

CAPITOLUL I

1. SUBPROGRAME

1.1. Prezentare generală

Să ne imaginăm că dorim să scriem un program în care să citim o mulțime de numere întregi ale căror proprietăți dorim să le studiem și eventual să efectuăm în funcție de proprietățile lor diferite operații cu numerele citite.

De exemplu presupunem că dorim să știm dacă numerele citite sunt prime sau nu, dacă nu sunt prime dorim să le descompunem în factori primi, vrem să verificăm dacă au proprietatea de palindrom sau să le convertim în alte baze și altele.

Operațiile pe care le-am înșirat mai sus sunt suficiente să ne dăm seama că programul pe care urmează să-l scriem este lung, greu de scris de urmărit și de corectat în situația în care mai facem și greșeli pe parcursul scrierii lui.

Ce este de făcut pentru a nu abandona înainte de a începe rezolvarea unei probleme complexe?

Limbajul C++ ne pune la dispoziție instrumentele necesare pentru a ne ușura munca, pentru a nu fi nevoiți să repetăm de multe ori aceleași secvențe de program, pentru a fi ușor de gândit, de realizat, de corectat și de înțeles și de către o altă persoană decât cea care l-a scris.

Ne rămâne doar să mai învățăm câte ceva despre subprograme.

Utilizarea subprogramelor ne permite accesul la ceea ce poate oferi programarea structurată în C++.

Beneficiem astfel de următoarele avantaje:

- Putem **reutiliza cu ușurință codul scris**.
- **Elaborarea algoritmilor devine mult mai ușoară** datorită descompunerii problemei în probleme mai simple, mai ușor de rezolvat.
- **Corectarea erorilor se realizează cu mai mare ușurință**.
- Descompunând problema și rezolvând la un moment dat probleme mai simple, **reducem numărul erorilor** la elaborarea algoritmilor.

În C++ subprogramele sunt de tip funcție.

Exemplu:

Se citesc două numere întregi. Să se calculeze suma lor.

```
#include <iostream.h>           //SUMA
int suma(int a, int b);         //antetul funcției
void main()
{int x,y,z;
 cout<<"x="; cin>>x;
 cout<<"y="; cin>>y;
 z=suma(x,y);                   //apelul funcției suma
 cout<<"suma="<<z;
 }
int suma(int a, int b)          //funcția suma de tip int, cu parametrii a și b de tip int
{int c;
 c=a+b;
 return c;
}
```

În exemplul de mai sus am definit funcția **suma**, de tip întreg, cu parametrii **a** și **b** de tip întreg. După cum se poate observa funcția returnează o valoare întreagă, adică de același tip cu tipul funcției.

return c;

Funcția se apeleză prin numele ei, are două argumente, **x** și **y** de același tip cu parametrii funcției.

La apelul funcției valoarea returnată de funcție se depune în variabila **z**.

Este necesar să facem câteva precizări legate de variabile, înainte de a studia mai amănunțit funcțiile.

1.2. Variabile globale și variabile locale

Domeniul de vizibilitate, durata de viață și zona de memorie alocată

Variabilele pot fi definite în orice poziție într-un program, astfel putem defini variabile înafara funcțiilor (chiar înafara funcției principale main), în acest caz poartă numele de **variabile globale**, în interiorul unei funcții caz în care le vom numi **variabile locale**.

În realitate în C++ variabilele pot fi definite în orice punct al programului. Sigur nu este indiferentă poziția în care se definește o variabilă, de aceasta depinde domeniul de vizibilitate al variabilei respective, adică zona de program din care poate fi accesată variabila.

Domeniul de vizibilitate al unei variabile începe în momentul declarării ei și sfârșește în momentul în care se încheie structura în interiorul căreia a fost definită.

1. **Variabilele globale**, declarate la început de program (înaintea oricărei funcții) sunt accesibile din orice punct al programului.

Dacă există funcții definite înaintea acestor variabile, pentru acele funcții variabilele nu sunt vizibile (nu pot fi accesate din interiorul acestor funcții).

Durata de viață a unei variabile globale, este atâta timp cât programul se execută.

Are alocat spațiu de memorie pe tot parcursul execuției programului.

La declarare, variabilele globale se inițializează în mod automat cu 0.

2. **Variabilele locale** definite în interiorul unei funcții, sunt accesibile doar funcției, nu pot fi utilizate înafara ei.

Durata de viață a unei variabile locale este pe tot parcursul execuției funcției dacă a fost definită la începutul blocului de instrucțiuni.

Are alocată zonă de memorie numai pe parcursul execuției funcției respective.

Dacă o variabilă este definită în interiorul unui bloc de instrucțiuni, poate fi accesată doar în interiorul acelui bloc, nu poate fi utilizată înafara lui.

Durata de viață a acestei variabile este până la terminarea execuției blocului în care a fost definită.

Are alocată zonă de memorie numai pe parcursul execuției blocului respectiv.

Variabilele locale nu se inițializează în mod automat. Dacă nu le inițializează programatorul, ele rețin o așa numită *valoare reziduală*, adică ceea ce se găsește în memorie în acel moment.

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

```
float a;
```

-**variabilă globală** accesibilă pe tot parcursul programului

```
void main()
```

```
{
```

```
char b;
```

- **variabilă locală** accesibilă doar în interiorul funcției main

```
{
```

```
int c=5;
```

-**variabilă locală accesibilă doar în interiorul blocului** în care a fost definită

```
cout<<a<<b<<c;
```

```
}
```

```
}
```

Observație:

În cazul în care într-un bloc sau într-o funcție sunt vizibile mai multe variabile cu același nume, dar cu domenii de vizibilitate diferite, se accesează variabila cu domeniul cel mai mic de vizibilitate.

Exemplu:

În programul de mai jos sunt definite două variabile cu numele **a**. O variabilă globală și o variabilă locală definită în cadrul funcției **f**.

În timpul execuției, funcția **f accesează variabila locală**.

```
#include <iostream.h>
int a; //a este variabilă globală

int f() //funcția suma de tip int, cu parametrii a și b de tip int
{int a=5; //funcția folosește variabila locală a(definită în funcție), nu pe cea globală
  a++;
  return a;
}

void main()
{
  a=f(); //apelul funcției suma
  cout<<"a="<<a;
}
```

Rezultatul rularii programului:

a=6

În ceea ce privește **zona de memorare a variabilelor**, putem spune că și din acest punct de vedere sunt diferențe între variabilele globale și cele locale.

Sistemul de operare alocă fiecărui program trei zone distincte de memorie internă, pentru memorarea variabilelor programului.

Există și posibilitatea ca o variabilă să fie memorată într-un registru al microprocesorului. Timpul de acces la variabilele memorate în registrul este foarte mic și astfel poate fi îmbunătățit timpul de execuție al programului.

Zonele de memorie internă alocte:

Segmentul de date	-Variabile globale
Segmentul de stivă	-Variabile locale
Heap	-Date alocate dinamic

1. **Variabilele globale** se memorează în cadrul **segmentului de date**.
2. **Variabilele locale** se memorează pe **segmentul de stivă** pus la dispoziția funcției în interiorul careia sunt definite.
3. Datele alocate dinamic nu constituie obiectul acestui manual.

1.3. Structura funcțiilor și apelul lor

O **funcție** se compune din **antet** și **corpul funcției**.

- **Antetul** unei funcții cuprinde tipul de dată returnat de funcție, numele funcției și lista parametrilor.
- **Corpul funcției** este format din una sau mai multe instrucțiuni cuprinse între acolade.

1.3.1. Formatul unei funcții:

tip nume(parametru1, parametru2, ...) { instrucțiuni }

- **tip** – reprezintă un tip de dată și desemnează tipul datei returnate de funcție (nu este obligatoriu).
- **nume** – este un identificator prin intermediul căruia se apelează funcția.
- **parametrii** – permit transmiterea argumentelor către funcție atunci când este apelată. Fiecare parametru constă dintr-un tip de dată urmat de un identificator, ca orice variabilă și acționează ca o variabilă locală.
- **Instrucțiuni** – reprezintă corpul funcției și trebuie să fie cuprins între acolade {...}.

1.3.2. Declararea și definirea unei funcții

A **declara o funcție** înseamnă a o “anunța”, adică a face cunoscut programului că această funcție există și caracteristicile ei, adică tipul, numele și lista eventualilor parametri. Declararea unei funcții se face cu ajutorul antetului funcției.

tip nume(parametru1, parametru2, ...);

A defini o funcție înseamnă a o descrie împreună cu corpul funcției.

tip nume(parametru1, parametru2, ...) { instrucțiuni }

definiția unei funcții ține loc și de declarație în cazul în care funcția este plasată înaintea funcției principale *main()*.

1.3.3. Structura unui program care conține subprograme

Un program poate să conțină oricâte funcții.

Așa cum o variabilă nu poate fi accesată în limbajul de programare C înainte de a fi definită, nici o funcție nu poate fi apelată fără ca programul să știe caracteristicile acelei funcții. Pentru a putea fi apelată funcția trebuie să fie cel puțin declarată.

Există două modalități de a plasa funcțiile într-un program:

a. **Înainte funcției principale main()**, ca în exemplul de mai jos:

```
#include <iostream.h>
int f()
{int a=3;
 return a;
}
void main()
{cout<<f;
}
```

b. **După funcția principală main()**, caz în care funcția trebuie declarată înainte,

```
#include <iostream.h>
int f();           //declararea funcției f

void main()
{cout<<f;
}
int f()           //definirea funcției f
{int a=3;
 return a;
}
```

1.3.4. Apelul unei funcții

O funcție poate fi apelată din funcția principală *main*, dintr-o altă funcție sau chiar din funcția însăși. Acest din urmă caz îl vom trata într-un capitol separat.

După cum am spus în paragraful 1.3.1, antetul unei funcții poate conține tipul datei returnate de funcție sau nu. Aceasta datorită faptului că o funcție poate să returneze o valoare sau nu.

Din acest punct de vedere putem împărți funcțiile în două categorii:

1. **Funcții care returnează o valoare**, definite astfel:

tip nume(parametru1, parametru2, ...) { instrucțiuni }

O funcție care returnează o valoare trebuie să conțină linia de cod:

return expresie;

Rezultatul returnat de funcție poate fi utilizat într-o expresie sau poate fi afișat.

Prin urmare apelul unei funcții care returnează o valoare poate fi de forma:

variabila=nume(parametrul1,parametrul2,...);

sau

cout<<nume(listă parametrii);

Exemplu:

Să se însumeze primele n numere naturale.

```
#include <iostream.h>
#include <conio.h>
int n;                //declararea variabilei globale
int suma();           //declararea funcției – funcția nu are parametrii

void main()
{cout<<"n="; cin>>n;
  cout<<"suma primelor n numere naturale:"<<suma();    //apelul funcției
  getch();
}

int suma()             //definirea funcției
{int i,s=0;
  for(i=1;i<=n;i++)
    s+=i;
  return s;            //valoarea returnata de funcție
}
```

2. Funcțiile care nu returnează o valoare, în loc de *tip* vor conține cuvântul **void**

void nume(parametru1, parametru2, ...) { instrucțiuni }

Apelul unei astfel de funcții:

nume(parametrul1,parametrul2,...);

```
#include <iostream.h>          //AFISARE
int n;
int afisare();                 //declararea funcției – funcția nu are parametrii

void main()
{cout<<"n="; cin>>n;
  afisare();                   //apelul funcției
}

int afisare()                  //definirea funcției
{int i,s=0;
  for(i=1;i<=n;i++)
    cout<<i<<" ";
}
```

Observație:

O funcție poate să nu conțină parametrii.

În acest caz atât la declarare cât și la definire și apel numele funcției va fi urmat de paranteze rotunde, dar fără să conțină nici un parametru.

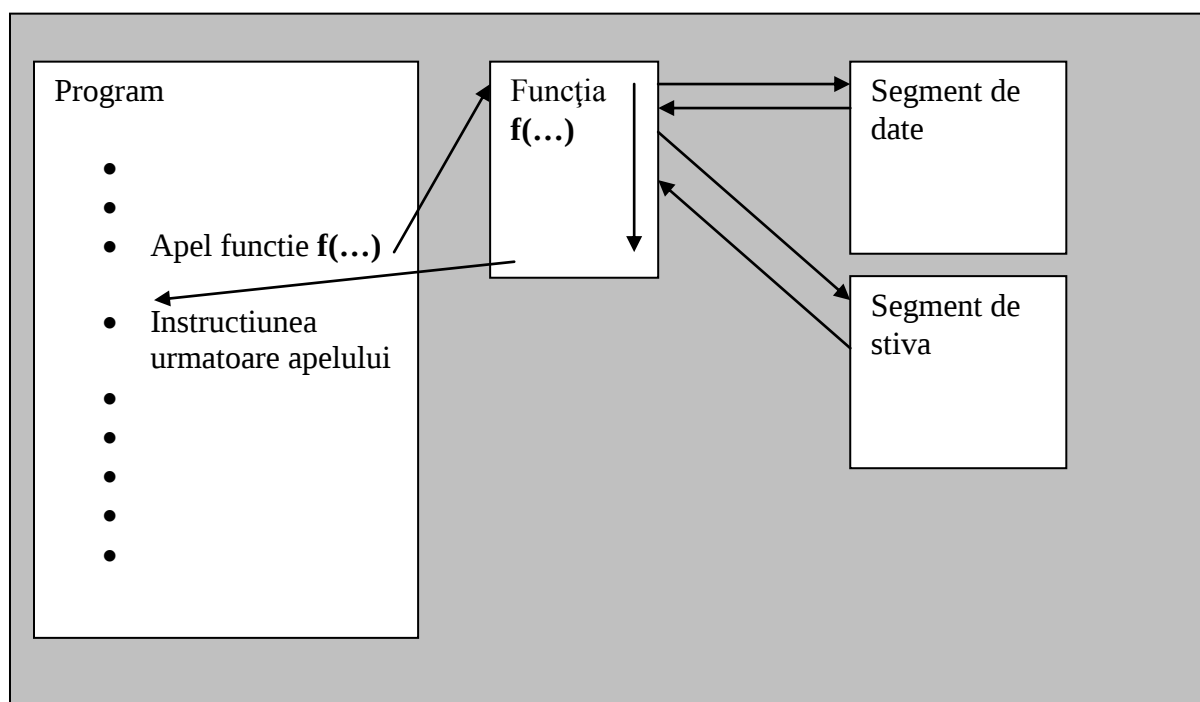
Parantezele sunt obligatorii pentru a informa compilatorul că este vorba despre o funcție și nu de o variabilă.

1.3.5. Modul de funcționare al unui program care conține una sau mai multe funcții

În momentul în care se apelează o funcție controlul programului se va da funcției respective pentru a fi executată.

În momentul apelării unei funcții se întâmplă mai următoarele:

- **Se alocă spațiu pe segmentul de stivă unde se memorează parametrii funcției** (acă funcția are parametrii).
- Se **salvează adresa instrucțiunii următoare apelului**, pentru ca la revenirea din funcție programul să continue cu instrucțiunea următoare.
- Salt la prima instrucțiune a funcției
- **Se alocă spațiu pe segmentul de stivă unde se memorează variabilele locale** (dacă au fost declarate variabile locale).
- Se **execută în secvență instrucțiunile funcției** apelate.
- Se **eliberează segmentul de stivă alocat funcției**.
- Se **revine în punctul în care s-a apelat funcția** și programul **continuă cu instrucțiunea următoare**



1.3.6. Transmiterea parametrilor

Parametrii unei funcții au rolul de a transmite date funcției și/sau de a transmite date din funcție spre zona de program care a apelat funcția. Parametrii se descriu prin tipul și numele lor, în antetul funcției, despărțiți între ei prin virgulă.

Parametrii care se află în antetul funcției se numesc **parametri formali** iar cei care se află în instrucțiunea de apel a funcției se numesc **parametri efectivi**.

Numele (identificatorii) parametrilor formali poate să fie diferit de numele parametrilor efectivi sau poate să coincidă.

Transmiterea parametrilor se face prin corespondență, din acest motiv între parametri formali și cei efectivi trebuie să existe o anumită concordanță:

- **Numărul** parametrilor formali trebuie să coincidă cu numărul parametrilor reali. Există o singură excepție și anume parametri cu valoare implicită, despre care vom vorbi mai târziu.
- **Ordinea** parametrilor formali trebuie să fie aceeași cu cea a parametrilor efectivi, deoarece după cum am spus transmiterea parametrilor se face prin corespondență.
- **Tipul** parametrilor efectivi trebuie să poată fi convertit implicit în tipul parametrilor formali sau să coincidă.

Exemplu:

```
int suma (int a, float b)
      ↑      ↑
x=suma(  2,  3.14  );
```

- Parametrii se memorează pe segmentul de stivă pus la dispoziția funcției în momentul apelării ei. Aceștia se memorează în ordinea în care au fost trecuți în antetul funcției.
- Parametrii transmiși și memorați pe segmentul de stivă sunt variabile locale, iar numele lor este cel din lista de parametri formali.

Din punctul de vedere al modului de transmitere a parametrilor putem împărți parametrii în două categorii: **parametrii transmiși prin valoare** și **parametrii transmiși prin referință**.

1. Parametrii transmiși prin valoare

Parametrii transmiși prin valoare au rolul de a transmite date din zona de program în care s-a apelat funcția înspre funcție.

Funcția va memora aceste date pe segmentul de stivă pus la dispoziție. După terminarea execuției funcției, segmentul de stivă se eliberează și astfel programul nu mai are acces la variabilele memorate pe stivă. Din acest motiv parametri transmiși prin valoare nu pot transmite datele modificate în cadrul funcției înspre zona de program de unde a fost apelată funcția.

Prin valoare se pot transmite: **variabile**, **expresii** și chiar **funcții**.

Exemplu:

```
#include <iostream.h>          //PARAMETRII TRANSMISI PRIN VALOARE
                                //parametrii transmisi prin valoare

void f(int a,int b);

void main()
{ int x=2,y=3;
  f(x,y);                      //parametrii efectivi sunt variabile
  cout<<"x="<<x<<" y="<<y<<endl;    //la revenire x si y sunt nemodificati
  f(2*x,2*y);                  //parametrii efectivi sunt expresii
  cout<<"x="<<x<<" y="<<y;          //la revenire x si y sunt nemodificati
}

void f(int a,int b)            //functia modifica valorile parametrilor, dar nu le transmite inafara
{ a=a+2;
  b=a+3;
  cout<<"a="<<a<<" b="<<b<<endl;
}
```

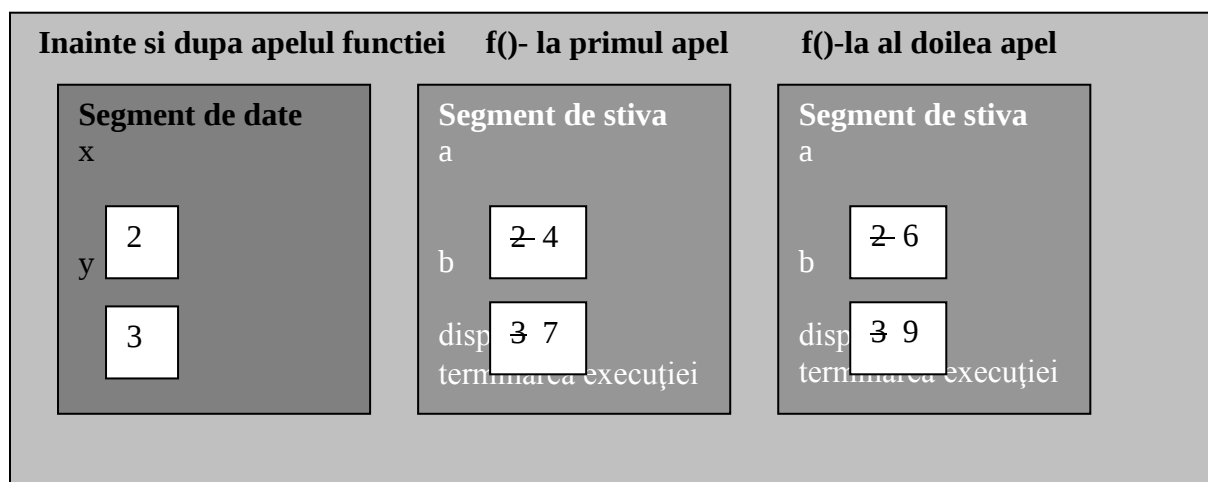
Rezultatul rulării programului:

```
a=4 b=7
x=2 y=3
a=6 b=9
x=2 y=3
```

Observați că în urma apelării funcției **f()**, deși în interiorul funcției variabilele **a** și **b** și-au modificat valorile, noile valori nu s-au transmis variabilelor **x** și **y**, acestea rămân nemodificate.

Acest lucru se întâmplă datorită faptului că pe segmentul de stivă **s-a creat o copie** a variabilelor **x** și **y** cu numele **a** și **b**, iar funcția operează asupra acestora.

La terminarea execuției funcției, se eliberează segmentul de stivă iar valorile variabilelor **a** și **b** se pierd.



La declararea unei funcții putem specifica o valoare implicită pentru unii parametrii.

În cazul în care nu există parametri efectivi corespunzători, vor fi folosite de către funcție valorile implicite.

În cazul în care se transmit parametri efectivi, se vor folosi valorile acestora.

Priviți exemplul de mai jos:

```
#include <iostream.h>
void afisare(int a,int b=1,int c=2); //b si c sunt parametri cu valori implicite

void main()
{ //am specificat valori pentru toti parametri, functia le va utiliza pe cele specificate ca
  //parametri efectivi
  afisare(3,4,5);
  //am specificat valori doar pentru a si b, pentru c se va folosi valoarea implicita
  afisare(5,6);
  //am specificat valori doar pentru a , pentru b si c se vor folosi valorile implicite
  afisare(7);
}

void afisare(int a,int b,int c) //nu se specifică de două ori valoarea implicită
{ cout<<"a="<<a<<" b="<<b<<" c="<<c<<endl;
}
}
```

În exemplul de mai sus antetul funcției conține doi parametri cu valori implicite b=1 și c=2.

La apelul **afisare(3,4,5);** s-au transmis toți cei 3 parametri iar funcția a folosit valorile transmise.

La apelul **afisare(5,6);** s-au transmis 2 parametri iar funcția a folosit valorile transmise pentru a și b și valoarea implicită pentru c.

La apelul **afisare(7);** s-a transmis doar valoarea parametrului a iar pentru b și c s-au folosit valorile implicite.

2. Parametrii transmiși prin referință

De multe ori este necesar să modificăm în interiorul unei funcții valorile unei variabile externe funcției. În acest scop se folosesc parametri transmiși prin referință.

Prin urmare rolul parametrilor transmiși prin referință este ca la revenirea din funcție, variabila transmisă, să rețină valoarea modificată în timpul execuției funcției.

Pentru ca un parametru să fie transmis prin referință **numele lui trebuie să fie precedat de caracterul ampersand &**, la declararea parametrilor, în antetul funcției.

Caracterul ampersand specifică faptul că parametrul este transmis prin referință și nu prin valoare.

Exemplu:

```
int f(int &a, int &b);  
      ↑      ↑  
f (  x   ,   y  );
```

Atunci când o variabilă este transmisă prin referință, **nu** se face o copie a ei pe segmentul de stivă ci **se reține pe segmentul de stivă adresa variabilei însăși**. Cu alte cuvinte modificările se efectuează la adresa reținută, prin urmare chiar asupra variabilei transmise.

O funcție poate avea unul sau mai mulți parametri transmiși prin referință.

Parametrii efectivi corespunzători parametrilor transmiși prin referință trebuie să fie nume de variabile.

Să analizăm următorul exemplu:

```
#include <iostream.h>    //PARAMETRII TRANSMISI PRIN REFERINTA  
void f(int &a,int b); //parametrul a este transmis prin referinta  
                        //parametrul b este transmis prin valoare  
  
void main()  
{ int x=1,y=2;  
  cout<<"x="<<x<<" y="<<y<<endl;    //x si y inainte de apelarea functiei  
  f(x,y);  
  cout<<"x="<<x<<" y="<<y<<endl;    //x si y dupa apelarea functiei  
}  
  
void f(int &a,int b)  
{a=2*a;                //modificarea valorilor prametrilor a si b  
 b=2*b;  
 cout<<"a="<<a<<" b="<<b<<endl;    // a si b in functie  
}
```

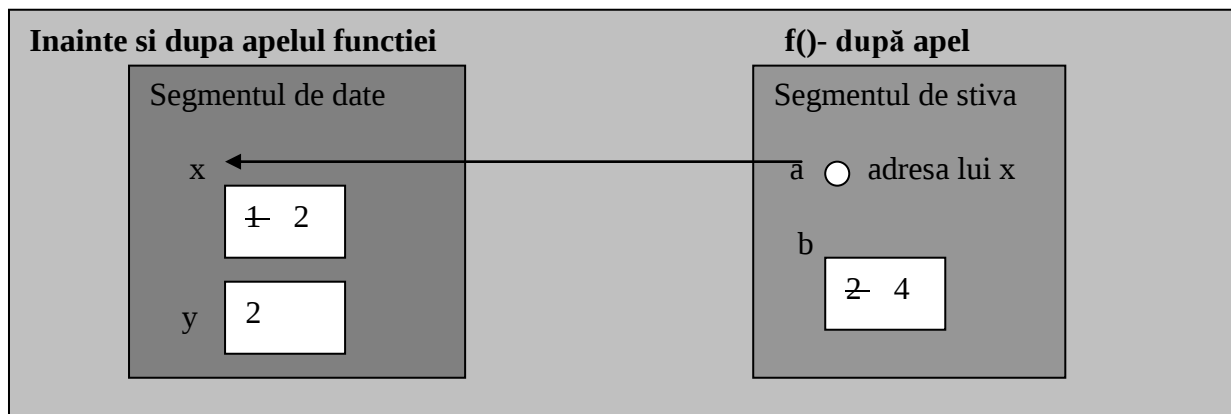
Rezultatul rulării:

```
x=1 y=2  
a=2 b=4  
a=2 b=2
```

Parametrul **a** este transmis prin referință iar parametrul **b** este transmis prin valoare.

Inițial **x=1** și **y=2**, în cadrul funcției **a devine 2** iar **b devine 4**.

Deoarece **a** este un parametru transmis prin referință valoarea lui este transmisă în **x**. Întrucât **b** este parametru transmis prin valoare, **y nu** preia valoarea modificată a lui **b**



1.4. Probleme rezolvate

1. Se citește o literă și un număr întreg n. Să se calculeze:

-pentru litera a $S1=1^2 + 3^2 + 5^2 + \dots + (2n-1)^2$

-pentru litera b $S2=1 + 1*2+1*2*3 + \dots + 1*2*3\dots n$

```
#include <iostream.h>      //SUME
#include <math.h>           //conține funcțiile matematice
long s1(int n);
long s2(int n);

void main()
{ int n; char lit;          //n-nr. elemente, lit-litera citita
  cout<<"n="; cin>>n;
  cout<<"lit(a sau b)="; cin>>lit;
  if(lit=='a')
    cout<<"S1="<<s1(n);
  else
    cout<<"s2="<<s2(n);
}

long s1(int n)              //funcția s1 are un parametru transmis prin valoare
{ int i; long s=0;
  for(i=1;i<=n;i++)
    s+=pow(2*i-1,2);
  return s;                 //funcția s1 returnează valoarea calculată
}

long s2(int n)              //funcția s2 are un parametru transmis prin valoare
{ int i;
  long s=0,p=1;
  for(i=1;i<=n;i++)
    {p*=i;
     s+=p; }
  return s;                 //funcția s2 returnează valoarea calculată
}
```

2. Se citește un număr natural n . Să se numere câte cifre pare și câte cifre impare are numărul.

Vom realiza o funcție *cifre* pentru numărarea cifrelor pare și impare, căreia îi transmitem 3 parametrii.

n – numărul citit, **parametru transmis prin valoare**

c_pare -numărul cifrelor pare, **parametru transmis prin referință**, care va transmite numărul cifrelor pare ale numărului, din funcție înspre programul apelant.

c_impare -numărul cifrelor impare, **parametru transmis prin referință**, care va transmite numărul cifrelor impare ale numărului, din funcție înspre programul apelant.

Funcția *cifre* va returna programului principal numărul cifrelor pare și numărul cifrelor impare, prin intermediul parametrilor transmiși prin referință.

```
#include <iostream.h>          //NR. CIFRE PARE SI IMPARE
void cifre(int n, int &x, int &y); //declararea functiei cifre

void main()
{long n;
 int c_pare=0,c_impare=0; //c_pare-nr. cifrelor pare,c_impare-nr. Cifrelor impare
 cout<<"n="; cin>>n;
 cifre(n,c_pare,c_impare);
 cout<<"nr. cifre pare="<<c_pare<<endl;
 cout<<"nr. cifre impare="<<c_impare<<endl;
}

void cifre(int n, int &x, int &y) //definirea functiei cifre
{ while (n)                      //cat timp numarul mai are cifre
 { if ((n%10)%2==0)              //verifica daca ultima cifra a numarului este para
   x++;                          //incrementează numărul cifrelor pare
   else
   y++;                          //incrementează numărul cifrelor impare
   n/=10;                       //elimina ultima cifra
 }
}
```

3. Se citește un număr natural n . Să se verifice și să se afișeze dacă este un număr prim, altfel să se afișeze descompunerea în factori primi.

Definim o funcție *prim*, care primește ca parametru transmis prin valoare numărul n și returnează 1 dacă numărul este prim și 0 dacă numărul nu este prim.

Definim de asemenea o funcție *factori*, funcție fără tip, care primește ca parametru numărul n și afișează factorii primi și ordinul lor de multiplicitate.

```

#include <iostream.h>          //PRIM si FACTORI PRIMI
int prim(int n);             //declararea funcției prim, n parametru transmis prin valoare
void factori(int n);        //declararea funcției factori, n parametru transmis prin valoare

void main()
{long n;
  cout<<"n="; cin>>n;
  if(prim(n))                //apelul funcției prim ca expresie logică
    cout<<n<<" este numar prim";
  else
    {cout<<"descompunerea in factori primi: ";
      factori(n);           //apelul funcției factori
    }
}

int prim(int n)            //definirea functiei prim
{int i;
  if (n==0 || n==0)
    return 0;
  for(i=2;i<=n/2;i++)
    if(n%i==0)
      return 0;
  return 1;
}

void factori(int n)        //definirea functiei factori
{int i=2,k;
  while (n!=1)
    {k=0;
      while(n%i==0)
        {k++;
          n/=i;
        }
      if (k)
        cout<<i<<"^"<<k<<" ";
      i++;
    }
}

```

4. Se citește un număr natural n . Să se descompună în sumă de numere fibonacci.

Definim funcția *max_f*, care determină cel mai mare număr fibonacci mai mic sau egal cu un număr dat. Funcția primește ca parametru transmis prin valoare numărul dat.

Definim funcția *fibonacci*, care apelează funcția *max_f* și astfel determină *max*, cel mai mare număr fibonacci mai mic sau egal cu numărul dat, apoi scade din numărul dat, numărul *max*.

Se reia algoritmul atâta timp cât numărul dat este diferit de zero.

//DESCOMPUNEREA UNUI NUMAR IN SUMA DE NUMERE FIBONACCI

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

```
void fibo(unsigned long m);           // declararea funcției fibo  
unsigned long max_f(unsigned long nr); //declararea funcției max_f
```

```
void main()  
{unsigned long n;  
  clrscr();  
  cout<<"n=";<<cin>>n;  
  cout<<n<<" = ";  
  fibo(n);           //apelul funcției fibo  
}
```

```
unsigned long max_f(unsigned long nr) //definirea funcției max_f  
{unsigned long a,b,c;  
  a=0;b=1;  
  while(b<=nr)  
  {c=a+b;  
   a=b;  
   b=c;  
  }  
  return a;           //valoarea returnată de funcția max_f  
}
```

```
void fibo(unsigned long m)           //definirea funcției fibo  
{int max;  
  while (m)  
  {max=max_f(m);           //apelul funcției max_f din interiorul funcției fibo  
   cout<<"+"<<max;  
   m-=max;  
  }  
}
```

5. Se citește un număr natural n în baza 10 și b un număr natural $2 \leq b \leq 9$ reprezentând o bază de numerație. Să se transforme numărul n în baza b .

Vom scrie o funcție *conv*, de tip void, cu doi parametri transmiși prin valoare, respectiv n și b .

Rezultatul conversiei îl vom memora într-un tablou unidimensional t .

În finalul funcției afișăm conținutul tabloului în ordinea inversă construirii lui.

```

//CONV 10->b

#define MAX 20
void conv(int n, int b);    //antetul funcției

void main()
{int n,b;
 cout<<"n="; cin>>n;      //numărul în baza 10
 cout<<"b="; cin>>b;      //baza de numerație
 conv(n,b);               //apelul funcției de conversie
}

void conv(int n, int b)    //definirea funcției conv
{int i=0,j,t[MAX];
 while (n)
 {t[i++]=n%b;              //restul împărțirii lui n la b se depune în tabloul t
  n=n/b;                   //n devine câtul împărțirii lui n la b
 }
 for(j=i-1;j>=0;j--)      //afisarea numărului în baza b
  cout<<t[j];
}

```

6. Se citește un număr natural n și apoi n cifre reprezentând un număr în baza b , $1 \leq b \leq 9$. Să se transforme numărul dat din baza b în baza 10.

Vom realiza două funcții. Funcția citire, pentru citirea numărului în baza b și a bazei. Funcția va avea 3 parametri:

- n , numărul de cifre, parametru transmis prin referință
- t - tablou unidimensional care va conține cifrele numărului, acest parametru se transmite prin valoare deși rezultatul citirii trebuie transmis și celorlalte funcții. Întrucât numele unui tablou este adresa primului octet din tablou. Fiind adresă, modificările se realizează în funcție chiar asupra tabloului la adresa la care este memorat tabloul în segmentul de date.
- b -baza de numerație, parametru transmis prin referință

Funcția `conv`, convertește numărul din baza b în baza 10. Returnează numărul în baza 10. Funcția are trei parametri transmiși prin valoare: n , t , b , cu semnificația de mai sus.

```

#include <iostream.h>          //CONV b->10
#include <math.h>
#define MAX 20
void citire(int &n, int t[MAX], int &b);
int conv(int n, int t[MAX], int b);

void main()
{int n,b,t[MAX],nr;  //n-nr.de cifre,b-baza,t-conține numarul in baza b
  citire(n,t,b);      //apelul funcției citire
  nr=conv(n,t,b);     //apelul funcției conv
  cout<<"nr. in baza 10= "<<nr;
}
void citire(int &n, int t[MAX], int &b)  //citeste datele de intrare
{int i;
  cout<<"baza=";cin>>b;
  cout<<"n=";cin>>n;
  cout<<"introduceti "<<n<<" cifre in baza "<<b<<":"<<endl;
  for(i=n-1;i>=0;i--)
    {cin>>t[i];}
}
int conv(int n,int t[MAX],int b)      //converteste numarul in baza 10
{int i,nr=0;
  for(i=n-1;i>=0;i--)
    nr+=t[i]*pow(b,i);
  return nr;
}

```

1.6. Probleme propuse

1. Să se afișeze numerele naturale mai mari decât 100 și mai mici decât 500 care au toate cifrele distincte utilizând o funcție care primește ca parametru un număr și returnează 1 dacă are toate cifrele distincte și 0 dacă nu sunt distincte.
2. Se citesc n numere. Să se afișeze numerele obținute prin inversarea cifrelor.
3. Să se afișeze toate numerele până la 1000 al căror pătrat se termină cu cifra n .
4. Se citesc numere naturale până la întâlnirea lui 0. Să se afișeze inversele numerelor citite, pentru care media aritmetică a cifrelor este strict mai mică decât 5.
5. Se citesc n numere naturale. Să se calculeze suma numerelor care au toate cifrele pare. Dacă nu există nici un astfel de număr se va afișa un mesaj.
6. Calculați suma și produsul divizorilor proprii ai unui număr citit.
7. Fiind dat un număr natural n , să se afișeze toți divizorii săi și media aritmetică a divizorilor săi cuprinși între două valori citite a și b sau un mesaj, dacă nu are divizori între aceste valori.
8. Fiind dat un număr natural n , afișați divizorii săi și numărați câți dintre aceștia sunt numere prime.
9. Să se genereze toate numerele prime mai mici sau egale cu un număr n citit de la tastatură.
10. Scrieți un program care calculează suma a două fracții ordinale. Afișarea se va face după simplificarea fracțiilor.
11. Se citește un număr natural n . Să se stabilească dacă acesta este un termen al șirului lui Fibonacci.
12. Se citește un număr natural n diferit de 0. Să se scrie toate tripletele de numere pitagoreice ($a^2+b^2=c^2$), mai mici decât n , nenule.
13. Se citește un număr natural n , în baza 10. Să se transforme numărul în baza 16.
14. Se citesc n caractere reprezentând un număr în baza 16. Să se transforme în baza 10.