FUNCȚII

6

6.1. Structura unei funcții
6.2. Apelul şi prototipul unei funcții
6.3. Transferul parametrilor unei funcții
6.3.1.Transferul parametrilor prin valoare
6.3.2.Transferul prin pointeri
6.3.3.Transferul prin referință
6.3.4.Transferul parametrilor către funcția main
6.4. Tablouri ca parametri
6.5. Funcții cu parametri impliciți
6.6. Funcții cu număr variabil de parametri
6.7. Funcții predefinite
6.8. Clase de memorare
6.9. Moduri de alocare a memoriei
6.10.Funcții recursive
6.11.Pointeri către funcții

6.1. STRUCTURA UNEI FUNCȚII

Un program scris în limbajul C/C++ este *un ansamblu de funcții*, fiecare dintre acestea efectuând o activitate bine definită. Din punct de vedere conceptual, ***funcția*** reprezintă *o aplicație definită pe o mulțime D (D=mulțimea, domeniul de definiție), cu valori în mulțimea C (C=mulțimea de valori, codomeniul), care îndeplineşte condiția că oricărui element din D îi corespunde un unic element din C.*

Funcțiile *comunică prin argumennte*: ele primesc ca parametri (argumente) datele de intrare, efectuează prelucrările descrise în corpul funcției asupra acestora şi pot returna o valoare (rezultatul, datele de ieşire). Execuția programului începe cu funcția principală, numită **`main`**. Funcțiile pot fi descrise în cadrul aceluiaşi fişier, sau în fişiere diferite, care sunt testate şi compilate separat, asamblarea lor realizându-se cu ajutorul linkeditorului de legături.

O funcție este formata din antet si corp:

```
antet_funcție
{
     corpul_funcției
}
```

Sau:

```
tip_val_return nume_func (lista_declarațiilor_param_ formali)
{
     declarații_variabile_locale
     instrucțiuni
     return valoare
}
```

Prima linie reprezintă ***antetul*** funcției, în care se indică: tipul funcției, numele acesteia şi lista declarațiilor parametrilor formali. La fel ca un operand sau o expresie, o funcție are un *tip*, care este dat de tipul valorii returnate de funcție în funcția apelantă. Dacă funcția nu întoarce nici o valoare, în locul *tip_vali_return* se specifică **`void`**. Dacă *tip_val_return* lipseşte, se consideră, implicit, că acesta este **`int`**. *Nume_funcție* este un identificator.

Lista_declarațiilor_param_formali (încadrată între paranteze rotunde) constă într-o listă (enumerare) care conține tipul şi identificatorul fiecărui parametru de intrare, despărțite prin virgulă. Tipul unui parametru poate fi oricare, chiar şi tipul pointer. Dacă lista parametrilor formali este vidă, în antet, după numele funcției, apar doar parantezele *( )*, sau ***(void)***.

Corpul funcției este un bloc, care implementează algoritmul de calcul folosit de către funcție. În corpul funcției apar (în orice ordine) declarații pentru variabilele locale şi instrucțiuni. Dacă funcția întoarce o

valoare, se foloseşte instrucţiunea ***return*** *valoare*. La execuţie, la întâlnirea acestei instrucţiuni, se revine în funcţia apelantă.
În limbajul C/C++ se utilizează ***declaraţii şi definiţii*** de funcţii.
Declaraţia conţine antetul funcţiei şi informează compilatorul asupra tipului, numelui funcţiei şi a listei parametrilor formali (în care se poate indica doar tipul parametrilor formali, nu şi numele acestora). Declaraţiile de funcţii se numesc ***prototipuri***, şi sunt constituite din antetul funcţiei, din care pot lipsi numele parametrilor formali.
Definiţia conţine antetul funcţiei şi corpul acesteia. Nu este admisă definirea unei funcţii în corpul altei funcţii.

O formă învechită a antetului unei funcţii este aceea de a specifica în lista parametrilor formali doar numele acestora, nu şi tipul. Această libertate în omiterea tipului parametrilor constituie o sursă de erori.

```
tipul_valorii_returnate nume_funcţie (lista_parametrilor_ formali)
declararea_parametrilor_formali
{
declaraţii_variabile_locale
instrucţiuni
return valoare
}
```

6.2. APELUL ŞI PROTOTIPUL FUNCŢIILOR

O funcţie poate fi ***apelată*** printr-o construcţie urmată de punct şi virgulă, numită ***instrucţiune de apel***, de forma:

```
nume_funcţie (lista_parametrilor_efectivi);
```

Parametrii efectivi trebuie să corespundă cu cei formali ca ordine şi tip. La apel, se atribuie parametrilor formali valorile parametrilor efectivi, după care se execută instrucţiunile din corpul funcţiei. La revenirea din funcţie, controlul este redat funcţiei apelante, şi execuţia continuă cu instrucţiunea următoare instrucţiunii de apel, din funcţia apelantă. O altă posibilitate de a apela o funcţie este aceea în care apelul funcţiei constituie operandul unei expresii. Acest lucru este posibil doar în cazul în care funcţia returnează o valoare, folosită în calculul expresiei.

Parametrii declaraţi în antetul unei funcţii sunt numiţi ***formali***, pentru a sublinia faptul că ei nu reprezintă valori concrete, ci numai ţin locul acestora pentru a putea exprima procesul de calcul realizat prin funcţie. Ei se concretizează la execuţie prin apelurile funcţiei.
Parametrii folosiţi la apelul unei funcţii sunt ***parametri reali, efectivi, concreţi***, iar valorile lor vor fi atribuite parametrilor formali, la execuţie. Utilizarea parametrilor formali la implementarea funcţiilor şi atribuirea de valori concrete pentru ei, la execuţie, reprezintă un prim nivel de abstractizare în programare. Acest mod de programare se numeşte ***programare procedurală*** şi realizează un proces de ***abstractizare prin parametri***.

Variabilele declarate în interiorul unei funcţii, cât şi parametrii formali ai acesteia nu pot fi accesaţi decât în interiorul acesteia. Aceste variabile sunt numite *variabile locale* şi nu pot fi accesate din alte funcţii. Domeniul de vizibilitate a unei variabile este porţiunea de cod la a cărei execuţie variabila respectivă este accesibilă. Deci, domeniul de vizibilitate a unei variabile locale este funcţia în care ea a fost definită (vezi şi paragraful 6.8.).

Exemplu:

```
int f1(void)
{ double a,b; int c;
. . .
return c; // a, b, c - variabile locale, vizibile doar în corpul funcţiei
}
void main()
{ . . . . . .        // variabile a şi b nu sunt accesibile în main()
}
```

Dacă în interiorul unei funcţii există instrucţiuni compuse (blocuri) care conţin declaraţii de variabile, aceste variabile nu sunt vizibile în afara blocului.

Exemplu:

```
void main()
{ int a=1, b=2;
 cout << "a="<<a<<" b="<<b<<" c="<<c'\n';          // a=1 b=2, c nedeclarat
        . . . . . . . .
      { int a=5; b=6; int c=9;
      cout << "a="<<a<<" b="<<b<<'\n';          // a=5 b=6  c=9
        . . . . . . . .
      }
cout << "a="<<a<<" b="<<b<<" c="<<c'\n';          // a=1 b=6, c nedeclarat
. . . . . . . . . . . .
}
```

Exerciţiu: Să se scrie următorul program (pentru înţelegerea modului de apel al unei funcţii) şi să se urmărească rezultatele execuţiei acestuia.

```
#include <iostream.h>
void f_afis(void)
{
cout<<"Se execută instrucţiunile din corpul funcţiei\n";
double a=3, b=9.4; cout<<a<<"*"<<b<<"="<<a*b<<'\n';
cout<<"Ieşire din funcţie!\n"; }

void main()
{
cout<<"Intrare în funcţia principală\n";
f_afis( );          //apelul funcţiei f_afis, printr-o instrucţiune de apel
cout<<"Terminat MAIN!\n"; }
```

Exerciţiu: Să se scrie un program care citeşte două numere şi afişează cele mai mare divizor comun al acestora, folosind o funcţie care îl calculează.

```
#include <iostream.h>
int cmmdc(int x, int y)
{
if (x==0 || y==1 ||  x==1 || y==0)  return 1;
if (x<0)    x=-x;
if (y<0)    y=-y;
while (x != 0){
      if ( y > x )
            {int z=x; x=y; y=z; }
      x-=y;        // sau: x%=y;
}
return y;}

void main()
{
int n1,n2; cout<<"n1=";cin>>n1; cout<<"n2=";cin>>n2;
int diviz=cmmdc(n1,n2);
cout<<"Cel mai mare divizor comun al nr-lor:"<<n1<<" şi ";
cout<<n2<<" este:"<<diviz<<'\n';
/* sau:
cout<<"Cel mai mare divizor comun al nr-lor:"<<n1<<" şi ";
cout<<n2<<" este:"<<cmmdc(n1,n2)<<'\n';*/   }
```

Exerciţiu: Să se calculeze valoarea lui y, u şi m fiind citite de la tastatură:

$z=2\omega(2\varphi(u)+1, m)+\omega(2u^2-3, m+1)$, unde:

$\omega(x,n) = \sum_{i=1}^{n}\sin(ix)\cos(2ix)$, $\varphi(x) = \sqrt{1+e^{-x^2}}$, $\omega: R \times N \rightarrow R$, $\varphi: R \rightarrow R$

```
#include <iostream.h>
#include <math.h>
double omega(double x, int n)
{ double s=0; int i;
for (i=1; i<=n; i++)    s+=sin(i*x)*cos(i*x);
return s;
}

double psi( double x)
{     return sqrt( 1 + exp (- pow (x, 2)));}

void main()
{double u, z; int m; cout<<"u="; cin>>u; cout<<"m="; cin>>m;
z=2*omega(2* psi(u) + 1, m) + omega(2*pow(u,2) - 3, m+1);
cout<<"z="<<z<<'\n'; }
```

În exemplele anterioare, înainte de apelul funcţiilor folosite, acestea au fost definite (antet+corp). Există cazuri în care definirea unei funcţii nu poate fi făcută înaintea apelului acesteia (cazul funcţiilor care se apelează unele pe altele). Să rezolvăm ultimul exerciţiu, folosind declaraţiile funcţiilor omega şi psi, şi nu definiţiile lor.

Exerciţiu:

```
#include <iostream.h>
#include <math.h>
double omega(double, int);
// prototipul funcţiei omega - antet din care lipsesc numele parametrilor formali
double psi(double);              // prototipul funcţiei psi

void main()
{double u, z; int m; cout<<"u="; cin>>u; cout<<"m="; cin>>m;
z=2*omega(2* psi(u) + 1, m) + omega(2*pow(u,2) - 3, m+1);
cout<<"z="<<z<<'\n'; }

double omega(double x, int i);        // definiţia funcţiei omega
{ double s=0; int i;
for (i=1; i<=n; i++)    s +=  sin (i*x) * cos (i*x);
return s;   }

double psi( double x)            // definiţia funcţiei psi
{     return sqrt( 1 + exp (- pow (x, 2))); }
```

Prototipurile funcţiilor din biblioteci (predefinite) se găsesc în headere. Utilizarea unei astfel de funcţii impune doar includerea în program a headerului asociat, cu ajutorul directivei preprocesor ***#include***.
Programatorul îşi poate crea propriile headere, care să conţină declaraţii de funcţii, tipuri globale, macrodefiniţii, etc.

Similar cu declaraţia de variabilă, domeniul de valabilitate (vizibilitate) a unei funcţii este:
- fişierul sursă, dacă declaraţia funcţiei apare în afara oricărei funcţii (la nivel global);
- funcţia sau blocul în care apare declaraţia.

6.3. TRANSFERUL PARAMETRILOR UNEI FUNCŢII

Funcţiile comunică între ele prin argumente (parametrii).

Există următoarele moduri de transfer (transmitere) a parametrilor către funcţiile apelate:
- Transfer prin valoare;
- Transfer prin pointeri;
- Transfer prin referinţă.

6.3.1. TRANFERUL PARAMETRILOR PRIN VALOARE

În exemplele anterioare, parametrii de la funcţia apelantă la funcţia apelată au fost transmişi ***prin valoare***. De la programul apelant către funcţia apelată, prin apel, se transmit valorile partametrilor efectivi, reali. Aceste valori vor fi atribuite, la apel, parametrilor formali. Deci procedeul de transmitere a parametrilor prin valoare constă *în încărcarea valorii parametrilor efectivi în zona de memorie a parametrilor formali (în stivă)*. La apelul unei funcţii, parametrii reali trebuie să corespundă - ca ordine şi tip - cu cei formali.

Exerciţiu: Să se scrie următorul program (care ilustrează legătura dintre pointeri şi vectori) şi să se urmărească rezultatele execuţiei acestuia.

```
void f1(float intr,int nr)// intr, nr - parametri formali
{
for (int i=0; i<nr;i++) intr *= 2.0;
cout<<"Val. Param. intr="<<intr<<'\n';// intr=12
}
void main()
{
float data=1.5; f1(data,3);
// apelul funcţiei f1; data, 3 sunt parametri efectivi
cout<<"data="<<data<<'\n';
// data=1.5 (nemodificat)
}
```

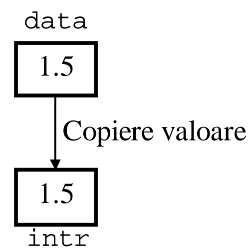


Figura 6.1. Transmiterea prin valoare

Fiecare argument efectiv utilizat la apelul funcţiei este evaluat, iar valoarea este atribuită parametrului formal corespunzător. În interiorul funcţiei, o copie locală a acestei valori va fi memorată în parametrul formal. O modificare a valorii parametrului formal în interiorul funcţiei (printr-o operaţie din corpul funcţiei), nu va modifica valoarea parametrului efectiv, ci doar valoarea parametrului formal, deci a copiei locale a parametrului efectiv (figura 6.1.). Faptul că variabila din programul apelant (parametrul efectiv) şi parametrul formal sunt obiecte distincte, poate constitui un mijloc util de protecţie. Astfel, în funcţia `f1`, valoarea parametrului formal intr este modificată (alterată) prin instrucţiunea ciclică for. În schimb, valoarea parametrului efectiv (data) din funcţia apelantă, rămâne nemodificată.

În cazul transmiterii parametrilor prin valoare, parametrii efectivi pot fi chiar expresii. Acestea sunt evaluate, iar valoarea lor va iniţializa, la apel, parametrii formali.

Exemplu:

```
double psi(int a, double b)
{
if (a > 0)  return a*b*2;
else  return -a+3*b; }
void main()
{ int x=4; double y=12.6, z; z=psi ( 3*x+9, y-5) + 28;
cout<<"z="<<z<<'\n';    }
```

Transferul valorii este însoţit de eventuale conversii de tip. Aceste conversii sunt realizate automat de compilator, în urma verificării apelului de funcţie, pe baza informaţiilor despre funcţie, sau sunt conversii explicite, specificate de programator, prin operatorul "cast".

Exemplu:

```
float f1(double, int);
void main()
{
int a, b; float g=f1(a, b);     // conversie automată: int a -> double a
float h=f1( (double) a, b);     // conversie explicită
}
```

Limbajul ***C*** este numit ***limbajul apelului prin valoare***, deoarece, de fiecare dată când o funcţie transmite argumente unei funcţii apelate, este transmisă, de fapt, o copie a parametrilor efectivi. În acest mod, dacă

valoarea parametrilor formali (iniţializaţi cu valorile parametrilor efectivi) se modifică în interiorul funcţiei apelate, valorile parametrilor efectivi din funcţia apelantă nu vor fi afectate.

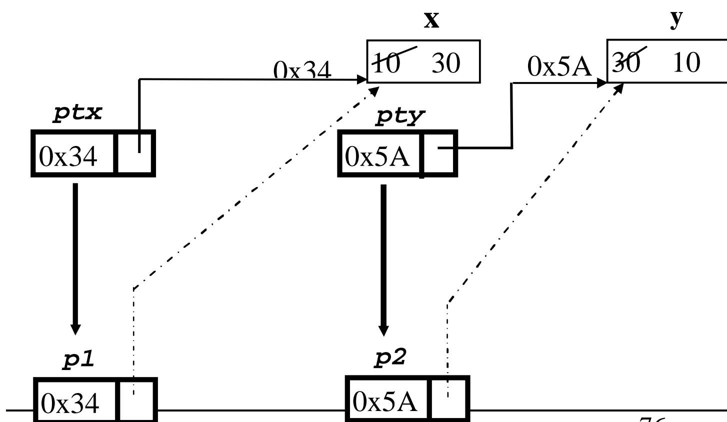
6.3.2. TRANSFERUL PARAMETRILOR PRIN POINTERI

În unele cazuri, parametrii transmişi unei funcţii pot fi pointeri (variabile care conţin adrese). În aceste cazuri, parametrii formali ai funcţiei apelate vor fi iniţializaţi cu valorile parametrilor efectivi, deci cu valorile unor adrese. Astfel, *funcţia apelată* ***poate modifica*** *conţinutul locaţiilor spre care pointează argumentele (pointerii).*

Exerciţiu: Să se citească 2 valori întregi şi să se interschimbe cele două valori. Se va folosi o funcţie de interschimbare.

```
#include <iostream.h>
void schimbă(int *, int *);            //prototipul functiei schimba
void main()
{
int x, y, *ptx, *pty;   ptx=&x;     pty=&y;
cout<<"x=";cin>>x;cout<<"y=";cin>>y;cout<<"x="<<x;cout<<"y="<<y<<'\n';
cout<<"Adr. lui x:"<<&x<<" Val lui x:"<<x<<'\n';
cout<<"Adr.lui y:"<<&y<<" Val y:"<<y<<'\n'; cout<<"Val. lui ptx:"<<ptx;
cout<<" Cont. locaţiei spre care pointează ptx:"<<*ptx<<'\n';
cout<<"Val. lui pty:"<<pty;
cout<<"Cont. locaţiei spre care pointează pty:"<<*pty;
schimba(ptx, pty);
// SAU: schimba(&x, &y);
cout<<"Adr. lui x:"<<&x<<" %x Val lui x: %d\n", &x, x);
cout<<"Adr. y:"<<&y<<" Val lui y:"<<y<<'\n';cout<<"Val. lui ptx:"<<ptx;
cout<<" Cont. locaţiei spre care pointează ptx:"<<*ptx<<'\n';
cout<<"Val. lui pty:"<<pty;
cout<<" Cont. locaţiei spre care pointează pty:"<<*pty<<'\n';
}
void schimbă( int *p1, int *p2)
{
cout<<"Val. lui p1:"<<p1;
cout<<" Cont. locaţiei spre care pointează p1:"<<*p1<<'\n';
cout<<"Val. lui p2:"<<p2;
cout<<" Cont. locaţiei spre care pointează p2:"<<*p2<<'\n';
int t = *p1;        // int *t ; t=p1;
*p2=*p1;    *p1=t;
cout<<"Val. lui p1:"<<p1;
cout<<" Cont. locaţiei spre care pointează p1:"<<*p1<<'\n';
cout<<"Val. lui p2:"<<p2;
cout<<" Cont. locaţiei spre care pointează p2:"<<*p2<<'\n';
}
```

Dacă parametrii funcţiei `schimbă` ar fi fost transmişi prin valoare, această funcţie ar fi interschimbat copiile parametrilor formali, iar în funcţia `main` modificările asupra parametrilor transmişi nu s-ar fi păstrat. În figura 6.2. sunt prezentate mecanismul de transmitere a parametrilor (prin pointeri) şi modificările efectuate asupra lor de către funcţia `schimbă`.



Parametrii formali `p1` şi `p2`, la apelul funcţiei `schimbă`, primesc valorile parametrilor efectivi `ptx` şi `pty`, care reprezintă adresele variabilelor `x` şi `y`. Astfel, variabilele pointer `p1` şi `ptx`, respectiv `p2` şi `pty` pointează către `x` şi `y`. Modificările asupra valorilor variabilelor `x` şi `y` realizate în corpul funcţiei `schimbă`, se păstrează şi în funcţia `main`.

Figura 6.2. Transmiterea parametrilor unei funcţii prin pointeri

Exercițiu: Să se scrie următorul program şi să se urmărească rezultatele execuției acestuia.

```
#include <iostream.h>
double omega (long *k)
{
cout<<"k=", k);
// k conține adr. lui i
cout<<"*k=";
cout<<k<<'\n';
// *k = 35001
double s=2+(*k)-3;
// s = 35000
cout<<"s="<<s<<'\n';
*k+=17; // *k = 35017
cout<<"*k="<<*k;
cout<<'\n';
return s;   }

void main()
{
long i = 35001; double w;
cout<<"i="<<i;cout<<"Adr.lui i:"<<&i<<'\n';
w=omega(&i); cout<<"i="<<i<< w="<<w<<'\n'; // i = 350017 w = 35000
}
```

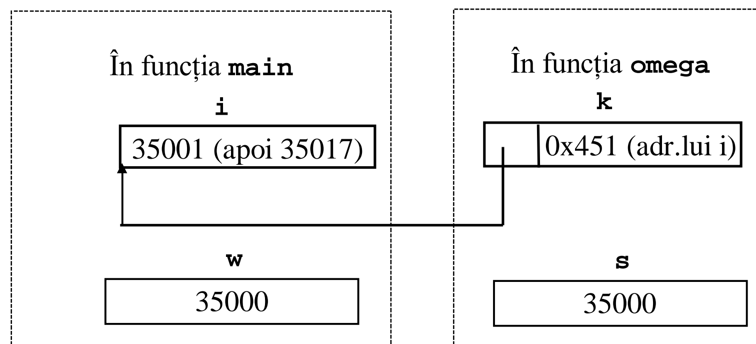


Figura 6.3. Transferul parametrilor prin pointeri

6.3.2.1. Funcții care returnează pointeri

Valoarea returnată de o funcție poate fi pointer, aşa cum se observă în exemplul următor:

Exemplu:

```
#include <iostream.h>
double *f (double *w, int k)
{   // w conține adr. de început a vectorului a
cout<<"w="<<w<<" *w="<<*w<<'\n';        // w= adr. lui a ;*w = a[0]=10
return w+=k;
/*incrementeaza pointerului w cu 2(val. lui k); deci w pointează către elementul de indice 2 din vectorul a*/
}
void main()
{double a[10]={10,1,2,3,4,5,6,7,8,9}; int i=2;
cout<<"Adr. lui a este:"<<a;
double *pa=a;        // double *pa; pa=a;
cout<<"pa="<<pa<<'\n'     // pointerul pa conține adresa de început a tabloului a
// a[i] = * (a + i)
// &a[i] = a + i
pa=f(a,i); cout<<"pa="<<pa<<" *pa="<<*pa<<'\n';
// pa conține adr. lui a[2], adica adr. a + 2 * sizeof(double);
*pa=2;
}
```

6.3.3. TRANSFERUL PARAMETRILOR PRIN REFERINȚĂ

În acest mod de transmitere a parametrilor, unui parametru formal i se poate asocia (atribui) chiar obiectul parametrului efectiv. Astfel, parametrul efectiv poate fi modificat direct prin operaţiile din corpul funcţiei apelate.

În exemplul următor definim variabila `br`, ***variabilă referinţă*** către variabila `b`. Variabilele *b* şi *br* se găsesc, în memorie, la *aceeaşi* adresă şi sunt variabile sinonime.

Exemplu:

```
#include <stdio.h>
#include <iostream.h>
void main()
{
int b,c;
int &br=b;   //br referinţă la altă variabilă (b)
br=7;
cout<<"b="<<b<<'\n';       //b=7
cout<<"br="<<br<<'\n';     //br=7
cout<<"Adr. br este:"<<&br;           //Adr. br este:0xfff4
printf("Adr. b este:"<<&b<<'\n';      //Adr. b este:0xfff4
b=12; cout<<"br="<<br<<'\n';          //br=12
cout<<"b="<<b<<'\n';                  //b=12
c=br; cout<<"c="<<c<<'\n';            //c=12
}
```

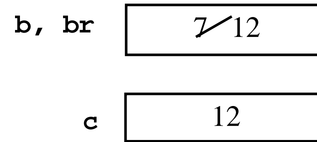


Figura 6.4. Variabilele referinţă `b, br`

Exemplul devenit clasic pentru explicarea apelului prin referinţă este cel al funcţiei de permutare (interschimbare) a două variabile.

Fie funcţia *schimb* definită astfel:

```
void schimb (double x, double y)
{ double t=x; x=y; y=t; }

void main()
{ double a=4.7, b=9.7;
. . . . . . . . . . .
schimb(a, b);            // apel funcţie
. . . . . . . . . . .  }
```

Parametri funcţiei `schimb` sunt transmişi prin valoare: parametrilor formali `x`, `y` li se atribuie (la apel) valorile parametrilor efectivi `a`, `b`. Funcţia `schimb` permută valorile parametrilor formali `x` şi `y`, dar permutarea nu are efect asupra parametrilor efectivi `a` şi `b`.

Pentru ca funcţia de interschimbare să poată permuta valorile parametrilor efectivi, în limbajul C/C++ parametrii formali trebuie să fie *pointeri către valorile care trebuie inter*schimbate:

```
void pschimb(double *x, double *y)
{ int t=*x; *x=*y; *y=t; }
void main()

{ double a=4.7, b=9.7;
. . . . . . . . . . .
pschimb(&a, &b);         // apel funcţie
/* SAU:
double *pa, *pb;
pa=&a; pb=&b;
pschimb(pa, pb);*/
. . . . . . . . . . .   }
```

Se atribuie pointerilor `x` şi `y` valorile pointerilor `pa`, `pb`, deci adresele variabilelor `a` şi `b`. Funcţia `pschimb` permută valorile spre care pointează pointerii `x` şi `y`, deci valorile lui `a` şi `b` (figura 6.5).

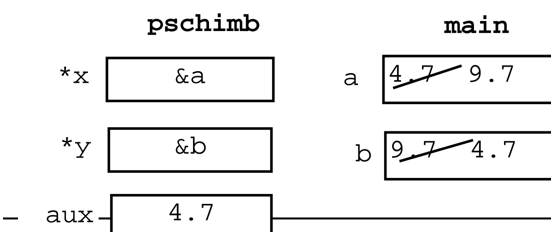


Figura 6.5. Transferul parametrilor prin pointeri

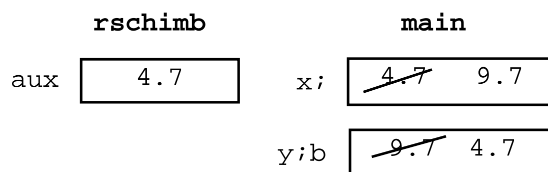


Figura 6.6. Transferul parametrilor prin referinţă

În limbajul C++ aceeaşi funcţie de permutare se poate defini cu parametri formali de *tip referinţă*.

```
void rschimb(double &x, double &y)
{ int t=x; x=y; y=t; }
void main()
{ double a=4.7, b=9.7;
. . . . . . . . . . . . . . .
rschimb(a, b);          // apel funcţie
. . . . . . . . . . . . . . .   }
```

În acest caz, x şi y sunt sinonime cu a şi b (nume diferite pentru aceleaşi grupuri de locaţii de memorie). Interschimbarea valorilor variabilelor de x şi y înseamnă interschimbarea valorilor variabilelor a şi b (fig. 6.6.).

Comparând funcţiile `pschimb` şi `rschimb`, se observă că diferenţa dintre ele constă în modul de declarare a parametrilor formali. În cazul funcţiei `pschimb` parametrii formali sunt *pointeri* (de tip *double), în cazul funcţiei `rschimb`, parametrii formali sunt *referinţe* către date de tip double. În cazul transferului parametrilor prin referinţă, parametrii formali ai funcţiei referă aceleaşi locaţii de memorie (sunt sinonime pentru) parametrii efectivi.

Comparând cele trei moduri de transmitere a parametrilor către o funcţie, se poate observa:
1. La apelul *prin valoare* transferul datelor este *unidirecţional*, adică valorile se transferă numai de la funcţia apelantă către cea apelată. La apelul *prin referinţă* transferul datelor este *bidirecţional*, deoarece o modificare a parametrilor formali determină modificarea parametrilor efectivi, care sunt sinonime (au nume diferite, dar referă aceleaşi locaţii de memorie).
2. La transmiterea parametrilor *prin valoare*, ca parametrii efectivi pot apare *expresii* sau *nume de variabile*. La transmiterea parametrilor *prin referinţă*, ca parametri efectivi nu pot apare expresii, ci *doar nume de variabile*. La transmiterea parametrilor *prin pointeri*, ca parametri efectivi pot apare expresii de pointeri.
3. Transmiterea parametrilor unei funcţii prin referinţă este specifică limbajului C++.
4. Limbajul C este numit limbajul apelului prin valoare. Apelul poate deveni, însă, *apel prin referinţă* în cazul variabilelor simple, folosind pointeri, sau aşa cum vom vedea în paragraful 6.4., în cazul în care parametru efectiv este un tablou.
5. În limbajul C++ se poate alege, pentru fiecare parametru, tipul de apel: prin valoare sau prin referinţă, aşa cum ilustrează exemplele următoare:

Exerciţiu: Să se scrie următorul program şi să se urmărească rezultatele execuţiei acestuia.

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

double func(int a, double b, double *c, double &d)
{cout<<"********************** func ****************\n";
cout<<"a="<<a<<" b="<<b;            //a=7 (a=t prin val); b=21 (b=u prin val)
cout<<" c="<<c<<" *c="<<*c<<'\n';   // c=ffe(c=w=&u) *c=21
cout<<" d="<<d;                     //d=17
cout<<"Adr d="<<&d<<'\n';           //Adr d=ffe6 (adr d=adr v)
a+=2; cout<<"a="<<a<<'\n';               //a=9
d=2*a+b; cout<<"d="<<d<<'\n';                //d=39
/*c=500;
cout<<" c="<<c<<" *c="<<*c<<'\n';     // c=ffe(c=w=&u) *c=21*/
cout<<"********************** func ****************\n";
```

```
return b+(*c);
}

void main()
{cout<<"\n\n \n MAIN MAIN";
int t=7; double u=12, v=17, *w, z; cout<<"u="<<u<<'\n';       //u=12
w=&u; *w=21;
cout<<"t="<<t<<" u="<<u<<" v="<<v;   //t=7 u=12 v=17 *w=21
cout<<" *w="<<*w<<" u="<<u<<'\n';    //*w=21 u=21
printf("w=%x Adr. u=%x\n",w,&u);     //w=ffee Adr. u=ffee
printf("v=%f Adr v=%x\n",v,&v);      //v=17.000 Adr v=ffe6
z=func(t,u,w, v);
cout<<"t="<<t<<"u="<<u<<"v="<<v;     //t=7 u=21 (NESCHIMBATI) v=39 (v=d)
cout<<" *w="<<*w<<" z="<<z<<'\n';    //*w=21 w=ffee z=42
printf(" w=%x\n",w);
}
```

Exemplul ilustrează următoarele probleme:
La apelul funcției `func`, parametrii `t` şi `u` sunt transmişi prin valoare, deci valorile lor vor fi atribuite parametrilor formali `a` şi `b`. Orice modificare a parametrilor formali `a` şi `b`, în funcția `func`, nu va avea efect asupra parametrilor efectivi `t` şi `u`. Al treilea parametru formal al funcției `func` este transmis prin pointeri, deci `c` este de tip `double *` (pointer către un real), sau `*c` este de tip `double`.

La apelul funcției, valoarea pointerului `w` (adresa lui `u` : `w=&u`) este atribuită pointerului `c`. Deci pointerii `w` şi `c` conțin aceeaşi adresă, pointând către un real. Dacă s-ar modifica valoarea spre care pointează `c` în `func` (vezi instrucțiunile din comentariu `*c=500`), această modificare ar fi reflectată şi în funcția apelantă, deoarece pointerul `w` are acelaşi conținut ca şi pointerul `c`, deci pointează către aceeaşi locație de memorie. Parametrul formal `d` se transmite prin referință, deci, în momentul apelului, `d` şi `v` devin similare (`d` şi `v` sunt memorate la aceeaşi adresă). Modificarea valorii variabilei `d` în `func` se reflectă, deci, şi asupra parametrului efectiv din funcția `main`.

Exercițiu: Să se scrie următorul program (care ilustrează legătura dintre pointeri şi vectori) şi să se urmărească rezultatele execuției acestuia.

```
#include <iostream.h>
#include <stdio.h>
double omega(long &k)
{printf("Adr k=%x Val k=%ld\n",&k,k);          //Adr k=fff2 Val k=200001
double s=2+k-3;cout<<"s="<<s<<'\n';            //s=200000
k+=17;printf("Adr k=%x Val k=%ld\n",&k,k);     //Adr k=fff2 Val k=200018
return s;
}
void main()
{long a=200001;
printf("Adr a=%x Val a=%ld\n",&a,a);       //Adr a=fff2 Val a=200001
double w=omega(a); cout<<"w="<<w<<'\n';    //s=200000
}
```

Aşa cum s-a prezentat în paragrafele 2.5.3.2. şi 5.6., modificatorii sunt cuvinte cheie utilizați în declarații sau definiții de variabile sau funcții. Modificatorul de acces **`const`** poate apare în:

- Declarația unei variabile (precede tipul variabilei) restricționând modificarea valorii datei;
- La declararea variabilelor pointeri definind pointeri constanți către date neconstante, pointeri neconstanți către date constante şi pointeri constanți către date constante.
- În *lista declarațiilor parametrilor formali ai unei funcții*, conducând la imposibilitatea de a modifica valoarea parametrului respectiv în corpul funcției, ca în exemplul următor:

Exemplu:

```
#include <iostream.h>
#include <stdio.h>
int func(const int &a)
{printf("Adr a=%x Val a=%d\n",&a,a);int b=2*a+1;
//modificarea valorii parametrului a nu este permisă
cout<<"b="<<b<<'\n';return b;}
void main()
{const int c=33;int u;printf("Adr c=%x Val c=%d\n",&c,c);
u=func(c);cout<<"u="<<u<<'\n'; }
```

6.3.4. TRANSFERUL PARAMETRILOR CĂTRE FUNCŢIA main

În situaţiile în care se doreşte transmiterea a unor informaţii (opţiuni, date iniţiale, etc) către un program, *la lansarea în execuţie* a acestuia, este necesară definirea unor parametri către funcţia `main`. Se utilizează trei parametrii speciali: `argc`, `argv` şi `env`. Trebuie inclus headerul **`stdarg.h`**.
Prototipul funcţiei main cu parametri în linia de comandă este:

`main (int argc, char *argv[ ], char *env[ ])`

Dacă ***nu*** *se lucrează cu un mediu de programare integrat*, argumentele transmise către funcţia main trebuie editate (specificate) în linia de comandă prin care se lansează în execuţie programul respectiv. Linia de comandă tastată la lansarea în execuţie a programului este formată din grupuri de caractere delimitate de spaţiu sau tab. Fiecare grup este memorat într-un şir de caractere. Dacă se lucrează cu un mediu integrat (de exemplu, BorlandC), selecţia comanzii `Arguments`... din meniul `Run` determină afişarea unei casete de dialog în care utilizatorul poate introduce argumentele funcţiei `main`.

- Adresele de început ale acestor şiruri sunt memorate în tabloul de pointeri `argv[ ]`, în ordinea în care apar în linia de comandă (`argv[0]` memorează adresa şirului care constituie numele programului, `argv[1]` - adresa primului argument, etc.).
- Parametrul întreg `argc` memorează numărul de elemente din tabloul argv (`argc>=1`).
- Parametrul `env[ ]` este un tablou de pointeri către şiruri de caractere care pot specifica parametri ai sistemului de operare.

Funcţia `main` poate returna o valoare întreagă. În acest caz în antetul funcţiei se specifică la tipul valorii returnate `int,` sau nu se specifică nimic (implicit, tipul este int), iar în corpul funcţiei apare instrucţiunea *return valoare_intreagă*;. Numărul returnat este transferat sistemului de operare (programul apelant) şi poate fi tratat ca un cod de eroare sau de succes al încheierii execuţiei programului.

<u>Exerciţiu:</u> Să se implementeze un program care afişează argumentele transmise către funcţia main.

```
#include <iostream.h>
#include <stdarg.h>
void main(int argc, char *argv[], char *env[])
{cout<<"Nume program:"<<argv[0]<<'\n';//argv[0] contine numele programului
if(argc==1)
      cout<<"Lipsa argumente!\n";
else
      for (int i=1; i<argc; i++){
            cout<<"Argumentul "<<i<<":"<<argv[i]<<'\n';
}
```

6.4. TABLOURI CA PARAMETRI

În limbajul C, cazul parametrilor tablou constituie o excepţie de la regula transferului parametrilor prin valoare. Numele unui tablou reprezintă, de fapt, adresa tabloului, deci a primului element din tablou.

Exerciţiu: Să se afle elementul minim dintr-un vector de maxim 10 elemente. Se vor scrie două funcţii: de citire a elementelor vectorului şi de aflare a elementului minim:

```
#include <iostream.h>
int min_tab(int a[], int nr_elem)
{int elm=a[0];
for (int ind=0; ind<nr_elem; ind++)
      if (elm>=a[ind])  elm=a[ind];
return elm;
}
void citireVector(int b[], int nr_el)
{ for (int ind=0; ind<nr_el; ind++){
      cout<<"Elem "<<ind+1<<"="; cin>>b[ind];}
}
void main()
{
int a[10]; int i,j,n; cout<<"n="; cin>>n;
citireVector(a,n);
int min=min_tab(a,n); cout<<"Elem. min:"<<min<<'\n'; }
```

Aceleeaşi problemă poate fi implementată folosind aritmetica pointerilor:

```
#include <iostream.h>
void citireVector(int *b, int nr_el)
{ for (int ind=0; ind<nr_el; ind++){
      cout<<"Elem "<<ind+1<<"="; cin>>*b(ind+ind);}
}
int min_tab(int *a, int nr_elem)
{int elm=*a;
for (int ind=0; ind<nr_elem; ind++)
      if ( elm>=*(a+ind) )    elm=*(a+ind);
return elm;
}
void main()
{
int a[10]; int i,j,n; cout<<"n="; cin>>n;
citireVector(a, n);
int min=min_tab(a,n);
cout<<"Elem. min:"<<min<<'\n';
}
```

Din exemplele anterioare se poate observa:

1. Prototipul funcţiei min_tab poate fi unul dintre:
```
int min_tab(int a[], int nr_elem);
int min_tab(int *a, int nr_elem);
```
2. Echivalenţe:
```
int *a      ⇔    int a[]
a[i]  ⇔    *(a+i)
```
3. Apelul funcţiilor:
```
citireVector(a,n);
int min=min_tab(a,n);
```
4. Pentru tablourile unidimensionale, la apel, nu trebuie specificat numărul de elemente. Dimensiunea tabloului trebuie să fie cunoscută în funcţia care îl primeşte ca parametru. De obicei, dimensiunea tabloului se transferă ca parametru separat (`nr_elem`).

Exerciţiu: Să se scrie următorul program şi să se urmărească rezultatele execuţiei acestuia.

```
#include <iostream.h>
#include <stdio.h>
double omega(int j, double x, double t[], double *w)
{double s;  cout<<"În funcţia omega:";
cout<<"j="<<j<<" t[j]="<<t[j]<<" t[j+1]="<<t[j+1]<<'\n';
```

```
//j=2 (=i din main)
//t[j]=-3.21 t[j+1]=7.44
cout<<"j="<<j<<" w[j]="<<w[j]<<" w[j+1]="<<w[j+1]<<'\n';
//j=2 (=i din main)
//w[j]=-21.16 w[j+1]=92.2
t[j]=100;   *(t+j+1)=200;     w[j]=300;   *(w+j+1)=400;
cout<<"După atribuiri:\n";
cout<<"j="<<j<<" t[j]="<< t[j]<<" t[j+1]="<<t[j+1]<<'\n';
//După atribuiri:
//j=2
//t[j]=100 t[j+1]=200
//w[j]=300 w[j+1]=400
cout<<"j="<<j<<" w[j]="<<w[j]<<" w[j+1]="<<w[j+1]<<'\n';
int i=2*j+1;      x=x+2.29*i; s=x+2*t[0]-w[1];
cout<<"i="<<i<<" x="<<x<<" s="<<s<<'\n';
//i=5 x=1.123+2.29+5 s=x+2*1.32-(-15.34)
return s;
}
void switch1(double *x, double *y)
{double t=*x; *x=*y; *y=t;}
void switch2(double &x, double &y)
{double t; t=x;x=y;y=t;}
void main()
{double a=123, b=456, u=1.123;
int i=2;
double r[]={1.32, 2.15, -3.21, 7.44, -15.8};
double q[]={12.26, -15.34, -21.16, 92.2, 71.6};
cout<<"i="<<i<<" u="<<u<<'\n';
double y=omega(i,u,r,q);
cout<<"i="<<i<<" u="<<u<<'\n';
//i=2 u=....
cout<<"omega(i,u,r,q)=y="<<y<<'\n';
cout<<"r[i]="<<r[i]<<" r[i+1]="<<r[i+1]<<;
cout<<" q[i]="<<q[i]<<" q[i+1]="<<q[i]<<'\n';
//r[i]=100 r[i+1]=200 q[i]=300 q[i+1]=400
cout<<"a="<<a<<" b="<<b<<'\n';        //a=123 b=456
switch1(&a,&b);
cout<<"Rez. intersch. a="<<a<<" b="<<b<<'\n';   //a=456 b=123
switch2(a,b);
cout<<"Rez. intersch. a="<<a<<" b="<<b<<'\n';   //a=123 b=456
cout<<"r[i]="<<r[i]<<" r[i+1]="<<r[i+1]<<'\n';
switch1(r+i,r+i+1);
cout<<"Rez. intersch. r[i]="<<r[i]<<" r[i+1]="<<r[i+1]<<'\n';
switch2(r[i],r[i+1]);
//switch2(*(r+i),*(r+i+1));
cout<<"Rez. intersch. r[i]="<<r[i]<<" r[i+1]="<<r[i+1]<<'\n';
}
```

În exemplul anterior, parametrii formali `i` şi `x` din funcţia `omega` sunt transmişi prin valoare; parametrii `t` şi `w` sunt parametri tablou, transmişi prin referinţă (referinţă şi pointeri). În funcţia `switch1` parametrii sunt transmişi prin pointeri. În funcţia `switch2` parametrii sunt transmişi prin referinţă.

Pentru ***tablourile multidimensionale***, pentru ca elementele tabloului să poată fi referite în funcţie, compilatorul trebuie informat asupra modului de organizare a tabloului.

Pentru ***tablourile bidimensionale*** (vectori de vectori), poate fi omisă doar precizarea numărului de linii, deoarece pentru a adresa elementul `a[i][j]`, compilatorul utilizează relaţia: **`&mat[i][j]=&mat+(i*N+j)*sizeof(tip)`**, în care `N` reprezintă numărul de coloane, iar *tip* reprezintă tipul tabloului.

Exerciţiu: Fie o matrice de maxim 10 linii şi 10 coloane, ale cărei elemente se introduc de la tastatură. Să se implementeze două funcţii care afişează matricea şi calculează elementul minim din matrice.

```
#include <iostream.h>
int min_tab(int a[][10], int nr_lin, int nr_col)
{int elm=a[0][0];
for (int il=0; il<nr_lin; il++)
      for (int ic=0; ic<nr_col; ic++)
            if (elm>=a[il][ic]) elm=a[il][ic];
return elm;
}
void afisare(int a[][10], int nr_lin, int nr_col)
{
for (int i=0; i<nr_lin; i++)
      {for (int j=0; j<nr_col; j++) cout<<a[i][j]<<'\t';
       cout<<'\n';
       }
}
void main()
{
int mat[10][10];int i, j, M, N;cout<<"Nr. linii:"; cin>>M;
cout<<"Nr. coloane:"; cin>>N;
for (i=0; i<M; i++)
      for (j=0; j<N; j++)
      { cout<<"mat["<<i<<","<<j<<"]="; cin>>mat[i][j];}
afisare(mat, M, N);
int min=min_tab(mat, M, N);
cout<<"Elem. min:"<<min<<'\n';
}
```

Valoarea returnată de o funcţie poate să fie transmisă şi prin referinţă, cum ilustrează exemplul următor:

Exemplu:

```
#include <iostream.h>
#include <stdio.h>
double &func(double &a, double b)
{ printf("În funcţie:\n");
printf("Val a=%f Adr a=%x\n", a, &a);        //Val a=1.20 Adr a=fffe
cout<<"b="<<b<<'\n';                          //b=2.2
a=2*b+1; printf(" După atrib: val a=%f Adr a=%x\n", a, &a);
//Val a=5.40 Adr a=fffe
return a;
}
void main()
{double c=1.2;cout<<"***************MAIN****************\n";
printf("Val c=%f Adr c=%x\n",c, &c);         //Val c=1.20 Adr c=fffe
double d; printf("Adr. d=%x\n", &d);         //Adr. d=ffe6
d=func(c,2.2);
printf("Val d=%f Adr d=%x\n", d, &d);        //Val d=5.40 Adr d=ffe6
}
```

6.5. FUNCŢII CU PARAMETRI IMPLICIŢI

Spre deosebire de limbajul C, în limbajul C++ se pot face iniţializări ale parametrilor formali. Parametrii formali iniţializaţi se numesc ***parametri impliciţi***. De exemplu, antetul funcţiei cmmdc (care calculează şi returnează cel mai mare divizor comun al numerelor întregi primite ca argumente) poate avea aceasta formă:

```
int cmmdc(int x, int y=1);
```

Parametrul formal `y` este iniţializat cu valoarea 1 şi este *parametru implicit*. La apelul funcţiilor cu parametri impliciţi, unui parametru implicit, *poate să-i corespundă sau nu,* un parametru efectiv. Dacă la apel nu îi corespunde un parametru efectiv, atunci parametrul formal va primi valoarea prin care a fost iniţializat (valoarea implicită). Dacă la apel îi corespunde un parametru efectiv, parametrul formal va fi iniţializat cu valoarea acestuia, negijându-se astfel valoarea implicită a parametrului formal. În exemplul anterior, la apelul:

```
int div=cmmdc(9);
```

`x` va lua valoarea 9, iar `y` va lua valoarea 1 (implicită).

Dacă în lista de parametri formali ai unei funcţii există şi parametri impliciţi şi parametri *neiniţializaţi*, parametrii impliciţi trebuie să ocupe ultimele poziţii în listă, nefiind permisă intercalarea acestora printre parametrii neiniţializaţi.

6.6. FUNCŢII CU NUMĂR VARIABIL DE PARAMETRI

În limbajele C şi C++ se pot defini funcţii cu un număr variabil de parametri. Parametrii care trebuie să fie prezenţi la orice apel al funcţiei se numesc ***parametri ficşi***, ceilalţi se numesc ***parametri variabili***. Parametrii ficşi preced parametrii variabili. Prezenţa parametrilor variabili se indică în antetul funcţiei prin *trei puncte* care se scriu după ultimul parametru fix al funcţiei.

De exemplu, fie antetul funcţiei numite `vârf`:

```
void vârf (int n, double a, . . . )
```

Funcţia `vârf` are *doi parametri ficşi* (`n` şi `a`) şi ***parametri variabili***, pentru care nu se precizează în prealabil numărul şi tipul; numărul şi tipul parametrilor variabili diferă de la un apel la altul.

Funcţiile cu un număr variabil de parametri sunt, de obicei, funcţii de bibliotecă (ex: printf, scanf) şi se definesc folosind nişte macrouri speciale care permit accesul la parametrii variabili şi se găsesc în headerul `stdarg.h`.

6.7. FUNCŢII PREDEFINITE

Orice mediu de programare este prevăzut cu una sau mai multe biblioteci de funcţii predefinite. Orice bibliotecă este formată din:

- fişierele header (conţine prototipurile funcţiilor, declaraţiile de variabile);
- biblioteca (arhiva) propriu-zisă (conţine definiţii de funcţii).

Pentru ca funcţiile predefinite să poată fi utilizate, fişierele header în care se găsesc prototipurile acestora trebuie inclus în funcţia (programul) apelant printr-o directivă preprocesor (exemplu `#include <stdio.h>`). Deasemenea, utilizatorul îşi poate crea propriile headere proprii. Pentru a putea utiliza funcţiile proprii, el trebuie să includă aceste headere în programul apelant (exemplu `#include "my_header.h"`).

Pentru funcţiile predefinite, au fost create fişiere header orientate pe anumite numite tipuri de aplicaţii. De exemplu, funcţiile matematice se găsesc în headerul `<math.h>`. Headerul `<stdlib.h>` care conţine funcţii standard. Headerul `<values.h>` defineşte o serie de constante simbolice (exemplu `MAXINT`, `MAXLONG`) care reprezintă, în principal, valorile maxime şi minime ale diferitelor tipuri de date.

6.7.1. Funcţii matematice (headerul `<math.h>`)

Funcţii aritmetice

Valori absolute

`int abs(int x);`

Returnează un întreg care reprezintă valoarea absolută a argumentului.

`long int labs(long int x);`

Analog cu funcția abs, cu deosebirea că argumentul şi valoarea returnată sunt de tip long int.

`double fabs(double x);`

Returnează un real care reprezintă valoarea absolută a argumentului real.

Funcții de rotunjire

`double floor(double x);`

Returnează un real care reprezintă cel mai apropiat număr, fără zecimale, mai mic sau egal cu x (rotunjire prin lipsă).

`double ceil(double x);`

Returnează un real care reprezintă cel mai apropiat număr, fără zecimale, mai mare sau egal cu x (rotunjire prin adaos).

Funcții trigonometrice

`double sin(double x);`

Returnează valoarea lui sin(x), unde x este dat în radiani. Numărul real returnat se află în intervalul [-1, 1].

`double cos(double x);`

Returnează valoarea lui cos(x), unde x este dat în radiani. Numărul real returnat se află în intervalul [-1, 1].

`double tan(double x);`

Returnează valoarea lui tg(x), unde x este dat în radiani.

Funcții trigonometrice inverse

`double asin(double x);`

Returnează valoarea lui arcsin(x), unde x se află în intervalul [-1, 1]. Numărul real returnat (în radiani) se află în intervalul [-pi/2, pi/2].

`double acos(double x);`

Returnează valoarea lui arccos(x), unde x se află în intervalul [-1, 1]. Numărul real returnat se află în intervalul [0, pi].

`double atan(double x);`

Returnează valoarea lui arctg(x), unde x este dat în radiani. Numărul real returnat se află în intervalul [0, pi].

`double atan2(double y, double x);`

Returnează valoarea lui tg(y/x), cu excepția faptului ca semnele argumentelor x şi y permit stabilirea cadranului şi x poate fi zero. Valoarea returnată se află în intervalul [-pi,pi]. Dacă x şi y sunt coordonatele unui punct în plan, funcția returnează valoarea unghiului format de dreapta care uneşte originea axelor carteziene cu punctul, față de axa absciselor. Funcția foloseşte, deasemenea, la transformarea coordonatelor cartezine în coordonate polare.

Funcții exponențiale şi logaritmice

`double exp(double x);`
`long double exp(long double x);`

Returnează valoarea e^x.

`double log(double x);`

Returnează logaritmul natural al argumentului (ln(x)).

`double log10(double x);`

Returnează logaritmul zecimal al argumentului (lg (x)).

`double pow(double baza, double exponent);`

Returnează un real care reprezintă rezultatul ridicării bazei la exponent ($baza^{exponent}$).

`double sqrt(double x);`

Returnează rădăcina pătrată a argumentului $\sqrt{x}$.

`double hypot(double x, double y);`

Funcția distanței euclidiene - returnează $\sqrt{x^2 + y^2}$, deci lungimea ipotenuzei unui triunghi dreptunghic, sau distanța punctului P(x, y) față de origine.

Funcții de generare a numerelor aleatoare

`int rand(void)` `<stdlib.h>`

Generează un număr aleator în intervalul [0, RAND_MAX].

6.7.2. Funcții de clasificare (testare) a caracterelor

Au prototipul în headerul `<ctype.h>`. Toate aceste funcții primesc ca argument un caracter şi returnează un număr întreg care este pozitiv dacă argumentul îndeplineşte o anumită condiție, sau valoarea zero dacă argumentul nu îndeplineşte condiția.

`int isalnum(int c);`

Returnează valoare întreagă pozitivă daca argumentul este literă sau cifră. Echivalentă cu: `isalpha(c)||isdigit(c)`

`int isalpha(int c);`

Testează dacă argumentul este literă mare sau mică. Echivalentă cu `isupper(c)|| islower(c)`.

`int iscntrl(int c);`

Testează dacă argumentul este caracter de control (neimprimabil).

`int isdigit(int c);`

Testează dacă argumentul este cifră.

`int isxdigit(int c);`

Testează dacă argumentul este cifră hexagesimală (0-9, a-f, A-F).

`int islower(int c);`

Testează dacă argumentul este literă mică.

`int isupper(int c);`

Testează dacă argumentul este literă mare.

`int ispunct(int c);`

Testează dacă argumentul este caracter de punctuație (caracter imprimabil, dar nu literă sau spațiu).

`int isspace(int c);`

Testează dacă argumentul este spațiu alb (' ', '\n', '\t', '\v', '\r')

`int isprint(int c);`

Testează dacă argumentul este caracter imprimabil, inclusiv blancul.

6.7.3. Funcții de conversie a caracterelor (prototip în `<ctype.h>`)

`int tolower(int c);`

Funcția schimbă caracterul primit ca argument din literă mare, în literă mică şi returnează codul ASCII al literei mici. Dacă argumentul nu este literă mare, codul returnat este chiar codul argumentului.

`int toupper(int c);`

Funcția schimbă caracterul primit ca argument din literă mică, în literă mare şi returnează codul acesteia. Dacă argumentul nu este literă mică, codul returnat este chiar codul argumentului.

6.7.4. Funcții de conversie din şir în număr (de citire a unui număr dintr-un şir)

(prototip în `<stdlib.h>`)

`long int atol(const char *npr);`

Funcția converteşte şirul transmis ca argument (spre care pointează `npr`) într-un număr cu semn, care este returnat ca o valoare de tipul long int. Şirul poate conține caracterele `'+'` sau `'-'`. Se consideră că numărul este în baza 10 şi funcția nu semnalizează eventualele erori de depăşire care pot apare la conversia din şir în număr.

`int atoi(const char *sir);`

Converteste şirul spre care pointeaza `sir` într-un număr întreg.

`double atof(const char *sir);`

Funcţia converteste şirul transmis ca argument într-un număr real cu semn (returnează valoare de tipul double). În secvenţa de cifre din şir poate apare litera `'e'` sau `'E'` (exponentul), urmată de caracterul `'+'` sau `'-'` şi o altă secvenţă de cifre. Funcţia nu semnalează eventualele erori de depăşire care pot apare.

6.7.5. Funcţii de terminare a unui proces (program)

(prototip în **`<process.h>`**)

`void exit(int status);`

Termină execuţia unui program. Codul returnat de terminarea corectă este memorat în constanta simbolică `EXIT_SUCCES`, iar codul de eroare - în `EXIT_FAILURE`.

`void abort();`

Termină forţat execuţia unui program.

`int system(const char *comanda);` prototip în `<system.h>`

Permite execuţia unei comenzi DOS, specificate prin şirul de caractere transmis ca parametru.

6.7.6. Funcţii de intrare/ieşire (prototip în `<stdio.h>`)

Streamurile (fluxurile de date) implicite sunt: `stdin` (fişierul, dispozitivul standard de intrare), `stdout` (fişierul, dispozitivul standard de ieşire), `stderr` (fişier standard pentru erori), `stdprn` (fişier standard pentru imprimantă) şi `stdaux` (dispozitivul auxiliar standard). De câte ori este executat un program, streamurile implicite sunt deschise automat de către sistem. În headerul `<stdio.h>` sunt definite şi constantele `NULL` (definită ca 0) şi `EOF` (sfârşit de fişier, definită ca -1, CTRL/Z).

`int getchar(void);`

Citeşte un caracter (cu ecou) din fişierul standard de intrare (tastatură).

`int putchar(int c);`

Afişează caracterul primit ca argument în fişierul standard de ieşire (monitor).

`char *gets(char *sir);`

Citeşte un şir de caractere din fişierul standard de intrare (până la primul blank întâlnit sau linie nouă). Returnează pointerul către şirul citit.

`int puts(const char *sir);`

Afişează şirul argument în fişierul standard de ieşire şi adaugă terminatorul de şir. Returnează codul ultimului caracter al şirului (caracterul care precede NULL) sau -1 în caz de eroare.

`int printf(const char *format, ... );`

Funcţia permite scrierea în fişierul standard de ieşire (pe monitor) a datelor, într-un anumit format. Funcţia returnează numărul de octeţi (caractere) afişaţi, sau –1 în cazul unei erori.

1. *Parametrul fix* al funcţiei conţine:
   - Succesiuni de caractere afişate ca atare

     Exemplu:

     `printf("\n Buna ziua!\n\n");` // afişare: *Buna ziua!*
   - *Specificatori de format* care definesc conversiile care vor fi realizate asupra datelor de ieşire, din formatul intern, în cel extren (de afişare).
2. *Parametrii variabili* ai funcţiei sunt expresii. Valorile obţinute în urma evaluării acestora sunt afişate corespunzător specificatorilor de format care apar în parametrul fix. De obicei, parametrul fix conţine atât specificatori de format, cât şi alte caractere. Numărul şi tipul parametrilor variabili trebuie să corespundă specificatorului de format.

Un *specificator de format* care apare în parametrul fix poate avea următoarea formă:

`% [-|c|][sir_cifre_eventual_punct_zecimal] una_sau_doua_litere`

- Implicit, datele se cadrează (aliniază) la dreapta câmpului în care se scriu. Prezenţa caracterului – determină cadrarea la stânga.

Şirul de cifre defineşte dimensiunea câmpului în care se scrie data. Dacă scrierea datei necesită un câmp de lungime mai mare, lungimea indicată în specificator este ignorată. Dacă scrierea datei necesită un câmp de lungime mai mică, data se va scrie în câmp, cadrată la dreapta sau la stânga (dacă apare semnul -), completându-se restul câmpului cu caracterele nesemnificative implicite, adica

spaţii. Şirul de cifre aflate dupa punct definesc precizia (numarul de zecimale cu care este afişat un numar real - implicit sunt afişate 6 zecimale).

Literele definesc tipul conversiei aplicat datei afişate:

c – Afişează un caracter

s – Afişează un şir de caractere

d– Afişează date întregi; cele negative sunt precedate de semnul -.

o – Afişează date de tip int sau unsigned int în octal.

x ***sau*** **X** - Afişează date de tip int sau unsigned int în hexagesimal.

f–Afişează date de tip float sau double în forma: *parte_întreagă.parte_fract*

e ***sau*** **E**-Afişează date de tip float sau double în forma:
`parte_întreagă.parte_fractionară exponent`

Exponentul începe cu e sau E şi defineşte o putere a lui zece care înmulţită cu restul numărului dă valoarea reală a acestuia.

g ***sau*** **G**–Afişează o dată reală fie ca în cazul specificatorului terminat cu f, fie ca în cazul specificatorului terminat cu e. Criteriul de afisare se alege automat, astfel încât afişarea să ocupe un număr minim de poziţii în câmpul de afişare.

l – Precede una din literele d, o, x, X, u. La afişare se fac conversii din tipul *long* sau *unsigned long*.

L – Precede una din literele f, e, E, g, G. La afişare se fac conversii din tipul *long double*.

`int scanf(const char *format, ... );`

Funcţia citeşte din fişierul standard de intrare valorile unor variabile şi le depune în memorie, la adresele specificate. Funcţia returnează numărul câmpurilor citite corect.

1. *Parametrul fix* al funcţiei conţine:
   Specificatorii de format care definesc conversiile aplicate datelor de intrare, din formatul extern, în cel intren (în care sunt memorate). Specificatorii de format sunt asemanători celor folosiţi de funcţia *printf*: **c**, **s**, **d**, **o**, **x** *sau* **X**, **u**, **f**, **l**, **L**.
2. *Parametrii varaibili* reprezintă o listă de adrese ale variabilelor care vor fi citite, deci în această listă, numele unei varaibile simple va fi precedată de operatorul adresă **&**.

`int sprintf(char *sir_cu_format, const char *format, ... );`

Funcţia permite scrierea unor date în şirul transmis ca prim argument, într-un anumit format. Valoarea returnată reprezintă numărul de octeţi (caractere) scrise în şir, sau –1 în cazul unei erori.

`int sscanf(char *sir_cu_format, const char *format, ... );`

Funcţia citeşte valorile unor variabile din şirul transmis ca prim argument şi le depune în memorie, la adresele specificate. Returnează numărul câmpurilor citite corect.

Exemplu: Să se scrie următorul program (care ilustrează modalităţile de folosire a funcţiilor predefinite) şi să se urmărească rezultatele execuţiei acestuia.

```
#include <iostream.h>
#include <stdlib.h>
#include <math.h>
#include <ctype.h>
#include <stdio.h>
void main()
{ int x=-34;      int a=abs(x);      cout<<"a="<<a<<'\n';
long int y=-566666;
cout<<"labs(y)="<<labs(y)<<"fabs(-45.67)="<<fabs(-45.67)<<'\n';
cout<<"fabs(45.67)="<<fabs(45.67)<<'\n';
cout<<floor(78.99)<<'\n';      //78
cout<<floor(78.45)<<'\n';      //78
cout<<floor(-78.45)<<'\n';     //-79
cout<<ceil(78.99)<<'\n';       //79
cout<<ceil(78.45)<<'\n';       //79
cout<<ceil(-78.45)<<'\n';      //-78
```

```
cout<<isalpha('8')<<'\n';      //0
cout<<isalpha('f')<<'\n';      //val diferita de zero
cout<<isalpha('%')<<'\n';      //0
cout<<tolower('D')<<'\n';      //100 (codul caracterului 'd')
cout<<toupper('a')<<'\n';      //65 (codul caracterului 'A')
char s1[]="-56.234 h mk";      cout<<atol(s1)<<'\n'; //-56
cout<<atoi(s1)<<'\n';          //-56
cout<<atof(s1)<<'\n';          //-56.234
cout<<atof("45E+3  n")<<'\n'; //45000
cout<<"EXECUTIA COMENZII DOS DIR\n";      int cod_ret=system("dir");
cout<<"Val. cod retur="<<cod_ret<<'\n';
int c;cout<<"Astept car:"; c=getchar();    //Presupunem caracter introdus: e
cout<<"Caracterul citit este:"<<putchar(c);//Caracterul citit este: 101
// 101=codul carcterului e
cout<<'\n';puts(s1);cout<<'\n';     printf("Afisarea unui mesaj\n");
int intreg=-45;
printf("VALOAREA VARIABILEI INTREG ESTE:%d\n", intreg);
printf("VALOAREA VARIABILEI INTREG ESTE:%10d\n", intreg);
printf("VALOAREA VARIABILEI INTREG ESTE:%-10d\n", intreg);
double real=2.45;
printf("VALOAREA VARIABILEI real ESTE:%f\n", real);
printf("VALOAREA VARIABILEI real ESTE:%10.3f\n", real);
printf("VALOAREA VARIABILEI real ESTE:%10.5f\n", real);
printf("VALOAREA VARIABILEI real ESTE:%e\n", real);
printf("VAL VAR real:%f si\neste mem. la adr.%x\n",real,&real );
printf("astept sir:");scanf("%s",s1);
printf("Sirul citit este: %s \n", s1);
char sir_f[100];
sprintf(sir_f,"Codul caracterului %c este:%d",c, (int)c);
puts(sir_f);
}
```

6.8. CLASE DE MEMORARE

Definiții:

Variabilele declarate în afara oricărei funcții sunt ***variabilele globale***.
Variabilele declarate în interiorul unui bloc sunt ***variabilele locale***.
Porțiunea de cod în care o variabilă este accesibilă reprezintă ***scopul (domeniul de vizibilitate) al variabilei*** respective.

Parametrii formali ai unei funcții sunt *variabile locale* ale funcției respective.
Domeniul de vizibilitate al unei variabile locale este blocul în care variabila respectivă este definită.
În situația în care numele unei variabile globale coincide cu numele unei variabile locale, *variabila locală o "maschează" pe cea globală*, ca în exemplul următor: în interiorul blocului din funcția main s-a redefinit variabila `a`, care este *variabilă locală* în interiorul blocului. Variabila locală `a` maschează variablila globală numită tot `a`.

Exemplu:

```
#include <stdio.h>
void main()
{ int a,b; a=1; b=2;
printf("În afara blocului a=%d b=%d\n", a, b);
      {int a=5;         b=6;
      printf("În interiorul blocului a=%d b=%d\n",a,b);
      }
printf("În afara blocului a=%d b=%d\n", a, b);
}
```

În cazul variabilelor locale, compilatorul alocă memorie în momentul execuției blocului sau funcției în care acestea sunt definite. Când execuția funcției sau blocului se termină, se eliberează memoria pentru acestea şi valorile pentru variabilele locale se pierd.

Definiții:

Timpul de viață a unei variabile locale este durata de execuție a blocului (sau a funcției) în care aceasta este definită.

Timpul de viață a unei variabile globale este durata de execuție a programului.

În exemplul următor, variabila întreagă `x` este vizibilă atât în funcția `main`, cât şi în funcția `func1` (`x` este variabila globală, fiind definită în exteriorul oricărei funcții). Variabilele `a` şi `b` sunt variabile locale în funcția `main` (vizibile doar în main). Variabilele `c` şi `d` sunt variabile locale în funcția `func1` (vizibile doar în `func1`). Varabila `y` este variabilă externă şi este vizibilă din punctul în care a fost definită, până la sfârşitul fişierului sursă (în acest caz, în funcția `func1`).

Exemplu:

```
int x;
void main()
{int a,b;
//- - - - - - - - -
}
int y;
void func1(void)
{int c,d;
//- - - - - - - -
}
```

Clase de memorare

O variabilă se caracterizează prin: nume, tip, valoare şi clasă de memorare.

Clasa de memorare se specifică la declararea variabilei, prin unul din următoarele cuvinte cheie:

- **`auto;`**
- **`register;`**
- **`extern;`**
- **`static.`**

Clasa de memorare determină timpul de viață şi domeniul de vizibilitate (scopul) unei variabile (tabelul 6.1).

Exemplu:

```
auto int a;
static int x;
extern double y;
register char c;
```

- **Clasa de memorare auto**

  Dacă o *variabilă locală* este declarată fără a se indica în mod explicit o clasă de memorare, clasa de memorare considerată implicit este `auto`. Pentru acea variabilă se alocă memorie automat, la intrarea în blocul sau în funcția în care ea este declarată. Deci domeniul de vizibilitate al variabilei este blocul sau funcția în care aceasta a fost definită. Timpul de viață este durata de execuție a blocului sau a funcției.

- **Clasa de memorare register**

  Variabilele din clasa register au acelaşi domeniu de vizibilitate şi timp de viață ca şi cele din clasa auto. Deosebirea față de variabilele din clasa auto constă în faptul că pentru memorarea variabilelor register, compilatorul utilizează regiştrii interni (ceea ce conduce la creşterea eficienței). Unei variabile pentru care se specifică drept clasă de memorare register, *nu i se poate aplica operatorul de referențiere*.

- **Clasa de memorare extern**

  O *variabilă globală* declarată fără specificarea unei clase de memorare, este considerată ca având clasa de memorare `extern`. Domeniul de vizibilitate este din momentul declarării până la sfârşitul fişierului sursă. Timpul de viaţă este durata execuţiei fişierului. O variabilă din clasa extern este iniţializată automat cu valoarea 0.

- **Clasa de memorare static**

  Clasa de memorare static are două utilizări distincte:
  - Variabilele *locale statice* au ca domeniu de vizibilitate blocul sau funcţia în care sunt definite, iar ca timp de viaţă - durata de execuţie a programului. Se iniţializează automat cu 0.
  - Variabilele *globale statice* au ca domeniu de vizibilitate punctul în care au fost definite până la sfârşitul fişierului sursă, iar ca timp de viaţă - durata execuţiei programului.

Tabelul 6.1.

Clasa de memorare	Variabila	Domeniu vizibilitate	Timp de viaţă
`auto` (`register`)	locală (internă)	Blocul sau funcţia	Durara de execuţie a blocului sau a funcţiei
`extern`	globală	❑ Din punctul definirii, până la sfârşitul fişierului (ROF) ❑ Alte fişiere	Durara de execuţie a blocului sau a programului
`static`	globală	ROF	-"-
	locală	Bloc sau funcţie	-"-
nespecificat	globală	Vezi extern	Vezi extern
	locală	Vezi auto	Vezi auto

6.9. MODURI DE ALOCARE A MEMORIEI

Alocarea memoriei se poate realiza în următoarele moduri:
- ***alocare statică;***
- ***alocare dinamică;***
- ***alocare pe stivă.***
- *Se alocă static memorie* în următoarele cazuri:
  - pentru instrucţiunile de control propriu-zise;
  - pentru variabilele globale şi variabilele locale declarate în mod explicit static.
- *Se alocă memorie pe stivă* pentru variabilele locale.
- *Se aloca dinamic memorie* în mod explicit, cu ajutorul funcţiilor de alocare dinamica, aflate în headerul **`<alloc.h>`**.

Exemplu:
```
int a,b; double x;
double f1(int c, double v)
{int b;
static double z;
}
double w;
int f1(int w)
{double a;
}
void main()
{double b, c; int k;
b=f1(k,c);
}
```

6.9.1. Alocarea memoriei în mod dinamic

Pentru toate tipurile de date (simple sau structurate), la declararea acestora, compilatorul alocă automat un număr de locaţii de memorie (corespunzător tipului datei). Dimensiunea zonei de memorie necesară pentru păstrarea valorilor datelor este fixată înaintea lansării în execuţie a programului. În cazul declarării unui tablou de întregi cu maximum 100 de elemente vor fi alocaţi `100*sizeof(int)` locaţii de memorie succesive. În situaţia în care la un moment dat tabloul are doar 20 de elemente, pentru a aloca doar atâta memorie cât este necesară în momentul respectiv, se va aloca memorie în mod dinamic.

Este de dorit ca în cazul datelor a căror dimensiune nu este cunoscută a priori sau variază în limite largi, să se utilizeze o altă abordare: alocarea memoriei în mod dinamic. În mod dinamic, memoria nu mai este alocată în momentul compilării, ci ***în momentul execuţiei***. Alocarea dinamică elimină necesitatea definirii complete a tuturor cerinţelor de memorie în momentul compilării. În ***limbajul C***, alocarea memoriei în mod dinamic se face cu ajutorul funcţiilor ***malloc, calloc, realloc***; eliberarea zonei de memorie se face cu ajutorul funcţiei ***free***. Funcţiile de alocare/dezalocare a memoriei au prototipurile în header-ele **<stdlib.h>** şi **<alloc.h>**:

void *malloc(size_t nr_octei_de_alocat);
Funcţia `malloc` necesită un singur argument (numărul de octeţi care vor fi alocaţi) şi returnează un pointer generic către zona de memorie alocată (pointerul conţine adresa primului octet al zonei de memorie rezervate).
void *calloc(size_t nr_elemente, size_t mărimea_în_octeţi_ a_unui_elem);
Funcţia `calloc` lucrează în mod similar cu malloc; alocă memorie pentru un tablou de nr_elemente, numărul de octeţi pe care este memorat un element este mărimea_în_octeţi_a_unui_elem şi returnează un pointer către zona de memorie alocată.
void *realloc(void *ptr, size_t mărime);
Funcţia `realloc` permite modificarea zonei de memorie alocată dinamic cu ajutorul funcţiilor malloc sau calloc.
<u>Observaţie:</u>
În cazul în care nu se reuşeşte alocarea dinamică a memoriei (memorie insuficientă), funcţiile malloc, calloc şi realloc returnează un pointer null. Deoarece funcţiile malloc, calloc, realloc returnează un pointer generic, rezultatul poate fi atribuit oricărui tip de pointer. La atribuire, este indicat să se utilizeze operatorul de conversie explicită (vezi exemplu).

Eliberarea memoriei (alocate dinamic cu una dintre funcţiile malloc, calloc sau realloc) se realizează cu ajutorul funcţiei **free**.
void free(void *ptr);

<u>Exemplu:</u> Să se aloce dinamic memorie pentru 20 de valori întregi.
```
int *p;
p=(int*)malloc(20*sizeof(int));
//p=(int*)calloc(20, sizeof(int));
```

<u>Exerciţiu:</u> Să se scrie un program care implementează funcţia numită `introd_val`. Funcţia trebuie să permită introducerea unui număr de valori reale, pentru care se alocă memorie dinamic. Valorile citite cu ajutorul funcţiei `introd_val` sunt prelucrate în funcţia `main`, apoi memoria este eliberată.
```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
float *introd_val()
/* pentru a putea realiza eliberarea memoriei în funcţia main, funcţia introd_val trebuie să returneze adresa de început a zonei de memorie alocate dinamic */
{double *p; int nr;printf("Număr valori:"); scanf("%d", nr);
if (!(p=(float*)malloc(nr*sizeof(float))) ){
```

```
        printf("Memorie insuficientă!\n");return NULL;
}
for (int i=0; i<nr; i++){
        printf("Val %d=", i+1); scanf("%lf", p+i); return p;}
}
void main()
{float *pt; pt=introd_val();
// prelucrare tablou
free(pt);
}
```

Exercițiu: Să se scrie un program care citeşte numele angajaților unei întreprinderi. Numărul angajaților este transmis ca argument către funcția main. Alocarea memoriei pentru cei nr_ang angajați, cât şi pentru numele fiecăruia dintre aceştia se va face în mod dinamic.

```
#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>
void main(int argc, char *argv[])
{char **ang_ptr;
char *nume;
int nr_ang, i;
if (argc==2){
        nr_ang=atoi(argv[1]);/* numărul angajaților este transmis ca argument către funcția
main. El este convertit din şir de caractere în număr */
        ang_ptr=(char**)calloc(nr_ang, sizeof(char*));
        if ((ang_ptr==0){
                printf("Memorie insuficientă!\n");exit(1);}
        nume=(char*)calloc(30, sizeof(char));
        for (i=0; i<nr_ang; ++i){
        printf("Nume angajat:");
        scanf("%s",nume);
        ang_ptr[i]=(char*)calloc(strlen(nume)+1, sizeof(char));
        strcpy(ang_ptr[i], nume);
        }
        free(nume);
        printf("\n");
        for (i=0; i<nr_ang; i++)
                printf("Angajat nr %d: %s\n", i+1, ang_ptr[i]);
}
else
        printf("Lansare în execuție: %s număr_de_angajați\n", argv[0]);
}
```

În limbajul C++ alocarea dinamică a memoriei şi eliberarea ei se pot realiza cu operatorii **new** şi **delete**. Folosirea acestor operatori reprezintă o metodă superioară, *adaptată programării orientate obiect*.
Operatorul **new** este un operator unar care returnează un pointer la zona de memorie alocată dinamic. În situația în care nu există suficientă memorie şi alocarea nu reuşeşte, operatorul new returnează pointerul NULL. Operatorul **delete** eliberează zona de memorie spre care pointează argumentul său.

Sintaxa:

```
tipdata_pointer = new tipdata;
tipdata_pointer = new tipdata(val_inițializare);
//pentru inițializarea datei pentru care se alocă memorie dinamic
tipdata_pointer = new tipdata[nr_elem]; //alocarea memoriei pentru un tablou

delete tipdata_pointer;
delete [nr_elem] tipdata_pointer;         //eliberarea memoriei pentru
tablouri
```

`Tipdata` reprezintă tipul datei (predefinit sau obiect) pentru care se alocă dinamic memorie, iar `tipdata_pointer` este o variabilă pointer către tipul tipdata.

Pentru a putea afla memoria RAM disponibilă la un moment dat, se poate utiliza funcția `coreleft`:

```
unsigned coreleft(void);
```

Exercițiu: Să se aloce dinamic memorie pentru o dată de tip întreg:

```
int *pint;
pint=new int;
//Sau:
int &i=*new int;
i=100;          //i permite referirea la întregul păstrat în zona de memorie alocată dinamic
```

Exercițiu: Să se aloce dinamic memorie pentru o dată reală, dublă precizie, inițializând-o cu valoarea -7.2.

```
double *p;
p=new double(-7.2);
//Sau:
double &p=* new double(-7.2);
```

Exercițiu: Să se aloce dinamic memorie pentru un vector de m elemente reale.

```
double *vector; vector=new double[m];
```

Exemplu: Să se urmărească rezultatele execuției următorului program, care utilizează funcția coreleft.

```
#include <iostream.h>
#include <alloc.h>
#include <conio.h>
void main()
{ int *a,*b; clrscr();
cout<<"Mem. libera inainte de alocare:"<<coreleft()<<'\n';
cout<<"Adr. pointerilor a si b:"<<&a<<"    "<<&b<<'\n';
cout<<"Valorile pointeri a si b inainte de alocare:"<<a<<"    "<<b<<'\n';
a=new int;  b=new int[10];
cout<<"Mem. libera dupa alocare:"<<coreleft()<<'\n';
cout<<"Valorile pointerilor a si b dupa alocare:"<<a<<"    "<<b<<'\n';
cout<<"Continutul memoriei alocate:\n"<<"*a="<<*a<<"\n*b="<<*b<<'\n';
for (int k=0;k<10;k++) cout<<"\nb["<<k<<"]="<<b[k]; cout<<'\n';
getch();
*a=1732;
for (int u=0;u<10;u++) b[u]=2*u+1;
cout<<"Cont. zonelor alocate dupa atribuire:"<<"\n*a="<<*a<<"\nb=";
for (u=0;u<10;u++) cout<<"\nb["<<u<<"]="<<b[u];
delete a; delete b;
cout<<"Mem. libera dupa eliberare:"<<coreleft()<<'\n';
cout<<"Valorile pointerilor a si b dupa eliberare:"<<a<<"    "<<b<<'\n';
cout<<"Continutul memoriei eliberate:\n"<<"*a="<<*a<<"\n*b="<<*b<<'\n';
for (k=0;k<10;k++) cout<<"\nb["<<k<<"]="<<b[k]; cout<<'\n'; cout<<b[3];
getch();
}
```

6.10. FUNCȚII RECURSIVE

O funcție este numită ***funcție recursivă*** dacă ea se autoapelează, fie *direct (în definiția ei se face apel la ea însăși)*, fie *indirect* (prin apelul altor funcții). Limbajele C/C++ dispun de mecanisme speciale care permit suspendarea execuției unei funcții, salvarea datelor și reactivarea execuției la momentul potrivit. Pentru fiecare apel al funcției, parametrii și variabilele automatice se memorează pe stivă, având valori distincte. Variabilele statice ocupă tot timpul aceeași zonă de memorie (figurează într-un singur exemplar) și își păstrează valoarea de la un apel la altul. Orice apel al unei funcții conduce la o revenire în funcția respectivă, în punctul următor instrucțiunii de apel. La revenirea dintr-o funcție, stiva este curățată (stiva revine la starea dinaintea apelului).

Un exemplu de funcție recursivă este funcția de calcul a factorialului, definită astfel:
`fact(n)=1`, dacă `n=0`;
`fact(n)=n*fact(n-1)`, dacă `n>0`;

Exemplu: Să se implementeze recursiv funcția care calculează `n!`, unde `n` este introdus de la tastatură:

```
#include <iostream.h>
int fact(int n)
{if (n<0){
      cout<<"Argument negativ!\n";
      exit(2);
 }
else if (n==0)    return 1;
else        return n*fact(n-1);
}
void main()
{int nr, f; cout<<"nr="; cin>>nr;
f=fact(nr); cout<<nr<<"!="<<f<<'\n';
}
```

Se observă că în corpul funcției `fact` se apelează însăşi funcția `fact`. Presupunem că `nr=4` (inițial, funcția fact este apelată pentru a calcula 4!). Să urmărim diagramele din figurile 6.7. şi 6.8. La apelul funcției `fact`, valoarea parametrului de apel `nr` (nr=4) inițializează parametrul formal `n`. Pe *stivă* se memorează adresa de revenire în funcția apelantă (adr1) şi valoarea lui n (n=4) (figura 6.7.a.). Deoarece n>0, se execută intrucțiunea de pe ramura else (return n*fact(n-1)). Funcția fact se autoapelează direct. Se memorează pe stivă noua adresă de revenire şi noua valoare a parametrului n (n=3) (figura 6.7.b.).
La noul reapel al funcției fact, se execută din nou intrucțiunea de pe ramura else (return n*fact(n-1)). Se memorează pe stivă adresa de revenire şi noua valoare a parametrului n (n=2) (figura 6.7.c.). La noul reapel al funcției fact, se execută din nou intrucțiunea de pe ramura else (return n*fact(n-1)). Se memorează pe stivă adresa de revenire şi noua valoare a parametrului n (n=1) (figura 6.7.d.). La noul reapel al funcției fact, se execută din nou intrucțiunea de pe ramura else (return n*fact(n-1)). Se memorează pe stivă adresa de revenire şi noua valoare a parametrului n (n=0) (figura 6.7.e.).
În acest moment n=0 şi se revine din funcție cu valoarea 1 (1*fact(0)=1*1), la configurația stivei din figura 6.7.d.) (se curăță stiva şi se ajunge la configurația din figura 6.7.d.). În acest moment n=1 şi se revine cu valoarea 2*fact(1)=2*1=2, se curață stiva şi se ajunge la configurația stivei din figura 6.7.c. În acest moment n=2 şi se revine cu valoarea 3*fact(2)=3*2=6, se curață stiva şi se ajunge la configurația stivei din figura 6.7.b. Se curăță stiva şi se ajunge la configurația stivei din figura 6.7.a.. În acest moment n=3 şi se revine cu valoarea 4*fact(3)=4*6=24.
O funcție recursivă poate fi realizată şi ***iterativ***. Modul de implementare trebuie ales în funcție de problemă. Deşi implementarea recursivă a unui algoritm permite o descriere clară şi compactă, recursivitatea nu conduce la economie de memorie şi nici la execuția mai rapidă a programelor. În general, se recomandă utilizarea funcțiilor recursive în anumite tehnici de programare, cum ar fi unele metode de căutare (`backtracking`).

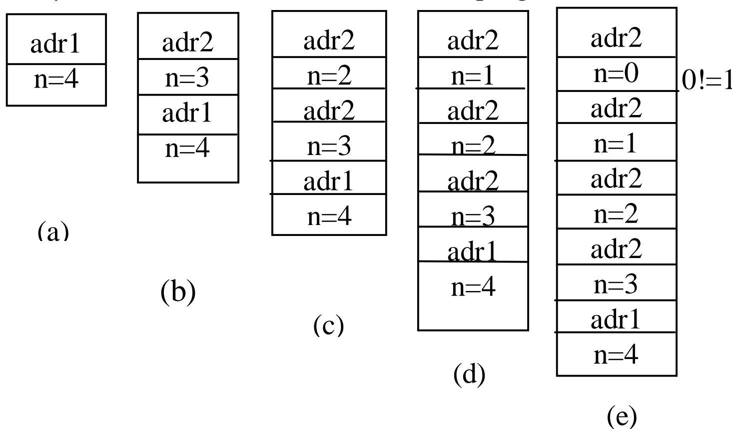


Figura 6.7. Configurația stivei

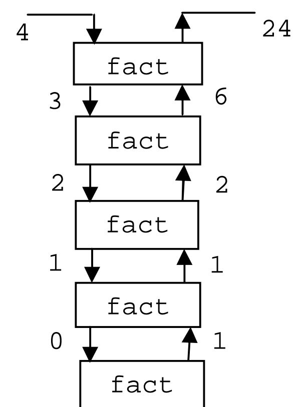


Figura 6.8. Parametri funcției `fact`

Exerciţiu: Fie şirul lui Fibonacci, definit astfel: `f(0)=0`, `f(1)=1`, `f(n)=f(n-1)+f(n-2)`, dacă n>1. Să se scrie un program care implementează algoritmul de calcul al şirului Fibonacci atât recursiv, cât şi iterativ. Să se compare timpul de execuţie în cele două situaţii.

```
#include <iostream.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>
#include <stdio.h>

long int iterativ_fib(long int n)     //varianta de implementare iterativă
{if (n==0) return 0;
if (n==1) return 1;
int i; long int a, b, c; a=0; b=1;
for (i=2; i<=n; i++){
      c=b; b+=a; a=c;}
return b;
}

long int recursiv_fib(long int n)     //varianta de implementare recursivă
{if (n==0) return 0;
if (n==1) return 1;
long int i1=recursiv_fib(n-1);
long int i2=recursiv_fib(n-2);
return i1+i2;
}

void main()
{int n; clrscr();
cout<<MAXLONG<<'\n';
for (n=10; n<=40; n++)  {
      clock_t t1, t2, t3;
      cout<<CLK_TCK<<'\n';
      t1=clock(); long int f1=iterativ_fib(n);
      t2=clock(); long f2=recursiv_fib(n); t3=clock();
      double timp1=(double)(t2-t1)/CLK_TCK;
      double timp2=(double)(t3-t2)/CLK_TCK;
      printf("ITERATIV: %10ld t=%20.10lf\n",f1,timp1);
      printf("RECURSIV: %10ld t=%20.10lf\n",f2,timp2);
      cout<<"Apasa o tasta....\n"; getch();
      } }
```

În exemplul anterior, pentru măsurarea timpului de execuţie s-a utilizat funcţia `clock`, al cărei prototip se află în header-ul `time.h`. Variabilele `t1`, `t2` şi `t3` sunt de tipul `clock_t`, tip definit în acelaşi header. Constanta simbolică `CLK_TCK` defineşte numărul de bătăi ale ceasului, pe secundă.

În general, orice algoritm care poate fi implementat iterativ, poate fi implementat şi recursiv. Timpul de execuţie a unei recursii este semnificativ mai mare decât cel necesar execuţiei iteraţiei echivalente.

Exerciţiu: Să se implementeze şi să se testeze un program care:

a) Generează aleator şi afişează elementele unui vector ;
b) Sortează aceste elemente, crescător, aplicând metodele de sortare `BubbleSort`, `InsertSort`, şi `QuickSort`;
c) Să se compare viteza de sortare pentru vectori de diverse dimensiuni (10,30,50,100 elemete).

- Metoda **`BubbleSort`** a fost prezentată în capitolul 4.

- Metoda **`QuickSort`** reprezintă o altă metodă de sortare a elementelor unui vector**.** Algoritmul este recursiv: se împarte vectorul în două partiţii, faţă de un element pivot (de obicei, elementul din "mijlocul vectorului"). Partiţia stângă începe de la indexul i (la primul apel `i=0`), iar partiţia dreaptă se termină cu indexul j (la primul apel `j=n -1`) (figura 6.9.).

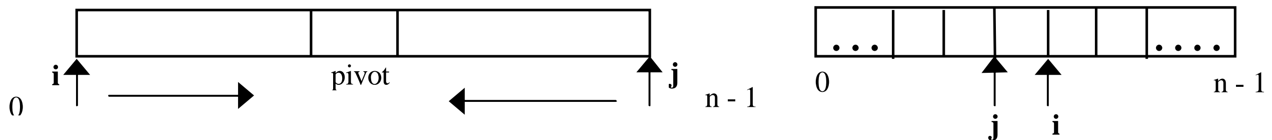


Figura 6.9. Sortare prin metoda QuickSort

Partiția stângă este extinsă la dreapta (i incrementat) până când se găseşte un element mai mare decât pivotul; partiția dreaptă este extinsă la stânga (j decrementat) până când se găseşte un element mai mic decât pivotul. Cele două elemente găsite, `vect[i]` şi `vect[j]`, sunt interschimbate.
Se reia ciclic extinderea partițiilor până când `i` şi `j` se "încrucişează" (`i` devine mai mare ca `j`). În final, partiția stângă va conține elementele mai mici decât pivotul, iar partiția dreaptă - elementele mai mari decât pivotul, dar nesortate.
Algoritmul este reluat prin recursie pentru partiția stângă (cu limitele între 0 şi j), apoi pentru partiția dreaptă (cu limitele între i şi n-1). Recursia pentru partea stângă se opreşte atunci când j atinge limita stângă (devine 0), iar recursia pentru partiția dreaptă se opreşte când i atinge limita dreaptă (devine n-1).

```
SUBALGORITM QuickSort (vect[ ], stg, drt)    //la primul apel stg = 0 si drt = n - 1
ÎNCEPUT SUBALGORITM
      i←stg
      j←drt
      DACĂ  i < j ATUNCI
      ÎNCEPUT
            pivot=vect[(stg+drt)/2]
            CÂT TIMP i <= j REPETĂ
            //extinderea partițiilor stânga şi dreapta până când i se încrucişează cu j
            ÎNCEPUT
                  CÂT TIMP i<drt si vect[i]<pivot REPETĂ
                        i = i + 1
                  CÂT TIMP j<stg si vect[j]>pivot REPETĂ
                        j = j - 1
                  DACĂ i<=j ATUNCI
                  ÎNCEPUT       //interschimbă elementele vect[i] şi vect[j]
                        aux←vect[i]
                        vect[i]←vect[j]
                        vect[j]←aux
                        i←i+1
                        j←j-1
                  SFÂRŞIT
            SFÂRŞIT
            DACĂ j  > stg  ATUNCI // partiția stângă s-a extins la maxim, apel qiuckSort pentru ea
                  CHEAMĂ QuickSort(vect, stg, j)
            DACĂ i  < drt  ATUNCI // partiția dreaptă s-a extins la maxim, apel qiuckSort pentru ea
                  CHEAMĂ QuickSort(vect, i, drt)
      SFÂRŞIT
SFÂRŞIT SUBALGORITM
```

- Metoda **`InsertSort`** (metoda inserției)

Metoda identifică cel mai mic element al vectorului şi îl schimbă cu primul element. Se reia procedura pentru vectorul inițial, fără primul element şi se caută minimul în acest nou vector, etc.

```
SUBALGORITM InsertSort (vect[ ], nr_elem)
ÎNCEPUT SUBALGORITM
   CÂT TIMP i< nr_elem REPETĂ
      ÎNCEPUT
```

```
            pozMin←cautMinim(vect, i)      // se apelează algoritmul cautMinim
            aux←vect[i]
            vect[i]←vect[pozMin]
            vect[pozMin]←aux
            i←i+1
      SFÂRŞIT
SFÂRŞIT SUBALGORITM
```

Funcția **`cautMin(vect[ ], indexIni, nr_elem)`** caută elementul minim al unui vector, începând de la poziția `indexIni` şi returnează poziția minimului găsit.

Mod de implementare (Se va completa programul cu instrucțiunile care obțin şi afişează timpului necesar ordonării prin fiecare metodă. Se vor compara rezultatele pentru un vector de 10, 30, 50, 100 elemente):

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define TRUE      1
#define FALSE     0
void gener(double v[], int n)
//functia de generare aleatoare a elementelor vectorului v, cu n elemente
{for (int i=0; i<n; i++)
      v[i]=1.0*rand()/100000;
}
void afis(double v[], int n)
//functia de afisare a vectorului
{for (int i=0; i<n; i++)
      printf("%10.2f",v[i]);
printf("\n");
}
void copie_vect(double v1[], double v[], int n)
//functie de "duplicare "a unui vector; copie vectorul v in vectorul v1
{for (int i=0; i<n; i++)
      v1[i]=v[i];
}
void bubbleSort(double v[], int n)
{int gata;  gata=FALSE;
while (!gata){
      gata=TRUE;
      for (int i=0; i<n-1; i++)
            if (v[i]>=v[i+1]){
                  double aux=v[i];
                  v[i]=v[i+1];
                  v[i+1]=aux;
//                printf("Interschimbare element %d cu %d",i,i+1);
//                afis(v,n);
                  gata=FALSE;}
}
}
int cautMin(double v[], int indexIni, int n)
// cauta elementul minim, incepând de la pozitia indexIni, inclusiv
{ double min=v[indexIni];
int pozMin=indexIni;
for (int i=indexIni; i<n; i++)
      if (v[i]<=min){
            min=v[i]; pozMin=i;
      }
return pozMin;
}
void insertSort(double v[], int n)
{ int i;
for (i=0; i<n; i++){
      int poz=cautMin(v, i, n);
      double aux=v[i];
```

```
      v[i]=v[poz];
      v[poz]=aux;
}
}
void quickSort(double v[], int stg, int drt)
{int i,j; i=stg; j=drt; double pivot, aux;
 if (i<j){
      pivot=v[(stg+drt)/2];
      while (i<=j){       //extindere partitie st si dr pana i se incrucis cu j
            while (i<drt && v[i]<pivot)   i++;
            while (j>stg && v[j]>pivot)   j--;
            if (i<=j){
                  aux=v[i];v[i]=v[j];v[j]=aux; //interschimbare elemente
                  i++; j--;
            }
      }
      if (j>stg)  quickSort(v, stg, j);
      if (i<drt)  quickSort(v, i, drt);
 }
}
void main()
{
clock_t ti,tf; int n;    //n = nr elemente vector
printf("Nr componente vector:"); scanf("%d", &n);
double v[200], v1[200], v2[200], v3[200];
gener(v, n);
copie_vect(v1,v,n);
printf("\nInainte de ordonare: v1="); afis(v1, n); ti=clock();
bubbleSort(v1,n); tf=clock(); printf("\nDupa ordonare  : v1=");afis(v1, n);
printf("%10.7f", dif_b);
printf("\n\n****** INSERT SORT ******\n");
copie_vect(v2,v,n);
printf("\nInainte de ordonare INSERT: v2="); afis(v2, n);
insertSort(v2,n); printf("\nDupa ordonare       INSERT: v2=");afis(v2, n);
int st=0; int dr=n-1; copie_vect(v3, v, n);
printf("\n\n****** QUICK SORT ******\n");
printf("\nInainte ordonare QUICK: v3="); afis(v3, n);
quickSort(v3, st, dr); printf("\nDupa ordonare    QUICK: v3="); afis(v3, n);
}
```

6.11. POINTERI CĂTRE FUNCŢII

Aşa cum s-a evidenţiat în capitolul 5, exista trei categorii de variabilele pointer:

- Pointeri cu tip;
- Pointeri generici (void);
- Pointeri ***către funcţii***.

Pointerii către funcţii sunt variabile pointer care conţin adresa de început a codului executabil al unei funcţii. Pointerii către funcţii permit:

- Transferul ca parametru al adresei unei funcţii;
- Apelul funcţiei cu ajutorul pointerului.

Declaraţia unui pointer către funcţie are următoarea formă:

`tip_val_intoarse (*nume_point)(lista_declar_param_formali);` ,

unde:

`nume_point` este un pointer de tipul "funcţie cu rezultatul tipul_valorii_întoarse". În declaraţia anterioară trebuie remarcat rolul parantezelor, pentru a putea face distincţie între declaraţia unei funcţii care întoarce un pointer şi declaraţia unui pointer de funcţie:

`tip_val_intoarse * nume_point (lista_declar_param_formali);`

```
tip_val_intoarse (* nume_point)(lista_declar_param_formali);
```

Exemplu:

```
int f(double u, int v);          //prototipul funcției f
int (*pf)(double, int);          //pointer către funcția f
int i, j; double d;
pf=f;                   //atribuie adresa codului executabil al funcției f pointerului pf
j=*pf(d, i);            //apelul funcției f, folosind pf
```

Exercițiu: Să se implementeze un program care calculează o funcție a cărei valoare este integrala altei funcții. Pentru calculul integralei se va folosi metoda trapezelor.

Relația pentru calculul integralei prin metoda trapezelor pentru $\int_a^b f(x)dx$ este:

$$I = (f(a)+f(b))/2 + \sum_{k=1}^{n-1} f(a+k*h)$$

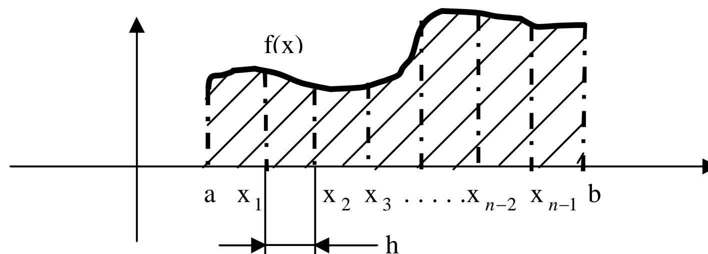


Figura 6.9. Calculul integralei prin metoda trapezelor

Să se calculeze $\int_a^b \frac{\sqrt{0.2+e^{\frac{|x|}{2}}}}{1+\sqrt{0.3+\ln(1+x^4)}} dx$,

cu o eroare mai mică decât eps (valoarea erorii introdusă de la tastatură).

```
#include <conio.h>
#include <math.h>
#include <iostream.h>
double functie(double x)
    {return sqrt(0.1+exp(0.5*fabs(x)))/(1+sqrt(0.3+log(1+pow(x,4))));}

double intrap(double a, double b, long int n, double (*f)(double))
{double h,s=0; long k;
if (a>=b)   return 0;
if (n<=0)   n=1;
h=(b-a)/n;
for (k=1; k<n; k++)     s+=f(a+k*h);
return ((f(a)+f(b))/2+s)*h;
}

void main()
{long int j; double p,q; double eps, d2;double dif;
cout<<"Marg. inf:";cin>>p; cout<<"Marg. sup:";cin>>q;
cout<<"Eroare:";cin>>eps; j=1;
double d1=intrap(p, q, j, functie);
do{
      j*=2;
if (j>MAXLONG || j<0)   break;
      d2=intrap(p, q, j, functie);
      dif=fabs(d1-d2); d1=d2;
      cout<<"Nr.intervale "<<j<<" Val.integralei "<<d2<<'\n';
}while (dif>eps);
cout<<"\n\n-----------------------------------------------\n";
cout<<"Val. integralei: "<<d2<<" cu eroare de:"<<eps<<'\n';
}
```

ÎNTREBĂRI ȘI EXERCIȚII

Chestiuni teoretice

1. Asemănări între transferul parametrilor unei funcţii prin pointeri şi prin referinţă.
2. Caracteristicile modului de transfer a parametrilor unei funcţii prin pointeri.
3. Caracteristicile variabilelor globale.
4. Caracteristicile variabilelor locale.
5. Care este diferenţa între antetul unei funcţii şi prototipul acesteia?
6. Care sunt modurile de alocare a memoriei?
7. Care sunt modurile de transfer a parametrilor unei funcţii?
8. Care sunt operatorii din C++ care permit alocarea/dezalocarea dinamică a memoriei?
9. Ce clase de memorare cunoasteţi?
10. Ce este domeniul de vizibilitate a unei variabile?
11. Ce este prototipul unei funcţii?
12. Ce este timpul de viaţă a unei variabile?
13. Ce loc ocupă declaraţiile variabilelor locale în corpul unei funcţii?
14. Ce reprezintă antetul unei funcţii?
15. Ce rol are declararea funcţiilor?
16. Ce se indică în specificatorul de format al funcţiei printf ?
17. Ce sunt funcţiile cu număr variabil de parametri? Exemple.
18. Ce sunt funcţiile cu parametri impliciţi?
19. Ce sunt pointerii către funcţii?
20. Ce sunt variabilele referinţă?
21. Cine determină timpul de viaţă şi domeniul de vizibilitate ale unei variabile?
22. Comparaţie între declararea şi definirea funcţiilor.
23. Diferenţe între modurile de transfer a parametrilor prin valoare şi prin referinţă.
24. Diferenţe între modurile de transfer a parametrilor unei funcţii prin pointeri şi prin referinţă.
25. Din apelul funcţiei printf se poate omite specificatorul de format?
26. Din ce este formată o funcţie?
27. În ce zonă de memorie se rezervă spaţiu pentru variabilele globale?
28. O funcţie poate fi declarată în corpul altei funcţii?
29. O funcţie poate fi definită în corpul unei alte funcţii?
30. Parametrii formali ai unei funcţii sunt variabile locale sau globale?
31. Transferul parametrilor prin valoare.
32. Ce rol au parametrii formali ai unei funcţii?

Chestiuni practice

1. Să se implementeze programele cu exemplele prezentate.
2. Să se scrie programele pentru exerciţiile rezolvate care au fost prezentate.
3. Să se modularizeze programele din capitolul **4** (3.a.-3.g., 4.a.-4.i, 5.a.-5.h.), prin implementarea unor funcţii (funcţii pentru: citirea elementelor unui vector, afişarea vectorului, calculul sumei a doi vectori, calculul produsului scalar a doi vectori, aflarea elementului minim din vector, citire a unei matrici, afişare a matricii, calculul transpusei unei matrici, calculul sumei a două matrici, calculul produsului a două matrici, calculul produsului elementelor din triunghiul haşurat, etc.).
4. Să se rescrie programele care rezolvă exerciţiile din capitolul **3**, folosind funcţii (pentru calculul factorialului, aflarea celui mai mare divizor comun, ordonarea lexicografică a caracterelor, etc). Utilizaţi funcţiile de intrare/ieşire `printf` şi `scanf`.
5. Să se scrie un program care citeşte câte două numere, până la întâlnirea perechii de numere 0, 0 şi afişează, de fiecare dată, cel mai mare divizor comun al acestora, folosind o funcţie care îl calculează.
6. Se introduce de la tastatura un număr întreg. Să se afişeze toţi divizorii numărului introdus. Se va folosi o funcţie de calcul a celui mai mare divizor comun a 2 numere.
7. Secvenţele următoare sunt corecte din punct de vedere sintactic? Dacă nu, identificaţi sursele erorilor.

```
❑ void a(int x, y)     {cout<<"x="<<x<<" y="<<y<<'\n'; }
  void main( )   {     int b=9; a(6, 7); }
❑ void main( )   { int x=8; double y=f(x); cout<<"y="<<y<<'\n';}
  int f(int z)   {return z+z*z;}
```

8. Scrieţi o funcţie `găseşte_cifra` care returnează valoarea cifrei aflate pe poziţia k în cadrul numărului n, începând de la dreapta (n şi k vor fi argumentele funcţiei).
9. Implementaţi propriile versiuni ale funcţiile de lucru cu şiruri de caractere (din paragraful 4.4).
10. Să se calculeze valoarea funcţiei `g`, cu o eroare EPS (a, b, EPS citite de la tastatură):

$$g(x)=\int_a^b \sqrt{(x^2+x+1)}*\ln|x+a|\,dx + \int_a^b x*arctg(b/(b+x))\,dx$$

11. Implementaţi funcţii iterative şi recursive pentru calculul valorilor polinoamelor Hermite $H_n(y)$, ştiind că: $H_0(y)=1$, $H_1(y)=2y$, $H_n(x)=2yH_{n-1}(y)-2H_{n-2}(y)$ dacă n>1. Comparaţi timpul de execuţie al celor două funcţii.
12. Să se scrie un program care generează toate numerele palindrom, mai mici decât o valoare dată, LIM. Un număr palindrom are cifrele simetrice egale (prima cu ultima, a doua cu penultima, etc). Se va folosi o funcţie care testează dacă un număr este palindrom.
13. Fie matricea C (NxN), N<=10, ale cărei elemente sunt date de relaţia:

$$C_{i,j} = \begin{cases} j! + \sum_{k=0}^{j} \sin(kx), & \text{dacă } i<j \\ x^i, & \text{dacă } i=j \\ i! + i\sum_{k=0} \cos(kx), & \text{dacă } i>j \end{cases}$$

, unde $x \in [0,1]$, x introdus de la tastatură

   a) Să se implementeze următoarele funcţii: de calcul a elementelor matricii; de afişare a matricii; de calcul şi de afişare a procentului elementelor negative de pe coloanele impare (de indice 1, 3, etc).
   b) Să se calculeze şi să se afişeze matricea B, unde: $B=C - C^2 + C^3 - C^4 + C^5$.
14. Să se creeze o bibliotecă de funcţii pentru lucrul cu matrici, care să conţină funcţiile utilizate frecvent (citirea elementelor, afisarea matricii, adunare a două matrici, etc). Să se folosească această bibliotecă.
15. Să se creeze o bibliotecă de funcţii pentru lucrul cu vectori, care să conţină funcţiile utilizate frecvent. Să se folosească această bibliotecă.