

 Universitatea "Politehnica" din București
 Facultatea de Electronică, Telecomunicații și
 Tehnologia Informației



Programarea Calculatoarelor (limbajul C)

Curs 7 – Funcții și Recursivitate

Prof. Bogdan IONESCU

2016-2017

Cuprins

- 7.1. Funcții
- 7.2. Recursivitate

Curs Programarea Calculatoarelor, Prof. Bogdan IONESCU, 2016-2017

1/36

7.1. Funcții

Curs Programarea Calculatoarelor, Prof. Bogdan IONESCU, 2016-2017

2/36

Definirea funcțiilor

- > Pentru rezolvarea unor probleme complexe de calcul, acestea trebuie descompuse în **sub-probleme** mai simple.
 - "divide et impera" ar spune romanii.
- > Descompunerea fizică a codului în porțiuni, ce realizează anumite sub-probleme de calcul, se face cu ajutorul **funcțiilor**.
 - modulare program (fiecare modul se ocupă de anumite operații).

> Pe lângă structurarea programului, funcția are avantajul de a fi scrisă **o singură dată**, dar folosită **de câte ori este nevoie**, evitând astfel rescrierea codului aferent de fiecare dată.

Curs Programarea Calculatoarelor, Prof. Bogdan IONESCU, 2016-2017

3/36

Subprograme – funcții (continuare)

- > Ce înseamnă o funcție ?
- > Din punct de vedere **matematic**:

valoarea returnată

$f(x,y) = \sin(x*y)$

corpul funcției (calculul realizat)

numele funcției

argumente funcție (variabile)

- > Din punct de vedere al **programării** în C, funcția își păstrează sensul matematic (programare funcțională).

Curs Programarea Calculatoarelor, Prof. Bogdan IONESCU, 2016-2017

4/36

Definirea funcțiilor (continuare)

- > Astfel, în limbajul C, funcțiile își păstrează sensul matematic:

Formă generală:

```

<tip_dată> <nume>(<var1>, ..., <varN>)
{
  <secvență instrucțiuni 1>;
  <secvență instrucțiuni 2>;
  ...
  <secvență instrucțiuni M>;
  [return <expresie>;]
}
  
```

[] = opțional

- **<tip_dată>**: reprezintă tipul datelor returnate de funcție, **Atenție: o funcție nu poate returna decât o singură valoare.**

Curs Programarea Calculatoarelor, Prof. Bogdan IONESCU, 2016-2017

5/36

Definirea funcțiilor (continuare)

• **<tip_dată>**: dacă funcția nu returnează nici o valoare atunci se specifică tipul **void** → procedură.

• **<nume>**: reprezintă identificatorul funcției pe baza căruia aceasta va fi apelată pe parcursul rulării programului.

• (var1,...,varN): reprezintă variabilele de intrare, pe care le primește funcția din programul, sau sub-programul în care este apelată. Este posibil ca funcția să nu primească date de intrare, caz în care lista este vidă: <nume>(), sau se poate folosi tipul void: <nume>(void).

```
<tip_dată> <nume>(<var1>,...,<varN>)  
{  
  <secvență instrucțiuni 1>;  
  <secvență instrucțiuni 2>;  
  ...  
  <secvență instrucțiuni M>;  
  [return <expresie>;]  
}
```

Definirea funcțiilor (continuare)

> **Observație**: o funcție chiar dacă nu primește variabile, aceasta poate interacționa cu programul pe baza

variabilelor globale din program (vizibile pentru orice funcție)

→ *de evitat*, deoarece funcția nu mai este independentă de program, portabilitatea într-un alt program fiind redusă.

• { secvențe instrucțiuni }: corpul funcției (ceea ce se execută) este specificat între “{” și “}”. Acesta poate fi văzut, el însuși ca un program.

```
<tip_dată> <nume>(<var1>,...,<varN>)  
{  
  <secvență instrucțiuni 1>;  
  <secvență instrucțiuni 2>;  
  ...  
  <secvență instrucțiuni M>;  
  [return <expresie>;]  
}
```

Definirea funcțiilor (continuare)

• **return <expresie>**: funcția return este folosită pentru a returna valoarea dorită.

> **Effect**: funcția se încheie în momentul în care se execută comanda **return**, iar <expresia> specificată este returnată ca dată de ieșire a funcției.

→ dacă aceasta nu corespunde tipului specificat (<tip_dată>), conversie la acesta, dacă nu este posibilă, atunci **eroare**.

> **Observație**: folosirea funcției **return** este opțională chiar dacă s-a specificat că funcția trebuie să returneze ceva (warning).

```
<tip_dată> <nume>(<var1>,...,<varN>)  
{  
  <secvență instrucțiuni 1>;  
  <secvență instrucțiuni 2>;  
  ...  
  <secvență instrucțiuni M>;  
  [return <expresie>;]  
}
```

Definirea funcțiilor (continuare)

Exemplu de funcție:

```
int adunare(int a, int b)  
{  
  int r;  
  r=a+b;  
  return r;  
}
```

-am definit funcția **adunare** care primește două numere întregi și returnează un număr întreg.

-calculează suma în variabila **r** și returnează valoarea lui **r**.

> *Cum se poate rescrie funcția mai eficient ???*

Optimizare:

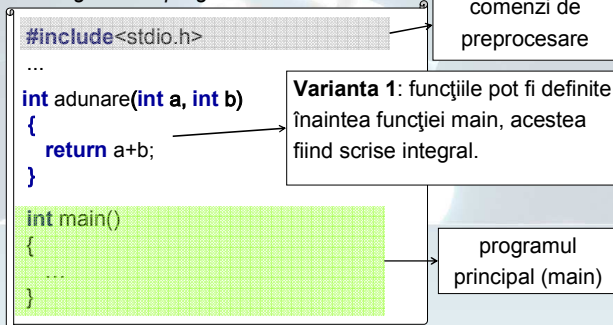
```
int adunare(int a, int b)  
{  
  return a+b;  
}
```

-funcția **return** poate primi o expresie și nu neapărat o singură variabilă.

Definirea funcțiilor (continuare)

> Unde se definesc funcțiile.

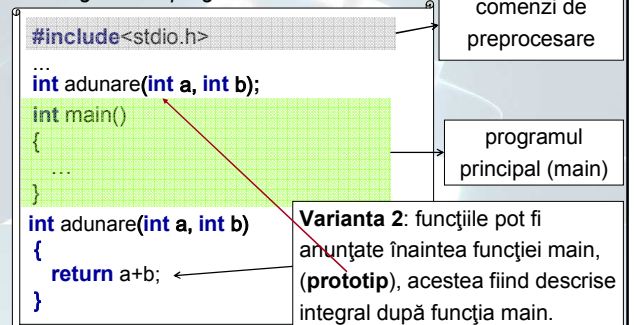
Formă generală program:



Definirea funcțiilor (continuare)

> Unde se definesc funcțiile (continuare).

Formă generală program:



Definirea funcțiilor (continuare)

> Unde se definesc funcțiile (continuare).

Exemplu:

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

double factorial(int n)
{
    int i; double fact=n;
    for (i=n-1; i>0; i--)
        fact=fact*i;
    return fact;
}

int main()
{
    printf("20!=%lf", factorial(20) );
}
```

-am definit funcția **factorial** care calculează factorialul unui număr întreg n și returnează această valoare sub forma unui double ?

-funcțiile se apelează în mod identic cu modul de apelare al funcțiilor disponibile în C.

factorial(20) →
-n=20,
-se efectuează calculele,
-**factorial(20)** devine val. specificată de **return**.

Definirea funcțiilor (continuare)

> Unde se definesc funcțiile (continuare).

Altă variantă:

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

double factorial(int n);
int main()
{
    printf("20!=%lf", factorial(20) );
}

double factorial(int n)
{
    int i; double fact=n;
    for (i=n-1; i>0; i--)
        fact=fact*i;
    return fact;
}
```

-am anunțat funcția și am specificat prototipul acesteia.

-corpul funcției este detaliat după programul principal.

-în cazul în care funcțiile nu sunt evidente (*complexe*), iar conceperea lor necesită un *timp semnificativ*, putem scrie mai întâi programul principal, ca și cum funcțiile există deja.

Definirea funcțiilor (continuare)

> Vizibilitatea variabilelor în cadrul funcțiilor

- variabilele declarate în corpul unei funcții sunt valabile doar în acesta, și doar pe parcursul execuției funcției respective.

Exemplu:

```
double factorial(int n)
{
    int i; double fact=n;
    for (i=n-1; i>0; i--)
        fact=fact*i;
    return fact;
}

int main()
{
    printf("%d, %lf", factorial(20) );
}
```

-variabila **i** este o variabilă **locală** a funcției **factorial** ce va fi alocată în memorie doar când funcția este apelată.

- variabila **i** și **fact** nu există în cadrul celorlalte funcții, inclusiv **main()**.

eroare!

Definirea funcțiilor (continuare)

> Vizibilitatea variabilelor în cadrul funcțiilor

- variabilele declarate în afara programului principal **main()**, în secțiunea de variabile globale, sunt vizibile peste tot în program: în funcția **main()**, în toate funcțiile definite, etc.

Exemplu:

```
#include<stdio.h>
int offset=3;

float produs(float a, float b)
{
    return a*b+offset;
}

int main()
{
    printf("%d, %f", offset, produs(5,6) );
}
```

-variabila **offset** este definită după comenzile de preprocesare, dar înaintea definirii funcțiilor programului,

-aceasta este astfel vizibilă atât în procedura **produs** cât și în **main**.

>3, 5*6+3=33

Definirea funcțiilor (continuare)

> Vizibilitatea variabilelor în cadrul funcțiilor

Exemplu:

```
#include<stdio.h>
float produs(float a, float b)
{
    return a*b+offset;
}

int offset=3;

int main()
{
    printf("%d, %f", offset, produs(5,6) );
}
```

-variabila **offset** este definită după definirea procedurii **produs**, ce se întâmplă ???

eroare!

variabila **offset** este vizibilă de aici în jos.

> **Principiu:** o declarație de variabile sau de funcții este vizibilă pentru toate funcțiile ce sunt definite **după** aceasta.

Definirea funcțiilor (continuare)

> Vizibilitatea variabilelor în cadrul funcțiilor

Exemplu:

```
#include<stdio.h>
float produs(float a, float b)
{
    int offset=4;
    return a*b+offset;
}

int offset=3;

int main()
{
    printf("%d, %f", offset, produs(5,6) );
}
```

-ce valoare va returna funcția **produs** ???

>3, 30+4=34.00

-în funcție, variabila **offset** este definită local și are valoarea 4.

-în **main()**, variabila **offset** este cea definită **global**, cea din funcție neexistând decât în interiorul fct. **produs** (execuție).

Definirea funcțiilor (continuare)

> Vizibilitatea variabilelor în cadrul funcțiilor

Exemplu:

```
#include<stdio.h>
int offset=3;

float produs(float a, float b)
{
    int offset=4;
    return a*b+offset;
}

int main()
{
    printf("%d, %f, %d", offset, produs(2,3), offset );
}
```

-ce valoare va returna funcția **produs** ???

>3, 2*3+4=10.00, 3

-cu toate că variabila **offset** este definită global și este vizibilă și în funcția **produs**, redefinirea ei local, duce la ignorarea variabilei globale.

Definirea funcțiilor (continuare)

P *Enunț:* să se realizeze o funcție care permite afișarea pe ecran a unei matrice pătratice de numere întregi. Matricea se va afișa pe linii și coloane.

Variabile de intrare/lucru:

```
int M[100][100];
int dim;
int i, j;
void AfisareM(int dim, int M[100][100]);
```

Variabile de ieșire:

nu

program principal
main()

definiție funcție
AfisareM()

Definirea funcțiilor (continuare)

```
#include<stdio.h>
#include<stdlib.h>
/* procedura este definita inaintea functiei main */
void AfisareM(int dim, int M[100][100])
{
    int i,j;
    printf("\n");
    for (i=0; i<dim; i++) // parcurgere linii
    {
        for (j=0; j<dim; j++) // parcurgere elemente de pe
            printf("%8d",M[i][j]); // coloane
        printf("\n");
    }
}
```

Definirea funcțiilor (continuare)

```
int main()
{
    int M[100][100], i, j, dim;
    printf("dim="); scanf("%d",&dim);
    for (i=0; i<dim; i++)
        for (j=0; j<dim; j++)
        {
            printf("M[%d][%d]=",i,j);
            scanf("%d",&M[i][j]);
        }
    AfisareM(dim, M);
}
```

-în felul acesta funcția AfisareM poate fi folosită ori de câte ori avem nevoie să afișăm pe ecran o matrice pătratică.

7.2. Recursivitate

Funcții recursive

> **Recursivitatea** unei funcții în programare, reprezintă procesul prin care funcția se autoapelează.

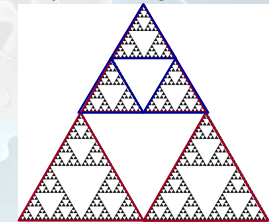
> Pentru ca procesul să nu se repete la infinit, autoapelarea se încheie de regulă pe baza unei condiții de oprire.

Efectul Droste:



etc.

Sierpinski triangle:



Funcții recursive (continuare)

> Astfel, un obiect sau un fenomen se definește în mod recursiv dacă în definiția sa există o referire la el însuși.

Exemple:

definirea numerelor întregi:

$$\begin{cases} 0 \in \mathbf{N} \\ \text{dacă } i \in \mathbf{N} \text{ atunci } i+1 \in \mathbf{N} \end{cases}$$

definirea funcției factorial:

$$n! = \begin{cases} 1 & \text{dacă } n = 0 \\ n \cdot (n-1)! & \text{dacă } n > 0 \end{cases}$$

definirea șirului Fibonacci:

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

$$fib(n) = \begin{cases} 0 & \text{dacă } n = 0 \\ 1 & \text{dacă } n = 1 \\ fib(n-1) + fib(n-2) & \text{dacă } n > 1 \end{cases}$$

Funcții recursive (continuare)

> O funcție recursivă trebuie să fie **bine formată**:

- o funcție nu se poate defini doar în funcție de sine însăși,

exemplu: $f(n) = 2 + f(n) \rightarrow$ greșit, se execută la ∞

- o funcție recursivă se poate folosi numai de noțiuni deja definite anterior,

exemplu: $f(n) = f(n+1) - 2 \rightarrow$ greșit (valoare inexistentă $f(n+1)$),

- orice șir de apeluri de funcții recursive trebuie să se oprească (nu va genera un calcul infinit).

exemplu: $f(0) = 1, f(n) = f(n-1) + 3, \rightarrow$

$f(3) = f(2) + 3 = f(1) + 3 + 3 = f(0) + 3 + 3 + 3 = 1 + 3 + 3 + 3 = 10$

Funcții recursive (continuare)

> În general distingem:

- **un caz de bază:** pentru care funcția este definită direct, de exemplu: $f(0) = 1$

- **un pas inductiv** (recursivitatea propriu-zisă): funcția este definită folosind aceeași funcție, dar pe un caz mai simplu, de exemplu: $f(n+1) = f(n) \cdot n!$

> În limbajul C crearea de funcții recursive nu necesită o anumită sintaxă sau folosirea unui anumit cuvânt cheie, aceasta reiese automat din modul de definire al funcției.

Funcții recursive (continuare)

Exemplu: funcția factorial

```
int Factorial(int n)
{
    if (n==0)
        return 1;
    else
        return n*Factorial(n-1);
}
```

- funcția primește valoarea **n** și returnează valoarea calculată a factorialului.

- **condiția de oprire** a recursivității: dacă **n** a ajuns la valoarea 0, se returnează 1 (0!).

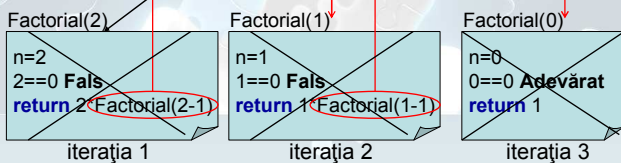
- **pasul inductiv:** relația de calcul recursiv ce depinde de valorile de rang inferior ale funcției: Factorial(n-1), Factorial(n-2) ...

Funcții recursive (continuare)

Modul de execuție

```
int main()
{
    printf("%d", Factorial(2));
}
```

```
int Factorial(int n)
{
    if (n==0)
        return 1;
    else
        return n*Factorial(n-1);
}
```



> Funcția returnează valoarea: $2 * 1 * 1$

Funcții recursive (continuare)

Exemplu: o funcție recursivă care numără de la a la b în sens crescător, și invers.

```
void printnum(int a, int b)
{
    printf("%d ", a);
    if (a < b)
        printnum(a+1, b);
    printf("%d ", a);
}
```

- funcția primește valoarea **a** și respectiv **b** și nu returnează nimic (rezultatul este afișat pe ecran)

- se afișează valoarea lui **a** și apoi se apelează recursiv funcția pentru a afișa în ordine crescătoare valorile.

- profită de închiderea în sens invers a procedurilor lansate pentru diverse valori ale lui **a** pentru afișarea în ordine inversă.

Funcții recursive (continuare)

Mod de execuție:

```
int main()
{
    printnum(2,4);
}
```

```
void printnum(int a, int b)
{
    printf("%d ",a);
    if (a<b)
        printnum(a+1,b);
    printf("%d ",a);
}
```

printnum(2,4)
printnum(3,4)
printnum(4,4)

iterația 1 iterația 2 iterația 3

Curs Programarea Calculatoarelor, Prof. Bogdan IONESCU, 2016-2017 30/36

Funcții recursive (continuare)

P Se dă o imagine binară (valori 0 și 1) sub forma unei matrice bidimensionale. În aceasta, obiectele sunt reprezentate cu valori de 1, iar fundalul cu valori de 0. Să se realizeze o funcție recursivă care permite calculul suprafeței unui obiect (metoda "flood fill").

Exemplu:

```
0 0 0 1
0 1 0 0
1 1 1 0
0 0 1 1
```

—obiect de suprafață 1
suprafață ~ număr valori de 1
obiect = mulțime de valori de 1 vecine (conexe).

Curs Programarea Calculatoarelor, Prof. Bogdan IONESCU, 2016-2017 31/36

Funcții recursive (continuare)

Problemă obiecte (continuare)

> Pentru a determina care valori de 1 sunt vecine, și astfel aparțin obiectului curent, se va folosi conectivitatea cu 4 vecini:

4 conectivitate

```
1 0 0 1
0 1 0 0
1 1 1 0
0 0 1 1
```

câte obiecte sunt în imagine?

Curs Programarea Calculatoarelor, Prof. Bogdan IONESCU, 2016-2017 32/36

Funcții recursive (continuare)

Problemă obiecte (continuare)

Strategie: "flood fill"

```
int matrice[20][20];
int dim;
int suprafata; (var.globală)
```

```
void SuprafataObiect(int matrice[20][20], int dim, int x, int y)
```

- nu returnează nimic, suprafața este stocată global
- primește matricea pentru a verifica valorile
- primește **dim** pentru a nu ieși din matrice la calcul
- (x,y) indică poziția curentă analizată din matrice

Curs Programarea Calculatoarelor, Prof. Bogdan IONESCU, 2016-2017 33/36

Funcții recursive (continuare)

Problemă obiecte (continuare)

```
void SuprafataObiect(int matrice[20][20], int dim, int x, int y)
{
    if ((matrice[x][y]==0) || (matrice[x][y]==-1)) return;
    suprafata++; matrice[x][y]=-1;
    if (x+1<dim)
        if (matrice[x+1][y]==1)
            SuprafataObiect(matrice, dim, x+1,y);
    if (y+1<dim)
        if (matrice[x][y+1]==1)
            SuprafataObiect(matrice, dim, x,y+1);
    if (x-1>=0)
        if (matrice[x-1][y]==1)
            SuprafataObiect(matrice, dim, x-1,y);
    if (y-1>=0)
        if (matrice[x][y-1]==1)
            SuprafataObiect(matrice, dim, x,y-1);
}
```

- condiție de oprire: (x,y) fundal sau (x,y) a fost deja luat în calcul (-1)
- (x,y) este luat în calcul la suprafață și apoi marcat,
- vecin Sud
- vecin Est
- vecin Nord
- vecin Vest

Curs Programarea Calculatoarelor, Prof. Bogdan IONESCU, 2016-2017 34/36

Funcții recursive (continuare)

Problemă obiecte (continuare)

Mod de execuție:

```
int Suprafata;
...
int main()
{
    int dim, matrice[20][20];
    ...
    Suprafata=0;
    SuprafataObiect(matrice, dim, 1,1);
}
```

x=1, y=1 x=2, y=1 x=2, y=2 x=3, y=2 x=2, y=0

Suprafata=1 Suprafata=2 Suprafata=3 Suprafata=4 Suprafata=5

lansare Sud lansare Sud lansare Sud lansare Sud lansare Sud

lansare Est lansare Est lansare Est lansare Est lansare Est

lansare Nord lansare Nord lansare Nord lansare Nord lansare Nord

lansare Vest lansare Vest lansare Vest lansare Vest lansare Vest

Curs Programarea Calculatoarelor, Prof. Bogdan IONESCU, 2016-2017 35/36



Sfârșitul Cursului 7