

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



Programarea Calculatoarelor (limbajul C)

Curs 2 – Limbaje de programare

Prof. Bogdan IONESCU

2016-2017

Cuprins

- 2.1. Rezolvarea unei probleme cu ajutorul calculatorului
- 2.2. Sintaxa limbajelor de programare
- 2.3. Etapele dezvoltării unui program

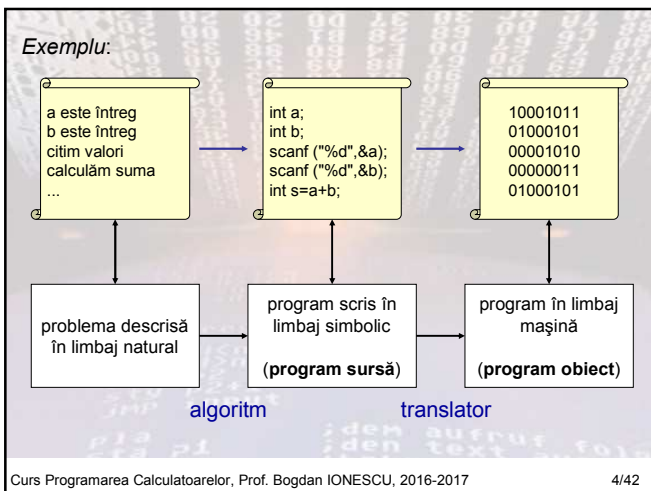
Curs Programarea Calculatoarelor, Prof. Bogdan IONESCU, 2016-2017 1/42

2.1. Rezolvarea unei probleme cu ajutorul calculatorului

Curs Programarea Calculatoarelor, Prof. Bogdan IONESCU, 2016-2017 2/42

- > O problemă de calcul este întotdeauna descrisă inițial de către programator într-un *limbaj natural* (apropiat de cel uman),
- > Rezolvarea acesteia cu ajutorul calculatorului implică traducerea **algoritmului** aferent într-un *limbaj simbolic* ce poate fi interpretat de sistemul de calcul,
- > Programul scris în limbaj simbolic este în final tradus în *limbaj mașină* pentru a putea fi înțeles și executat de sistemul de calcul,

Curs Programarea Calculatoarelor, Prof. Bogdan IONESCU, 2016-2017 3/42



Program translator = este un program special care face conversia între limbajul simbolic și limbajul mașină.

Poate fi:

- **asamblor**: translatorul unui *limbaj de asamblare* în cod mașină, operația de traducere poartă numele de asamblare,
- **compiler**: translatorul unui *program sursă* scris într-un limbaj de nivel înalt, în cod mașină sau într-un limbaj apropiat de acesta (de exemplu limbajul de asamblare "assembler"),
- **interpretor**: translatorul unui *program sursă* scris într-un limbaj de nivel înalt într-un cod intermediar mai eficient care este executat imediat,

Curs Programarea Calculatoarelor, Prof. Bogdan IONESCU, 2016-2017 5/42

- **compilator incremental**: este parte a unui program în curs de execuție și se folosește de compilator. Acesta permite noilor componente ale programului să fie compilate în orice moment al execuției,
 - extindere sau eliminare părți program.
 - **compilator încrucișat** ("cross compiler"): translator ce rulează pe un anumit tip de sistem de calcul dar care generează cod obiect pentru un alt tip de calculator,
 - necesar în cazul în care nu se poate executa programul compilator pe sistemul de destinație, ex.: microcontroller.
- > Traducerea unui *program sursă* se face păstrând semantica (înțelesul) programului astfel încât să se stabilească o *relație de echivalență* între programul sursă și programul obiect.

Crearea unui program executabil implică următoarele etape:

1. Programul sursă (cod) este **compilat** în **program obiect** (în cazul în care nu există erori de sintaxă)

> De regulă programul sursă nu conține toate secvențele de comenzi necesare, acesta poate reutiliza părți de program scrise anterior, ex.: disponibile în *biblioteci de funcții*.

> Mai mult, un program complex implică mai multe fișiere de cod (mai lizibil și astfel mai ușor de modificat).

Programul sursă = ansamblu de *fișiere sursă* ce trebuie compilate separat pentru obținere *fișiere obiect*

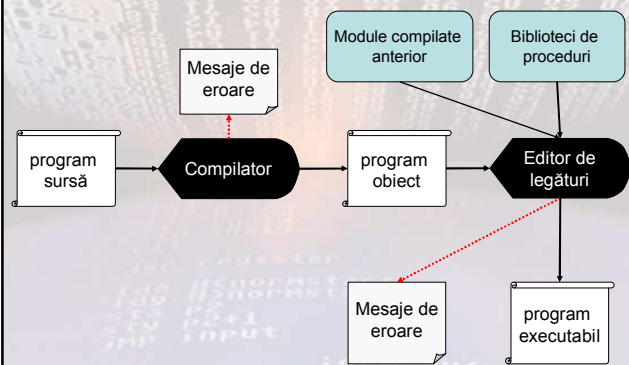
Avantaj: reducere importantă a timpului de compilare ?

→ pe parcursul dezvoltării unui program, în momentul unei compilări, nu toate fișierele sursă au fost modificate, astfel vor fi compilate doar fișierele ce prezintă modificări.

Atenție: compilatorul ca program este dedicat unui anumit limbaj de programare și unui anumit tip de sistem de calcul (modul de execuție și accesul la memorie fiind diferite)

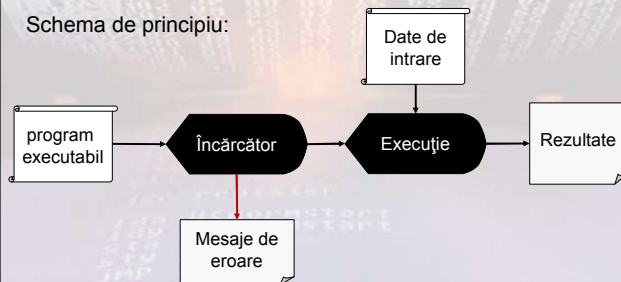
2. Programele obiect astfel obținute nu pot fi executate de sistem în forma aceasta, se va genera un program executabil folosind **editorul de legături**.

Schema de principiu crearea program executabil (rezumat):



3. Executarea programului este realizată prin *încărcarea* acestuia, trecerea din memoria externă (ex. HDD) în memoria internă (ex. RAM), și *executarea* acestuia.

Schema de principiu:



> Programele executabile obținute în acest fel pe o anumită platformă (sistem de operare) pot rula pe alte platforme ?

Portabilitatea = posibilitatea de a executa un program scris într-un anumit limbaj pe un alt tip de sistem de calcul fără a fi necesară rescrierea codului.

→ în cazul cel mai simplu programul necesită doar recompilarea pe noua platformă.

→ în realitate aceasta este o problemă destul de dificil de soluționat în cazul general.



2.2. Sintaxa limbajelor de programare

Din punct de vedere *ierarhic*, un program C conține următoarele informații:

- > **program principal**: acesta corespunde programului executabil (main),
- > **subprograme** (funcții): se pot rula de mai multe ori pe parcursul programului, returnează anumite valori ...
- > **instrucțiuni**

- > **declarații**: ex.: variabile, constante
- > **comenzi**: folosesc variabile, constante, operatori, apeluri de funcții ...

Exemplu program simplu:

```
#include <stdio.h>

int main(void)
{
    printf("hello, world");
    return 0;
}
```

sintaxă: mulțimea regulilor care descriu modul de alcătuire al programului

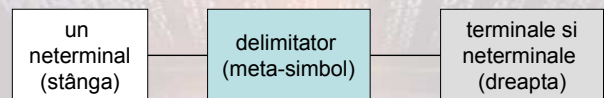
```
tip Nume(variabile)
{
    // corp program
}
```

semantică: mulțimea regulilor care definesc înțelesul fiecărui program

```
printf = tipărire pe
dispozitivul de ieșire
```

> Descrierea exactă a regulilor sintactice ale unui limbaj de programare, se face în general folosind forma BACKUS – NAUR (BNF) = *metalimbaj* (**independent de limbajul de programare**).

→ producții



neterminale: componente ale programului (instrucțiuni, expresii, variabile, etc.), cărora li se descrie sintaxa, modul în care vor apare formulate în program.

Formal este reprezentat de un șir de caractere încadrat de "<" și ">", ex.: <structura_if>

delimitatorul: are simbolul ::= , cu semnificația "este definit prin".

terminalele: sunt caractere sau șiruri de caractere predefinite, a căror sintaxă rezultă din însuși modul în care apar în definiție, ex.: if, "(", ")", "{", "}", "[", "]", etc.

Exemple:

```
<tip_simplu> ::= int | float | char
```

tip simplu este definit prin *int* sau *float* sau *char*

```
<structura_if> ::= if "(" <condiție> ")" "{" <cod> "}"
```

structura if este definită prin if urmat de o **condiție** specificată între () urmată de un bloc de **cod** încadrat de { }

2.3. Etapele dezvoltării unui program

> Dacă dezvoltarea unui program simplu poate fi realizată direct pe sistemul de calcul, un program complex implică o activitate sistematică de **cercetare-producție**

→ *tehnologie a programării*

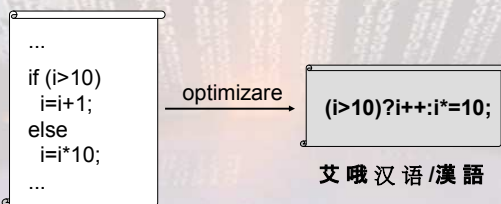
- procese,
 - metode,
 - tehnici,
 - instrumente,
- ce duc la obținerea de programe cu un **nivel calitativ cât mai înalt și cu un preț cât mai coborât**

preț cât mai coborât = (evident) preț de realizare redus, preț de distribuție redus, etc.

nivel calitativ înalt = realizarea obiectivului programului cu un factor de merit cât mai ridicat, acesta cuprinde:

- un număr mic de linii sursă,
- un timp cât mai scăzut pentru parcurgerea etapelor de dezvoltare ale programului,
- folosirea cât mai rațională a resurselor sistemului de calcul (timp scurt de compilare, timp scurt de execuție spațiu de memorie utilizat redus, etc.)
- precizie suficientă a rezultatelor
- furnizarea unei interfețe intuitive, naturale

> O parte din cerințe sunt uneori subiective, ex.: un număr redus de linii de cod poate implica un program greu de înțeles,



“Adevărul este la mijloc”: ideal se încearcă găsirea unui compromis care să împace toate cerințele de implementare.

> Pentru a dezvolta un program eficient, programatorul trebuie **să parcurgă o serie de etape**, pornind de la enunțarea problemei într-un limbaj natural, și ajungând în final la codul sursă al programului:

- **A. să analizeze problema** de rezolvat stabilind specificația programului,
- **B. să proiecteze programul**, adică să elaboreze mai întâi **algoritmul** care va realiza funcția dorită,
- **C. să implementeze programul** = editare cod, depanare cod (corectare erori) și alcătuire documentație program,
- **D. să întrețină programul** (upgrade, noi cerințe, etc.)

A. Analiza problemei → specificațiile programului.

- **lista variabilelor de intrare și de ieșire**: în funcție de problema de calcul se face o estimare a variabilelor necesare precum și o descriere a acestora,

*Exemplu: x număr întreg (introdus de la tastatură)
y real preia rezultatul final, etc.*

- **funcția programului**: corelația dintre intrare și ieșire, ce va face efectiv programul, cum va prelucra datele,
- **organizarea datelor de intrare și de ieșire**: ce date sunt introduse și cum, ce date sunt afișate sau folosirea altor dispozitive de intrare/ieșire.

Analiza problemei (continuare)

P *Enunț*: fiind date două numere pozitive, să se calculeze: media aritmetică, media geometrică și media armonică.

variabile de intrare: a, b, de tip real

variabile de ieșire: m_arit, m_geom, m_arm de tip real

programul calculează: $m_arit = (a + b) / 2$
 $m_geom = \sqrt{a * b}$
 $m_arm = 2 / (1/a + 1/b)$

cum *introduc/returnez* datele:

*datele de intrare de la tastatură
datele de ieșire în fișier de date*

B. Proiectarea programului

> Se caută un **algoritm**, cât mai eficient, care să rezolve cât mai repede (timp de calcul redus) problema de calcul vizată.

> Algoritmii sunt specificați într-un limbaj natural, dar destul de apropiat de limbajul în care va fi implementat programul (din punct de vedere al semnificației).

> **Algoritmii simpli** → pot fi redactați direct fără prea multă bătaie de cap, implică o singură "procedură",

> **Algoritmii complexi** → este foarte probabil ca algoritmul să nu poată fi scris de la început, problema trebuind mai întâi analizată în detaliu,

Proiectarea programului (continuare)

Exemplu 1, algoritmul pentru problema anterioară este unul simplu:

1. **citește** valorile variabilelor **a** și **b**
2. **calculează**:
 - 2.1. **atribuie** lui **m_arit** valoarea expresiei $(a+b)/2$
 - 2.2. **atribuie** lui **m_geom** valoarea expresiei $\sqrt{a*b}$
 - 2.3. **atribuie** lui **m_arm** valoarea expresiei $2/(1/a+1/b)$
3. **scrie** datele în fișier
4. **stop**

operațiile au o ordine

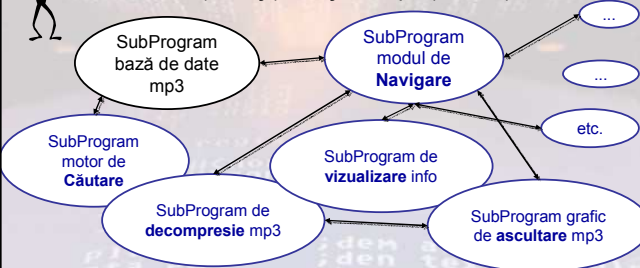
cuvintele cheie corespund instrucțiunilor existente

R *Enunț*: fiind date două numere pozitive, să se calculeze: media aritmetică, media geometrică și media armonică.

Proiectarea programului (continuare)

Exemplu 2, algoritmul unei probleme mai complexe nu poate fi scris așa ușor, acesta necesită alte etape de analiză,

Enunț: să se implementeze un program de accesare a unei baze (colecții) de fișiere mp3 (muzică).



Proiectarea programului (continuare)

> Pentru programe complicate trebuie adoptată o altă tactică ce presupune:

- descompunerea problemei de rezolvat în **subprobleme** cu o complexitate mult mai redusă (de regulă se ocupă de un **obiectiv unic**, ex.: decompresia mp3, citire pachet de date de intrare, etc.)

- enunțarea fiecărei subprobleme,
- specificația fiecărei subprobleme (variabile, etc.),
- **modul de interacție** cu celelalte subprobleme,
- eventuală descompunere în subsubprobleme (specificarea diverselor funcții),

Proiectarea programului (continuare)

- dezvoltarea **algoritmilor de rezolvare** a subproblemele folosind o abordare similară celei prezentate anterior,

Atenție: pentru elaborarea eficientă a algoritmului este necesară cunoașterea posibilităților de programare ale limbajului în care se va dezvolta programul.

Exemplu: program în JAVA - atenție programare obiect orientată, program în C - atenție necesită program principal.

> Există totuși o serie de modalități consacrate de specificare a algoritmilor = **tehnici de descriere**

pseudocodul

ȘI / SAU

schemele logice

Proiectarea programului (continuare)

Pseudocodul = *metalimbaj*, adică o combinație între un limbaj de programare de nivel înalt și un limbaj natural.

> Nu este un limbaj standardizat cu o sintaxă bine precizată.

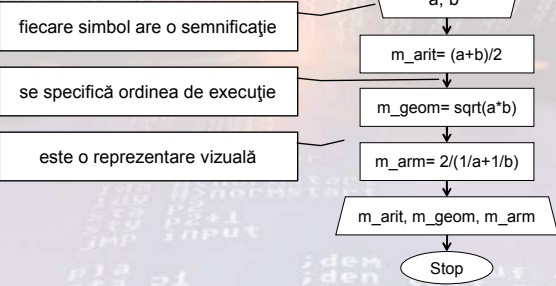
Exemplu:

```
citește a, b;  
atribuie m_arit ← (a+b)/2;  
atribuie m_geom ← sqrt(a*b);  
atribuie m_arm ← 2/(1/a+1/b);  
deschide fișier f, "c:\demo\data.txt";  
scrie în fișier m_arit, m_geom, m_arm;  
închide fișier f;  
stop ;
```

Proiectarea programului (continuare)

Schemele logice = reprezentări grafice ale algoritmilor (sub forma unei diagrame logice).

Exemplu:



Proiectarea programului (continuare)

Care tehnică de descriere este mai bună, pseudocodul sau schemele logice ?



Soluția: folosirea ambelor tehnici, **schema logică** → execuție algoritim **pseudocod** → funcții și cod

Schemele logice

Avantaje (pros):

- indică modul de execuție,
- ușor de urmărit,
- descriere vizuală (interes teoretic),

Dezavantaje (cons):

- program complex = schemă logică mult prea mare, dificil de realizat,
- limitarea simbolurilor

Pseudocodul

Avantaje (pros):

- limbaj apropiat de codul final,
- flexibilitate a descrierii,
- limbaj apropiat de cel natural,

Dezavantaje (cons):

- mai puțin lizibil (mai greoi),
- posibil să complice anumite descrieri ale unor probleme complexe

C. Implementarea programului

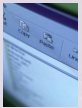
> **În primă fază** se editează (scrie, corectează, modifică, etc.) fișierul sursă al programului.

Se poate opta pentru:



Ediția rapidă într-un editor de text clasic, ex.: Edit (MS. Dos), Notepad (MS. Windows), etc.

→ *programatori experimențați*: rapid dar dificil (lipsă asistență, lipsă interfață ...)



Ediția într-un **mediu de dezvoltare**, ex.: Dev-C++, Borland C++, Visual Studio ...

→ interfață grafică vizuală specifică, asistență editare cod, pachet de utilitare (erori, structură,...), ...

Implementarea programului (continuare)

Implementarea programului (continuare)

Exemplu de program sursă în C:

```

// calculul mediilor a doua numere reale pozitive
#include<stdio.h>
#include<math.h>
void main(void)
{
    float a,b, fMAritm, fMGeom, fMArm;
    printf("introduceti valoarea lui a="); scanf("%f", &a);
    printf("introduceti valoarea lui b="); scanf("%f", &b);

    fMAritm=(a+b)/2;
    fMGeom=sqrt(a*b);
    fMArm=2/(1/a+1/b);

    printf("Mediile sunt: aritmetica=%f, geometrica=%f, armonica=%f",
           fMAritm, fMGeom, fMArm);
}
    
```

Labels in image:

- antet program (points to // comment)
- biblioteca de funcții (points to #include lines)
- program principal (main) (points to void main)
- delimitat de "{" și "}" (points to curly braces)

Implementarea programului (continuare)

```

// calculul mediilor a doua numere reale pozitive
#include<stdio.h>
#include<math.h>
void main(void)
{
    float a,b, fMAritm, fMGeom, fMArm;

    printf("introduceti valoarea lui a="); scanf("%f", &a);
    printf("introduceti valoarea lui b="); scanf("%f", &b);

    fMAritm=(a+b)/2;
    fMGeom=sqrt(a*b);
    fMArm=2/(1/a+1/b);

    printf("Mediile sunt: aritmetica=%f, geometrica=%f, armonica=%f",
           fMAritm, fMGeom, fMArm);
}
    
```

Labels in image:

- zonă de declarații variabile (points to float declaration)
- zonă de comenzi (points to printf and calculation lines)

Implementarea programului (continuare)

```
// calculul mediilor a doua numere reale pozitive
#include<stdio.h>
#include<math.h>

void main(void)
{
    float a,b, fMAritm, fMGeom, fMArm;
    printf("introduceti valoarea lui a="); scanf("%f", &a);
    printf("introduceti valoarea lui b="); scanf("%f", &b);

    fMAritm=(a+b)/2;
    fMGeom=sqrt(a*b);
    fMArm=2/(1/a+1/b);

    printf("Mediile sunt: aritmetica=%f, geometrica=%f, armonica=%f",
        fMAritm, fMGeom, fMArm);
}
```

specificați în numele variabilei și tipul (f=float)

o comandă este pe doua linii, problemă ?

Implementarea programului (continuare)

> **A doua fază** constă în corectarea eventualelor erori existente:

- **erori sintactice**: rezultat al scrierii incorecte a instrucțiunilor programului, ex.: nerespectarea sintaxei, nerespectarea regulilor de punctuație, lipsa sfârșitului de linie ";", etc.

→ sunt cele mai frecvente

- **erori la editarea legăturilor**: apar în special prin utilizarea unor funcții predefinite pentru care nu există descriere (fie nu există, fie nu am specificat biblioteca)

→ relativ simplu de corectat

Implementarea programului (continuare)

- **erori de execuție**: erorile care apar abia în momentul rulării programului. Cauzele sunt greșelile în modalitatea de implementare a programului, ex.: alocare necorespunzătoare a memoriei, accesare zone de memorie indisponibile, operații ilegale, etc.)

→ destul de dificil de corectat, de regulă se apelează la utilitare specifice ⇔ "debugger"

- **erori logice**: sunt erori de algoritm, programul rulează corect (nu sunt semnalate erori), dar rezultatele obținute sunt greșite.

→ **cel mai dificil de corectat**, trebuie verificat algoritmul pas cu pas folosind un set de date cunoscute (se știe reacția programului la acestea)

Implementarea programului (continuare)

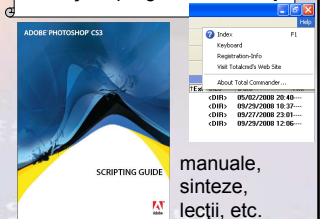
> **Ultima etapă** constă în alcatuirea documentației programului, este esențială pentru lizibilitatea și înțelegerea programului.

în cod

```
/*-----
Program calcul
-----*/
...
// declaratii
int a,b;
...
// s este calculat ca fiind...
s=(1/b)*exp(-2*sigma^2*a);
...
```

separată

disponibilă pe durata de execuție a programului = help



manuale, sinteze, lecții, etc.

D. Întreținerea programului

> Este un proces ce se desfășoară pe toată durata de existență a programului și constă în modificarea sistematică a acestuia.

Întreținerea implică:



- corectarea erorilor apărute ulterior (bugs)

feedback utilizatori



- adaptarea programului la noi cerințe (ex. alt sistem) pachete de **update**



- dezvoltarea de versiuni ulterioare care să includă noi prelucrări

prototip → **alpha** (ver. internă) → **beta** (ver. test) → **RC** (release candidate) → **finală v2.1**.

Dezvoltarea unui program (sinteză)

> **A. analiza problemei**

→ variabile, funcții realizate, date in-out,

> **B. proiectarea programului**

→ descriere algoritm,

> **C. implementarea programului**

→ redactare într-un limbaj, depanare ...

> **D. întreținerea programului**

→ adaptare, modificare,

Sfârșitul Cursului 2