioteci de funcții: funcții specifice sistemului de operare, funcții utile pentru anumite aplicații (grafică, baze de date, aplicații de rețea ş.a.).
Două entități sunt implicate în utilizarea unei funcții: un apelant, care apelează (cheamă) funcția şi **funcția apelată**. Apelantul, care este şi el o funcție, transmite **parametri** (numiti şi **argumente)** funcției apelate. Funcția primeşte aceşti parametri, efectuează operațiile din corpul funcției şi returnează rezultatul/rezultatele înapoi **funcției apelante**.



Comunicarea de date între funcții se face de regulă prin argumente şi numai în mod excepțional prin variabile externe funcțiilor – vezi domeniu de definiție.

Exemplu
Să presupunem că avem nevoie să evaluăm aria unui cerc de mai multe ori (pentru mai multe valori ale razei). Cel mai bine este să scriem o funcție numită **calculAria**(), şi să o folosim când avem nevoie.

```
#include <stdio.h>
#include <math.h>

// prototipul funcției (declararea)
double calculAria (double);

int main() {
        double raza1 = 1.1, aria1, aria2;

        // apeleaza  functia calculAria:
        aria1 = calculAria (raza1);
        printf("Aria 1 este %lf\n ", area1);

        // apeleaza  functia calculAria:
        aria2 = calculAria (2.2);
        printf("Aria 2 este %lf\n ", area2);

        // apeleaza  functia calculAria:
        printf("Aria 3 este %lf\n ",  calculAria (3.3));
        return 0;
}
// definirea functiei
double calculAria (double raza) {
        return raza* raza*M_PI; // M_PI definit in biblioteca math
```

```
}
```

Rezultatul va fi:

```
Aria 1 este 3.80134
Aria 2 este 15.2053
Aria 3 este 34.212
```

În acest exemplu este definită o funcție numită *calculAria* care primeşte un parametru de tip double de la funcția apelantă – în acest caz *main-ul* – efectuează calculul şi returnează un rezultat, tot de tip *double* funcției care o apelează. În *main*, funcția *calculAria* este chemată de trei ori, de fiecare dată cu o altă valoare a parametrului.

IB.06.2. Definirea şi utilizarea funcțiilor

Pentru a putea fi utilizată într-un program, definiția unei funcții trebuie să preceadă utilizarea (apelarea) ei.

Forma generală a unei definiții de funcție, conform standardului, este:

Sintaxa:

```
tip_rezultat_returnat nume_functie (lista_parametri_formali) {
   /* corpul functiei: */
   definirea variabilelor locale          //declarații
   prelucrari                             // instructiuni
}
```

unde:

- *tip_rezultat_returnat* este tipul rezultatului returnat de funcție. În limbajul C o parte din funcții au o valoare ca rezultat iar altele nu au(sunt de tip *void*). Pentru o funcție cu rezultat diferit de *void* tipul funcției este tipul rezultatului funcției. Tipul unei funcții C poate fi orice tip numeric, orice tip pointer, orice tip structură (*struct*) sau *void*.
  Dacă *tip_rezultat_returnat* este:
  - int - funcția este întreagă
  - void - funcția este void
  - float - funcția este reală.
- Lista parametrilor formali cuprinde declarația parametrilor formali, separați prin virgulă:

  Sintaxa:

  lista_parametri_formali = tip_p1 nume_p1, tip_p2 nume_p2, ..., tip_pn nume_pn

  unde:
  - ***tip_p1, tip_p2,., tip_pn*** **sunt tipurile parametrilor formali**
  - ***nume_p1, nume_p2, ..., nume_pn*** **sunt numele parametrilor formali**

Exemple:

1. Să se calculeze şi să se afişeze valoarea expresiei $x^m+y^n+(xy)^{m\wedge n}$, x, y, m, n fiind citiți de la tastatură, astfel încât întregii m,n să fie pozitivi. Ridicarea la putere se va realiza printr-o funcție putere care primeşte baza şi exponentul ca parametri şi returnează rezultatul.
   Modul în care se va apela (folosi) aceasta funcție îl vom arăta la **Apelul unei funcții.**

```
// functia care calculeaza baza^exp
double putere (double baza, int exp){
   int i;                        //declarare variabila locale
   float rez;              //declarare variabila locale

   //instructiuni:
   for(i=rez=1;i<=exp;i++)
        rez*=baza;

   return rez;             // returnare valoare calculata in functia apelanta
}
```

2. Numărarea şi afişarea numerelor prime mai mici ca un întreg dat n. Pentru aceasta vom scrie o funcție care testează dacă un număr este prim. Modul în care se va apela (folosi) această funcție îl vom arăta la **Apelul unei funcții.**

```
//returneaza 0 daca numarul nu este prim, 1 daca este prim

int prim (int numar){

   int div;                      //declarare variabila locale

   //instructiuni:
   for (div=2; div<=sqrt(numar); div++)

         if (numar % div ==0) return 0;// returnare 0 daca am gasit divizor
```

Parametrii formali sunt vizibili doar în corpul funcției care îi definește. Prin parametrii formali, funcția primește datele inițiale (de intrare) necesare şi poate transmite rezultate. Parametrii formali pot fi doar nume de variabile, adrese de variabile (pointeri) sau nume de vectori, deci nu pot fi expresii sau componente de vectori.

Observații:

- Definițiile funcțiilor nu pot fi incluse una în alta (ca în Pascal).
- Se recomandă ca o funcție să îndeplinească o singură sarcină şi să nu aibă mai mult de câteva zeci de linii sursă (preferabil sub 50 de linii).
- Este indicat ca numele unei funcții să fie cât mai sugestiv, poate chiar un verb (denotă o acțiune) sau o expresie ce conține mai multe cuvinte neseparate între ele. Primul cuvânt este scris cu literă mică, în timp ce restul cuvintelor sunt scrise cu prima literă mare.
  Exemple: calculAria(), setRaza(), mutaJos(), ePrim(), etc.

Funcții void

Să presupunem că avem nevoie de o funcție care să efectueze anumite acțiuni (de exmplu o tipărire), fără să fie nevoie să returneze o valoare apelantului. Putem declara această funcție ca fiind de tipul void.
Dacă funcția nu returnează nici un rezultat dar primeşte parametri, definiția va fi:

Sintaxa:
```
void nume_functie (lista_parametri_formali) {
    /* corpul functiei: */
    definirea variabilelor locale          //declarații
    prelucrari                             // instructiuni
}
```

Dacă funcția nu returnează nici un rezultat şi nu primeşte parametri, definiția va fi:

Sintaxa:
```
void nume_functie ( ) {
    /* corpul functiei: */
    definirea variabilelor locale          //declarații
    prelucrari                             // instructiuni
}
```

Observație: o funcție void poate întoarce rezultatele prelucrărilor efectuate prin parametri.

IB.06.3. Declararea unei funcții

O funcție trebuie să fie declarată înainte de utilizare. Putem realiza acest lucru în două moduri:
- amplasând în textul programului definiția funcției înaintea definiției funcției care o apelează - *main* sau alta funcție
- declarând prototipul (antetul) funcției înainte de definiția funcției care o apelează; în acest caz, definiția funcției poate fi amplasată oriunde în program.

Cu alte cuvinte, dacă funcția este definită după funcția în care este apelată, atunci este necesară o declarație anterioară a prototipului funcției.

Declarația unei funcții se face prin precizarea prototipului (antetului) funcției:

Sintaxa:
```
tip_rezultat_returnat nume_functie (lista_parametri_formali);
```

În prototip, numele parametrilor formali pot fi omişi, apărând doar tipul fiecăruia.
Exemple:
```
// prototip funcție, plasat înaintea locului în care va fi utilizată
double calculAria(double);  // fără numele parametrilor
int max(int, int);

/* Numele parametrilor din antet vor fi ignorate de compilator dar vor
servi la documentare: */
double calculAria(double raza);
int max(int numar1, int numar2);
```

Antetele funcțiilor sunt uzual grupate împreună şi plasate într-un aşa numit fişier antet (*header file).* Acest fişier antet poate fi inclus în mai multe programe.

Exemple:
O funcție max(int, int), care primeşte două numere întregi şi întoarce valoarea maximă. Funcția *max* este apelată din *main*.

```
int max(int, int); // prototip funcție (declarare)

int main() {
    printf("%d\n ", max(5, 8)); // apel al funcției max cu valori
constante
    int a = 6, b = 9, c;
    c = max(a, b);              // apel max() cu variabile
    printf("%d\n ", c);

    printf("%d\n ", max(c, 99)); // apel max()

    return 0;
}

// Definire functie
int max(int num1, int num2) {
    return (num1 > num2) ? num1 : num2;
}
```

```
// functia lines definită după main, dar declarată înainte:
void lines ( int);      // declarație funcție

int main () {
   lines (3);           // utilizare funcție
}

void lines (int n) {
   for (int i=0; i<n; i++)
   printf("\n");
}
```

Prototipul implicit al unei funcții, este:
int nume_functie(void);

Cu alte cuvinte, în lipsa unei declarații de tip explicite se consideră că tipul implicit al funcției este *int*.
Şi argumentele formale fără un tip declarat explicit sunt considerate implicit de tipul *int*, dar nu trebuie abuzat de această posibilitate.
Exemplu:

```
rest (a,b) { //echivalent cu: int rest (int a, int b)
      return a%b;
}
```

În limbajul C se pot defini şi funcții cu număr variabil de argumente, care pot fi apelate cu număr diferit de argumente efective.

Mai jos apar două variante de scriere a programelor. Prima variantă, în care funcția *main* e prezentă la începutul programului, după prototipurile funcțiilor apelate, oferă o imagine a prelucrărilor realizate de program.

Varianta 1:	Varianta 2:
prototipuri definiția funcției main definiții funcții	definiții funcții definiția funcției main

IB.06.4. Domeniu de vizibilitate (scope)

În C/C++ există următoarea convenție :

Convenție C:
Un nume (variabilă, funcție) poate fi utilizat numai după ce a fost declarat, iar domeniul de vizibilitate (scope) al unui nume este mulțimea instrucțiunilor (liniilor de cod) în care poate fi utilizat acel nume (numele este vizibil).

Prin urmare, avem ca regulă de bază: identificatorii sunt accesibili doar în blocul de instrucțiuni în care au fost declarați; ei sunt necunoscuți în afara acestor blocuri.

Într-un program putem avea două categorii de variabile:

- **locale - variabilele declarate:**
  - **în funcții (corpul unei functii este un bloc de instructiuni)**
  - **în blocuri de instructiuni**
  - **ca parametri formali.**
- **externe (globale) - variabile definite în afara funcțiilor.**

Locul unde este definită o variabilă determină domeniul de vizibilitate al variabilei respective: o variabilă definită într-un bloc poate fi folosită numai în blocul respectiv, iar variabilele cu acelaşi nume din funcții (blocuri) diferite sunt alocate la adrese diferite şi nu nici o legătură între ele.
Numele unei variabile este unic (nu pot exista mai multe variabile cu acelaşi nume), dar o variabilă locală poate avea numele uneia globale, caz în care, în interiorul funcției, e valabilă noua semnificație a numelui.
Pentru variabilele locale memoria se alocă la activarea funcției/blocului (deci la execuție) şi este eliberată la terminarea executării funcției/blocului. Inițializarea variabilelor locale se face tot la execuție şi de aceea se pot folosi expresii pentru inițializarea lor (nu numai constante).

Exemple:

```
#include<stdio.h>
#include<stdlib.h>

int fact=1;                    // declarare varriabila externa - globala

void factorial(int n)                // parametru formal
{ int i;                             // declarare variabila locala
  fact=1;
  for(i=2;i<=n;i++) fact=fact*i;
}
int main(void) {
  int v;                             // declarare variabila locala
  factorial(3);
  printf("3!=%d\n",fact);            // utilizare variabila externa
  printf("Introd o valoare:");
```

```
  // utilizare variabila locala:
  scanf("%d",&v);
  factorial(v);
  printf("%d!=%d\n",v,fact);
  return 0;
}
```

Atenție! Definiția unui identificator maschează pe cea a aceluiaşi identificator declarat într-un suprabloc sau în fişier (global)! Apariția unui identificator face referință la declararea sa în cel mai mic bloc care conține această apariție!

```
#include<stdio.h>
#include<stdlib.h>
int fact=1;                    // declarare variabila externa - globala

void factorial(int n)
{
  int i;                          // declarare variabila locala
  int fact=1;                     /* declarare variabila locala, acesta
                                  va fi folosita in functie mai departe! */
  for(i=2;i<=n;i++) fact=fact*i;
}

int main(void)
{
  int v;                          // declarare variabila locala
  factorial(3);
  printf("3!=%d\n",fact);         // utilizare variabila externa fact!!!

   printf("Introd o valoare:");
  scanf("%d",&v);
  factorial(v);
  printf("%d!=%d\n",v,fact);      // utilizare variabila externa fact!!!

return 1;
}
```

Atenție! Presupunând ca valoarea introdusă pentru v este 3, se va afişa 3!=1 deoarece valoarea variabilei globale fact nu este modificată. Rezultatul este 1 oricare ar fi valoarea lui v !!!

Observații:

- Unul dintre motivele pentru care se vor evita variabile externe este acela că, din neatenție, putem defini variabile locale cu acelaşi nume ca variabila externă, ceea ce poate conduce la erori.
- Nu se recomandă utilizarea de variabile externe decât în cazuri rare, când mai multe funcții folosesc în comun mai multe variabile şi se doreşte simplificarea utilizării funcțiilor, sau în cadrul unor biblioteci de funcții.
- O funcție poate deci utiliza variabilele globale, cele locale, precum si parametrii formali. Pentru reutilizare şi pentru a nu modifica accidental variabilele globale, este indicat ca o funcție să nu acceseze variabile globale.

IB.06.5.Apelul unei funcții

Apelul unei funcții, adică utilizarea acesteia se poate face astfel:

Sintaxa:
nume_functie (lista_parametri_actuali)
/* poate apare ca operand intr-o expresie, dacă funcția returnează un rezultat */

Observații:

- **pentru funcție fără tip (void) apelul este:**
  nume_funcție (lista_parametrii_efectivi);
- **pentru funcție cu tip T, unde t este de tipul T, apelul poate fi:**
  t = nume_funcție (lista_parametri_efectivi);
  sau poate apare într-o expresie în care se foloseşte rezultatul ei:
  if (nume_funcție (lista_parametri_efectivi) == 1) …

Parametrii folosiți la apelul funcției se numesc parametri efectivi şi pot fi orice expresii (constante, funcții etc.). Parametrii efectivi trebuie să corespundă ca număr şi ca tipuri (eventual prin conversie implicită) cu parametrii formali (cu excepția unor funcții cu număr variabil de argumente).
Este posibil ca tipul unui parametru efectiv să difere de tipul parametrului formal corespunzător, cu condiția ca tipurile să fie "compatibile" la atribuire. Conversia de tip (între numere sau pointeri) se face automat, la fel ca la atribuire:

```
x = sqrt(2);            // param. formal double, param.efectiv int
```

În cazul funcțiilor *void* fără parametri, apelul se face prin:

Sintaxa:
nume_functie ( ); **// instrucțiune expresie**

La apelul unei funcții, pe stiva de apeluri (păstrată de Sistemul de Operare) se crează o *înregistrare de activare*, care cuprinde, de jos în sus:

- adresa de revenire din funcție
- valorile parametrilor formali
- valorile variabilelor locale.

În momentul apelării unei funcții, se execută corpul său, după care se revine în funcția apelantă, la instrucțiunea următoare apelului.
Revenirea dintr-o funcție se face fie la întâlnirea instrucțiunii *return*, fie la terminarea execuției corpului funcției (funcția poate să nu conțină instrucțiunea *return* doar în cazul funcțiilor *void*).
Exemple - apelul funcțiilor *putere* şi *prim*:

1. Să se calculeze şi să se afişeze valoarea expresiei $x^m+y^n+(xy)^{m\wedge n}$, x, y, m, n fiind citiți de la tastatură, astfel încât întregii m,n să fie pozitivi. Ridicarea la putere se va realiza printr-o funcție *putere* care primeşte baza şi exponentul ca parametri şi returnează rezultatul – vezi definirea unei funcții. Modul de apel al funcției *putere*:

```
int main(void){
  int m,n;
  double x,y;
     while( printf(" m(>=0):"), scanf("%d",&m), m<0);
```

```
      while( printf(" n(>=0):"), scanf("%d",&n), n<0);
  if ( m==0 && n==0 ) {       //semnalare eroare 0^0
           puts("0^0 imposibil");
           return 0;                      // din main
  }
  printf("valorile lui x,y:");
      scanf("%lf%lf",&x,&y);
      printf("%lf^%d+%lf^%d+(%lf*%lf)^%d^%d (cu putere ):%lf\n", x, m, y, n,
    x, y, m, n,  putere(x,m)+putere(y,n) +     putere(x*y,putere(m,n)));
    // apel functie putere scrisa de utilizator
  printf("Rezultat cu pow: %lf\n", pow(x,m)+pow(y,n)+pow(x*y,pow(m,n)));
    // apel functie pow din biblioteca math

      return 0;
}
```

2. Numărarea şi afişarea numerelor prime mai mici ca un întreg dat n. Pentru aceasta vom scrie o funcție care testează dacă un număr este prim – vezi definirea unei funcții. Modul în care se va apela (folosi) aceasta funcție:

```
int main () {
   int n,m,contor ;
      /* contor de numere prime*/

   printf ("n= "); scanf ("%d",&n);
   contor=0;
   for (m=2;m<=n;m++)                    // incearca numerele m
            if ( prim(m) == 1 ){         // dacă m este prim
                  printf ("%d\n",m);
                  contor++;
            }

   printf ("există  %d numere prime mai mici decat %d\n", contor,n);
   return 0;
}
```

IB.06.6. Instrucțiunea return

În corpul funcției se poate folosi instrucțiunea return pentru a returna o valoare funcției apelante (o valoare corespunzătoare antetului funcției).
Forme ale instrucțiunii return:

Sintaxa:

- **return;** **//dacă funcția e void**
- **return expresie;**

Expresia e de acelaşi tip cu tip_rezultat al funcției(eventual prin conversie implicită).

Corpul unei funcții poate conține una sau mai multe instrucțiuni *return*. În corpul unei funcții de tip *void*, se poate folosi instrucțiunea *return* - fără o valoare returnată - pentru a reda controlul apelantului, însă în acest caz, al funcțiilor *void*, utilizarea lui *return* este opțională; dacă lipseşte, se adaugă automat *return* ca ultimă instrucțiune.
În funcția *main* instrucțiunea *return* are ca efect terminarea întregului program.

Exemplu de program cu două funcții:

```
#include <stdio.h>
void clear () {                 // şterge ecran prin defilare
int i;                          // variabila locala funcției clear
for (i=0; i<24; i++)
  putchar('\n');
}
int main(){
  clear();                     // apel funcție
}
```

Observații:

- O funcție de un tip diferit de *void* trebuie să conțină cel puțin o instrucțiune *return* prin care se transmite rezultatul funcției:

```
// factorial de n
long fact (int n) {
long nf=1L;              // initializare rezultat
while (n)
   nf=nf * n--;          // echivalent cu: nf=nf * n; n=n-1;
return nf;               // rezultat funcție
}
```

- Compilatoarele C nu verifică şi nu semnalează dacă o funcție de un tip diferit de *void* conține sau nu instrucțiuni *return*, iar eroarea se manifestă la execuție.
Cuvântul *else* după o instrucțiune *return* poate lipsi, dar de multe ori este prezent pentru a face codul mai clar.
Exemplu fără else:

```
// transforma caracterul din variabila c in literă mare

char toupper (char c) {
   if (c>='a' && c<='z')            // daca c este litera mica
   return c+'A'-'a';                // returnează cod litera mare
return c;                           // ramane neschimbat
}
```

- Instrucțiunea *return* poate fi folosită pentru ieşirea forțată dintr-un ciclu şi din funcție, cu reducerea lungimii codului sursă. Exemplu de funcție care verifică dacă un număr dat este prim:

```
int prim (int numar){
   int div;
   for(div=2; div<=sqrt(numar); div++)
      if (numar % div == 0) return 0;
   return 1;
}
```

IB.06.7. Transmiterea parametrilor

În C, transmiterea parametrilor se face prin valoare: parametrii actuali sunt transmişi prin valoare, la apelul funcției (valorile lor sunt depuse pe stiva program iar apoi copiate în parametrii formali. Modificarea valorii lor de către funcție nu este vizibilă în exterior!

Urmează un exemplu ce pune în evidență modul de transmitere a parametrilor prin valoare:

....

Observaţii:

- Dacă se doreşte ca o funcție să modifice valoarea unei variabile, trebuie să i se transmită *pointerul* la variabila respectivă - vezi Pointeri!
- Dacă parametrul este un tablou, cum numele este echivalent cu pointerul la tablou, funcția poate modifica valorile elementelor tabloului, primind adresa lui. A se observa că trebuie să se transmită ca parametri şi dimensiunea/ dimensiunile tabloului.
- Dacă parametrul este şir de caractere, dimensiunea tabloului de caractere nu trebuie să se transmită, sfârşitul şirului fiind indicat de caracterul terminator '\0'.

IB.06.8. Funcții cu argumente vectori

Pentru argumentele formale de tip vector nu trebuie specificată dimensiunea vectorului între parantezele drepte, oricum aceasta va fi ignorată.

Exemplu:

```
// calcul valoare maxima dintr-un vector:

float maxim (float a[], int n ) {
      float max=a[0];
      for (int k=1;k<n;k++)
            if ( max < a[k]) max=a[k];
      return max;
}

// exemplu de utilizare - in main:

float xmax, x[]= {3,6,2,4,1,5,3};
xmax = maxim(x,7);
```

Un argument formal vector poate fi declarat şi ca pointer, deoarece are ca valoare adresa de început a zonei unde este memorat vectorul.
Exemplu:

```
float maxim (float *a, int n ) { ... }
```

De remarcat că pentru un *parametru efectiv* vector nu mai trebuie specificat explicit că este un vector, deoarece există undeva o declarație pentru variabila respectivă, care stabileşte tipul ei. Este chiar greşit sintactic să se scrie:

```
xmax = maxim ( x[ ], 7 ) ;          // NU !
```

O funcție C nu poate avea ca rezultat direct un vector, dar poate modifica elementele unui vector primit ca parametru.

Exemplu de funcție pentru ordonarea unui vector prin metoda bulelor – ***Bubble Sort***:

```
void sort (float a[ ], int n) {

      int n,i,j, gata;
      float aux;

      do {

            gata = 1;// presupunem initial ca nu sunt necesare schimbari de elemente

            for (i=0;i<n-1;i++) // compara n-1 perechi de elemente vecine
                  if ( a[i] > a[i+1] ) {// daca nu sunt in ordine crescatoare
                        aux=a[i];
                        a[i]=a[i+1];
                        a[i+1]=aux;
                        // am interschimbat a[i] cu a[i+1]

                        gata =0;
      // seteaza ca s-a facut o schimbare
                  }
      } while (!gata); // repeta cat timp au mai fost schimbari de elemente
}
```

Probleme pot apare la parametrii efectivi de tip matrice din cauza interpretării diferite a zonei ce conține elementele matricei de către funcția apelată şi respectiv de funcția apelantă. Pentru a interpreta la fel matricea liniarizată este important ca cele două funcții să folosească acelaşi număr de coloane în formula de liniarizare. Din acest motiv nu este permisă absența numărului de coloane din declarația unui parametru formal matrice.

Exemple:

```
void printmat(int a[][10], int nl, int nc);    // corect, cu nc <= 10
void printmat(int a[][], int nl, int nc);      // greşit !
```

O altă soluție la problema parametrilor matrice este utilizarea de matrice alocate dinamic şi transmise sub forma unui pointer.
O sinteză a transmiterii parametrilor de tip tablou este dată în cele ce urmează:

Parametru formal	Parametru actual
tip_baza nume_vector[] **tip_baza nume_ vector[dim]**	**nume_vector**
tip_baza nume_ matrice[][dim2] //dim2 trebuie sa apara **tip_baza nume_ matrice[dim1][dim2]**	**nume_matrice**

IB.06.9. Funcții recursive

În continuare se va prezenta conceptul de recursivitate şi câteva exemple de algoritmi recursivi, pentru a putea înțelege mai bine această tehnică puternică de programare, ce permite scrierea unor soluții clare, concise şi rapide, care pot fi uşor înțelese şi verificate.

IB.06.9.1 Ce este recursivitatea ?

Un obiect sau un fenomen se defineşte în mod recursiv dacă în definiția sa există o referire la el însuşi.

Recursivitatea este folosită cu multa eficiență în matematică. Spre exemplu, definții matematice recursive sunt:

- **Definiția numerelor naturale:**

  0 € N
  dacă i € N, atunci succesorul lui i € N

- **Definiția funcției factorial**

  fact : N -> N
  fact (n) = 1, daca n=0
  n * fact (n-1), daca n>0

- **Definiția funcției Ackermann**

  ac: N x N -> N
  ac(m,n) = n+1, dacă m=0
  ac(m-1,1), dacă n=0
  ac(m-1, ac(m,n-1)), dacă m, n >0

- **Definiția funcției Fibonacci**

  fib : N -> N
  fib(n) = 1, dacă n=0 sau n=1
  fib(n-2) + fib(n-1), dacă n>1

Utilitatea practică a recursivității rezultă din posibilitatea de a defini un set infinit de obiecte printr-o singură relație sau printr-un set finit de relații.

Recursivitatea s-a impus în programare odată cu apariția unor limbaje de nivel înalt, ce permit scrierea de module ce se autoapelează (PASCAL, LISP, ADA, ALGOL, C sunt limbaje recursive, spre deosebire de FORTRAN, BASIC, COBOL, nerecursive).
Recursivitatea este strâns legată de iterație. ***Iterația*** este execuția repetată a unei porțiuni de program, până la îndeplinirea unei condiții (exemple: *while*, *do-while*, *for* din C).

Recursivitatea presupune execuția repetată a unui modul, însă în cursul execuției lui (şi nu la sfârşit, ca în cazul iterației), se verifică o condiție, a cărei nesatisfacere implică reluarea execuției modulului de la început, fără ca execuția curentă să se fi terminat. În momentul satisfacerii condiției se revine în ordine inversă din lanțul de apeluri, reluându-se şi încheindu-se apelurile suspendate.
Un program recursiv poate fi exprimat: P = M (Si , P) , unde M este mulțimea ce conține instrucțiunile Si şi pe P însuşi.

Structurile de program necesare şi suficiente în exprimarea recursivității sunt funcțiile:

Definiție:
O funcție recursivă este o funcție care se apelează pe ea însăşi, direct sau indirect.

Recursivitatea poate fi directă - o funcție P conține o referință la ea însăşi, sau indirectă - o funcție P conține o referință la o funcție Q ce include o referință la P. Vom pune accentul mai ales pe funcțiile direct recursive.

Se pot deosebi două feluri de funcții recursive:

- Funcții cu un singur apel recursiv, ca ultimă instrucțiune, care se pot rescrie uşor sub forma nerecursivă (iterativă).
- Funcții cu unul sau mai multe apeluri recursive, a căror formă iterativă trebuie să folosească o stivă pentru memorarea unor rezultate intermediare.

Recursivitatea este posibilă în C datorită faptului că, la fiecare apel al funcției, adresa de revenire, variabilele locale şi argumentele formale sunt puse într-o stivă (gestionată de compilator), iar la ieşirea din funcție (prin *return*) se scot din stivă toate datele puse la intrarea în funcție (se "descarcă" stiva).

Exemplu generic:

```
void p(){ //functie recursiva
    p(); //apel infinit
}

//apelul trebuie conditionat in una din variantele:
   • if(cond) p();
   • while(cond) p();
   • do p() while(cond);
```

Exemplu de funcție recursivă de tip *void*:

```
void binar (int n) {                  // se afişeaza n in binar
  if (n>0) {
    binar(n/2);                       // scrie echiv. binar al lui n/2
    printf("%d",n%2);                 // şi restul impartirii n la 2
  }
}
```

Funcția de mai sus nu scrie nimic pentru *n=0*, dar poate fi uşor completată cu o ramură *else* la instrucțiunea *if*.

Apelul recursiv al unei funcții trebuie să fie condiționat de o decizie care să împiedice apelul în cascadă (la infinit); aceasta ar duce la o eroare de program - depăşirea stivei.

- **Orice funcție recursivă trebuie să conțină cel puțin o instrucțiune *if*, plasată de obicei chiar la început!**
- **Prin această instrucțiune se verifică dacă mai este necesar un apel recursiv sau se iese din funcție.**
- **Absența instrucțiunii *if* conduce la o recursivitate infinită (la un ciclu fără condiție de terminare)!**

Pentru funcțiile de tip diferit de *void* apelul recursiv se face printr-o instrucțiune *return*, prin care fiecare apel preia rezultatul apelului anterior!

Anumite funcții recursive corespund unor relații de recurență.

Exemple:

```
// Calculul a^n:
double putere (double a, int n) {
   if (n==0)        return 1.;              // a^0 = 1
   else return a * putere(a, n-1);          // a^n=a*a^(n-1)
}

// Algoritmul lui Euclid poate folosi o relație de recurență:
int cmmdc (int a,int b) {
   if ( a%b==0) return b;
   return cmmdc( b,a%b);                    // cmmdc(a,b)=cmmdc(b,a%b)
}
```

Observații:

- Funcțiile recursive nu conțin în general cicluri explicite (cu unele excepții), iar repetarea operațiilor este obținută prin apelul recursiv. O funcție care conține un singur apel recursiv ca ultimă instrucțiune poate fi transformată într-o funcție nerecursivă, înlocuind instrucțiunea *if* cu *while*.
- Fiecare apel recursiv are parametri diferiți, sau măcar o parte din parametri se modifică de la un apel la altul.
- Se pune întrebarea: *"Ce este mai indicat de utilizat: recursivitatea sau iterația?"* Algoritmii recursivi sunt potriviți pentru a descrie probleme care utilizează formule recursive sau pentru prelucrarea structurilor de date definite recursiv (liste, arbori), fiind mai eleganți și mai simplu de înțeles și verificat. Iterația este uneori preferată din cauza vitezei mai mari de execuție și a memoriei necesare la execuție mai reduse. Spre exemplu, varianta recursivă a funcției Fibonacci duce la 15 apeluri pentru n=5, deci varianta iterativă este mult mai performantă.
- Apelul recursiv al unei funcții face ca pentru toți parametrii să se creeze copii locale apelului curent (în stivă) , acestea fiind referite și asupra lor făcându-se modificările în timpul execuției curente a funcției. Când execuția funcției se termină, copiile sunt extrase din stivă, astfel încât modificările operate asupra parametrilor nu afectează parametrii efectivi de apel, corespunzători. De asemenea, pentru toate variabilele locale se rezervă spațiu la fiecare apel recursiv.
- Pe parcursul unui apel, sunt accesibile doar variabilele locale și parametrii pentru apelul respectiv, nu și cele pentru apelurile anterioare, chiar dacă acestea poartă același nume!
- De reținut că, pentru fiecare apel recursiv al unei funcții se crează copii locale ale parametrilor valoare și ale variabilelor locale, ceea ce poate duce la risipa de memorie.

IB.06.9.2 Verificarea și simularea programelor recursive

Se face ca și în cazul celor nerecursive, printr-o demonstrație formală, sau testând toate cazurile posibile:

- Se verifică întâi dacă toate cazurile particulare (ce se execută când se îndeplineşte condiția de terminare a apelului recursiv) funcționează corect.
- Se face apoi o verificare a funcției recursive, pentru restul cazurilor, presupunând că toate componentele din codul funcției funcționează corect. Verificarea e deci inductivă. Acesta e un avantaj al programelor recursive, ce permite demonstrarea corectitudinii lor simplu şi clar.

Exemplu: Funcția recursivă de calcul a factorialului

```
//varianta recursiva
int fact(int n){
     if(n==1)
          return 1;
     return n*fact(n-1); //apel recursiv
}
//varianta nerecursiva
int fact(int n){
     int i,f;
     for(i=f=1; i<=n; i++)
           f*=i;
```

Verificarea corectitudinii în acest caz cuprinde doi paşi:

1. pentru n=1 valoarea 1 ce se atribuie factorialului este corectă
2. pentru n>1, presupunând corectă valoarea calculată pentru predecesorul lui n de către *fact(n-1)*, prin înmulțirea acesteia cu *n* se obține valoarea corectă a factorialului lui n.

IB.06.9.3 Utilizarea recursivitatii în implementarea algoritmilor de tip *Divide et Impera*

Tehnica ***Divide et Impera***, fundamentală în elaborarea algoritmilor, constă în descompunerea unei probleme complexe în mai multe subprobleme a căror rezolvare e mai simplă şi din soluțiile cărora se poate determina soluția problemei inițiale (exemple: găsirea minimului si maximului valorilor elementelor unui tablou, căutarea binară, sortare Quicksort, turnurile din Hanoi).

Un algoritm de divizare general s-ar putea scrie:

```
void rezolva ( problema pentru x ) {
     dacă x e divizibil in subprobleme  {
        divide pe x in parti x1,...,xk
        rezolva(x1);
         ...
         rezolva(xk);
        combina solutiile partiale intr-o solutie pentru x
     }
     altfel
        rezolva pe x direct
```

Vom oferi ca exemplu căutarea binară a unei valori *v* într-un vector ordonat *a*, prin metoda înjumătățirii intervalului de căutare. Se returnează indicele în vector sau *-1* dacă valoarea *v* nu se află în vector. Pentru o mai bună înțelegere, prima variantă este cea iterativă.

```
// varianta iterative, l- limita stanga, r – limita dreapta a intervalului
int bsearchIter(int v, int *a, int l, int r) {
   while(l<r) {
      int m = (l+r)/2;          // m – mijlocul intervalului
      if (v == a[m]) return m; // element gasit
      if (v > a[m]) l=m+1;     // reduc intervalul la jumatatea lui dreapta
      else r=m;                // reduc intervalul la jumatatea lui stanga
      }
   return -1;                  // element negasit
}

// varianta recursiva
int bsearchRec (int v, int *a,int l, int r) {
      int m;
      if (l<r) {
         int m = (l+r)/2;
         if (v == a[l]) return l;
         else
            if ( v>a[m] )
                 return bsearchRec(v, a, m+1, r);
            else
                 return bsearchRec(v, a, l, m);
      }
      else return -1;
}

int main() {
    int v, a[N], n;
    …… // citire date intrare
    printf("Elem. %d se afla in poz %d\n ", v, bsearchRec(v,a,0,n));
    printf("Elem. %d se afla in poz %d\n ", v, bsearchIter(v,a,0,n));
    return 0;
}
```

IB.06.10. Anexă: Funcții în C++

În C++ toate funcțiile folosite trebuie declarate şi nu se mai consideră că o funcție nedeclarată este implicit de tipul *int*.
Absența unei declarații de funcții este eroare gravă în C++ şi nu doar avertisment ca în C (nu trece de compilare)!

În C++ se pot declara valori implicite pentru parametrii formali de la sfârşitul listei de parametri; aceste valori sunt folosite automat în absența parametrilor efectivi corespunzători la un apel de funcție. O astfel de funcție poate fi apelată deci cu un număr variabil de parametri.
Exemplu:

```
// afişare vector, precedat de un titlu (şirul primit sau şirul nul)
void printv ( int v[], int n, char * titlu="") {
   printf ("\n %s \n", titlu);
   for (int i=0; i<n; i++)
     printf ("%d ", v[i]);
}

// Exemple de apeluri:
printv ( x,nx );                        // cu 2 parametri
printv (a,na," multimea A este");       // cu 3 parametri
```

În C++ funcțiile scurte pot fi declarate inline, iar compilatorul înlocuieşte apelul unei funcții inline cu instrucțiunile din definiția funcției, eliminând secvențele de transmitere a parametrilor. Funcțiile inline sunt tratate ca şi macrourile definite cu *define*. Orice funcție poate fi declarată inline, dar compilatorul poate decide că anumite funcții nu pot fi tratate inline şi sunt tratate ca funcții obişnuite. De exemplu, funcțiile care conțin cicluri nu pot fi inline. Utilizarea unei funcții inline nu se deosebeşte de aceea a unei funcții normale. Exemplu de funcție *inline*:

```
inline int max (int a, int b) { return a > b ? a : b; }
```

În C++ pot exista mai multe funcții cu acelaşi nume dar cu parametri diferiți (ca tip sau ca număr). Se spune că un nume este *supraîncărcat* cu semnificații ("function overloading"). Compilatorul poate stabili care din funcțiile cu acelaşi nume a fost apelată într-un loc analizând lista de parametri şi tipul funcției:

```
float abs (float f) { return fabs(f); }
long abs (long x) { return labs(x); }
printf ("%6d%12ld  %f \n", abs(-2),abs(-2L),abs(-2.5) );
```